AutoCorres2

Matthew Brecknell David Greenaway Johannes Hölzl
Fabian Immler Gerwin Klein Rafal Kolanski
Japheth Lim Michael Norrish Norbert Schirmer
Salomon Sickert-Zehnter Thomas Sewell Harvey Tuch
Simon Wimmer

March 17, 2025

Abstract

AutoCorres2 is a tool to facilitate the verification of C programs within Isabelle [5]. It is a fork of AutoCorres.

Contents

1 Introduction **11**

I Library **12**

2 Misc. Definitions and Lemmas **13**
2.1 Take, drop, zip, *list-all* etc rules 48

3 ML Antiquotations **56**
3.1 Building terms: *mk-term* 56
3.2 Term pattern: *term-pat* 57
3.2.1 Example 58
3.3 ML Antiquotation to Match and Instantiate (recombine) cterms and terms 60
3.4 Record Antiquotation 61
3.4.1 Motivation 61
3.4.2 Example 62
3.4.3 Implementation 62
3.4.4 Examples 62
3.5 Various Antiquotations 62
3.5.1 Antiquotations for terms with schematic variables . 62
3.5.2 Antiquotation for types with schematic variables . . 63

4 Proof Tools **64**
4.1 Tools for handling Tuples 64
4.1.1 Antiquotations for terms with schematic variables . 64
4.2 Tools for intro-rule based term synthesis *synthesize-rules* . . . 67
4.2.1 Commands 67
4.2.2 ML Antiquotations 67
4.2.3 Attributes 67

5 Rule by Method **68**
5.1 Tagging 70
5.1.1 Basic Definitions and Theorems 70
5.1.2 Conversions 71

5.1.3 Globbing . . . 72
5.1.4 Reordering Subgoals . . . 72
5.1.5 *subgoalT* and *subgoalsT*, and *prefersT* . . . 72
5.1.6 Syntax . . . 73

6 Verification Condition Generator *runs-to-vcg* **75**
6.1 Marked Assumptions . . . 75
6.2 *THEN-ALL-NEW-FORWARD* . . . 75
6.3 Basic VCG . . . 75
6.4 Setup for Tagging . . . 76
6.5 *runs-to-vcg* . . . 77
6.6 Check . . . 77

7 Proof Methods **78**
7.1 Debugging methods . . . 78
7.2 Simple Combinators . . . 78
7.3 Advanced combinators . . . 79
7.3.1 Protecting goal elements (assumptions or conclusion) from methods . . . 79
7.3.2 Safe subgoal folding (avoids expanding meta-conjuncts) 80
7.4 Utility methods . . . 81
7.4.1 Finding a goal based on successful application of a method . . . 81
7.4.2 Remove redundant subgoals . . . 81
7.5 Attribute methods (for use with *rule-by-method* attributes) . 82
7.6 Shortcuts for *prove-prop*. . . . 82

8 Option Monad (State Reader) **85**
8.1 "While" loops over option monad. . . . 91
8.2 Lift *option-while* to the ('*a*, '*s*) *lookup* monad . . . 92

9 Mutual CCPO Recursion *fixed-point* **96**
9.1 Relate orders between locale and type classes . . . 96
9.2 Prove admissibility of *corresXF* . . . 98

II C-Parser **101**

10 Theory Variants for Target Architectures via *L4V-ARCH* **102**

11 Unified Memory Model (UMM) **103**
11.1 More Word Lemmas . . . 103
11.2 Distinct Proposition . . . 115
11.3 Type setup . . . 123
11.3.1 Pointers . . . 124

11.3.2 Raw heap . . . 125
11.4 Intervals . . . 125
11.5 *dt-tuple*: a reimplementation of 3 item tuples . . . 126
11.6 Properties of pointers . . . 126
11.7 Properties of the raw heap . . . 127
11.8 Properties of intervals . . . 127
11.9 C types setup . . . 130
11.10 *super-update-bs* . . . 163
11.11 Rest . . . 164
11.12 Words and Pointers . . . 201
11.13 Instantiation, inferring type instantiation from term instantiation. . . . 214
11.14 Finite Cartesian Products . . . 216
11.15 Auxiliary Theorems . . . 221
11.16 Legacy definition *fg-cons* . . . 222
11.17 Lense definition using an update function *lense* . . . 222
11.18 Update for functions . . . 224
11.19 Scenes . . . 224
11.20 More properties of maps plus map disjuction. . . . 290
11.20.1 Properties of maps not related to restriction . . . 291
11.20.2 Properties of map restriction . . . 292
11.21 Definitions . . . 294
11.22 Properties of ($'-$) . . . 295
11.23 Properties of map disjunction . . . 295
11.23.1 Map associativity-commutativity based on map disjuction . . . 295
11.23.2 Basic properties . . . 296
11.23.3 Map disjunction and addition . . . 296
11.23.4 Map disjunction and updates . . . 298
11.23.5 Map disjunction and ($\subseteq_m$) . . . 298
11.23.6 Map disjunction and restriction . . . 299
11.24 *sep-point-tac* . . . 365
11.25 *sep-exists-tac* . . . 366
11.26 *sep-select-tac* . . . 367
11.27 (Partial) Pointer Lenses . . . 405
11.27.1 *pointer-lense* . . . 405
11.27.2 *partial-pointer-lense* . . . 405
11.28 *heap-upd* and *heap-upd-list* . . . 469
11.29 *heap-upd* . . . 469
11.30 *merge-ti* . . . 471
11.30.1 *merge-ti-list* . . . 473

12 More Building Blocks for our C-Language Model 475
12.1 addr bounds . . . 475

12.2 More Heap Typing . . . 475
12.2.1 Heap type tag and valid simple footprint . . . 477
12.3 Pointers to local (stack) variables . . . 480
12.4 Misc derived language elements . . . 512

13 Packed Types (no implicit padding) 526
13.1 Underlying definitions for the class axioms . . . 526
13.2 Lemmas about *td-fafu-idem* . . . 527
13.2.1 *td-fa-hi* . . . 529
13.3 Simp rules for deriving packed props from the type combinators 529
13.3.1 *td-fafu-idem* . . . 529
13.3.2 *td-fa-hi* . . . 531
13.4 The type class and simp sets . . . 533
13.5 Instances . . . 534
13.6 Theorems about packed types . . . 534
13.6.1 *td-fa-hi* . . . 534
13.6.2 *td-fafu-idem* . . . 535
13.6.3 Proof automation for packed types . . . 536
13.7 Prettier Printing for Programs . . . 538
13.8 Modelling Local Variables . . . 545

14 Setup Lex / Yacc and Translation from C to Simpl 550

15 Misc. Lemmas 562
15.1 Abbreviations and helpers . . . 562
15.2 Basic operations . . . 562
15.2.1 clift . . . 562
15.2.2 *h-val* . . . 563
15.2.3 *h-t-valid* . . . 563
15.3 *field-lvalue* . . . 563
15.3.1 *heap-update* . . . 563
15.3.2 *c-guard* . . . 564
15.3.3 clift . . . 564
15.3.4 cparent . . . 565
15.3.5 *h-val* . . . 566
15.3.6 *h-t-valid* . . . 567
15.3.7 Type Combinators and Padding . . . 568
15.3.8 The orphanage: miscellaneous lemmas pulled up to (roughly) where they belong. . . . 569

III AutoCorres 579

16 Spec-Monad 580

16.1 *rel-map* and *rel-project* . . . 580
16.2 Misc Theorems . . . 581
16.3 Galois Connections . . . 582
16.4 *post-state* type . . . 582
16.4.1 Order Properties . . . 585
16.4.2 *holds-post-state* . . . 585
16.4.3 *holds-post-state-partial* . . . 587
16.4.4 *sim-post-state* . . . 588
16.4.5 *rel-post-state* . . . 590
16.4.6 *lift-post-state* . . . 591
16.4.7 *map-post-state* . . . 591
16.4.8 *vmap-post-state* . . . 592
16.4.9 *pure-post-state* . . . 593
16.4.10 *bind-post-state* . . . 594
16.5 *exception-or-result* type . . . 596
16.6 *spec-monad* type . . . 602
16.6.1 *rel-spec-monad* . . . 614
16.7 VCG basic setup . . . 615
16.7.1 *res-monad* and *exn-monad* Types . . . 616
16.8 *res-monad* and *exn-monad* functions . . . 623
16.9 Monad operations . . . 624
16.9.1 ⊤ . . . 624
16.9.2 ⊥ . . . 624
16.9.3 *fail* . . . 624
16.9.4 *yield* . . . 625
16.9.5 *throw-exception-or-result* . . . 626
16.9.6 *throw* . . . 626
16.9.7 *get-state* . . . 626
16.9.8 *set-state* . . . 626
16.9.9 *select* . . . 627
16.9.10 *unknown* . . . 628
16.9.11 *lift-state* . . . 628
16.9.12 const*exec-concrete* . . . 628
16.9.13 const*exec-abstract* . . . 629
16.9.14 *bind-exception-or-result* . . . 630
16.9.15 *bind-handle* . . . 631
16.9.16 (≫=) . . . 634
16.9.17 *assert* . . . 642
16.9.18 *assume* . . . 642
16.9.19 *assume-outcome* . . . 643
16.9.20 *assume-result-and-state* . . . 644
16.9.21 *gets* . . . 644
16.9.22 *assert-result-and-state* . . . 645
16.9.23 *assuming* . . . 646

16.9.24 *guard* . . . 647
16.9.25 *assert-opt* . . . 648
16.9.26 *gets-the* . . . 649
16.9.27 *modify* . . . 649
16.9.28 condition . . . 650
16.9.29 *when* . . . 654
16.9.30 While . . . 654
16.9.31 *map-value* . . . 664
16.9.32 *liftE* . . . 666
16.9.33 *try* . . . 669
16.9.34 *finally* . . . 670
16.9.35 (<*catch*>) . . . 671
16.9.36 *check* . . . 673
16.9.37 *ignoreE* . . . 674
16.9.38 *on-exit′* . . . 676
16.9.39 *run-bind* . . . 678
16.9.40 Iteration of monadic actions . . . 679
16.9.41 *forLoop* . . . 679
16.9.42 *whileLoop-unroll-reachable* . . . 680
16.9.43 *on-exit* . . . 680
16.10 Setup for Tagging . . . 682
16.11 *succeeds* and *reaches* . . . 703
16.11.1 Relational rewriting for Monads . . . 704
16.11.2 ⊤ . . . 707
16.11.3 ⊥ . . . 707
16.11.4 *fail* . . . 707
16.11.5 *yield* . . . 707
16.11.6 *return* . . . 708
16.11.7 *skip* . . . 708
16.11.8 *throw-exception-or-result* . . . 708
16.11.9 *throw* . . . 709
16.11.10 *get-state* . . . 709
16.11.11 *set-state* . . . 709
16.11.12 *select* . . . 709
16.11.13 *unknown* . . . 710
16.11.14 *lift-state* . . . 710
16.11.15 const *exec-concrete* . . . 710
16.11.16 const *exec-abstract* . . . 710
16.11.17 *bind-handle* . . . 710
16.11.18 (≫=) . . . 711
16.11.19 *assert* . . . 713
16.11.20 *assume* . . . 713
16.11.21 *assume-outcome* . . . 713
16.11.22 *assume-result-and-state* . . . 714

16.11.23 *gets* . . . 714
16.11.24 *assert-result-and-state* . . . 714
16.11.25 *assuming* . . . 715
16.11.26 *guard* . . . 715
16.11.27 *assert-opt* . . . 715
16.11.28 *gets-the* . . . 715
16.11.29 *modify* . . . 716
16.11.30 condition . . . 716
16.11.31 *when* . . . 717
16.11.32 While . . . 717
16.11.33 *map-value* . . . 717
16.11.34 *liftE* . . . 717
16.11.35 *try* . . . 718
16.11.36 *finally* . . . 718
16.11.37 (<*catch*>) . . . 718
16.11.38 *check* . . . 719
16.11.39 *ignoreE* . . . 719
16.11.40 *bind-exception-or-result* . . . 720
16.11.41 *bind-finally* . . . 720
16.11.42 *run-bind* . . . 720

17 Basic Stuff **724**

18 L1 phase **734**
18.1 Peep-hole L1 optimisations . . . 749
18.2 Hoare-Triples for L1 (internal use) . . . 760

19 L2 phase: local variable abstraction with lambdas **766**
19.1 Some Relators . . . 779
19.2 Nested Exceptions . . . 808
19.3 Transformations from single level exceptions to nested exceptions . . . 808
19.4 Transformations for procedure local exceptions . . . 815
19.4.1 Transformations for *try* and *finally* . . . 815
19.5 Transformations on global exceptions . . . 815
19.5.1 Removing unused tuple components . . . 815
19.5.2 Setup basic rules . . . 815
19.6 Peep-hole optimisations applied to L2 . . . 817

20 IO phase: In/Out Parameters **829**
20.1 Heap Typing for Split Heap . . . 829
20.2 Valid root footprint . . . 829
20.3 More Stack Typing . . . 864
20.4 In Out Parameter Refinement . . . 876

21 HL phase: Heap Lifting / Split Heap **938**
21.1 Open Types . . . 956
21.1.1 Syntax *PTR-VALID*('a) . . . 972
21.1.2 Syntax *IS-VALID*('a) . . . 973
21.2 Refinement Lemmas . . . 974

22 WA phase: Word Abstraction **1031**
22.1 Basic Definitions . . . 1031
22.2 Abstracting values and expressions . . . 1032
22.3 Refinement Lemmas . . . 1041

23 TS phase: Type Strengthening (find suitable target monad) **1059**
23.1 Synthesize Rules Setup . . . 1061
23.2 Pure Monad . . . 1061
23.3 Reader Monad (Gets) . . . 1063
23.4 Option (Reader) Monad . . . 1065
23.5 Nondet Monad . . . 1068
23.5.1 Elimination of *L2-try* in the Error Monad . . . 1076
23.6 Error Monad (exit) . . . 1081

24 Polishing the Final Outcome **1086**
24.1 Support to normalise guards and array index expressions . . . 1100
24.2 Monad simplification with custom congruence rules . . . 1103

IV Documentation **1117**

25 Quickstart **1118**
25.1 Introduction . . . 1118
25.2 A First Proof with AutoCorres . . . 1118
25.2.1 Two simple functions: `min` and `max` . . . 1119
25.2.2 Invoking the C-parser . . . 1119
25.2.3 Invoking AutoCorres . . . 1121
25.2.4 Verifying `min` . . . 1121
25.2.5 Verifying `max` . . . 1122
25.3 More Complex Functions with AutoCorres . . . 1122
25.3.1 A simple loop: `mult_by_add` . . . 1123
25.3.2 `swap` . . . 1126
25.4 Command Options and Invocation . . . 1128
25.4.1 Session Structure . . . 1128
25.4.2 C-Parser . . . 1128
25.4.3 AutoCorres . . . 1129

26 Overview of AutoCorres **1132**
26.1 Building Blocks . . . 1132

26.2 C Parser 1133
26.3 AutoCorres 1134
26.4 AutoCorres Flow 1137
26.4.1 General Remarks 1137
26.4.2 Links to more documentation / examples 1138
26.5 Overview on the Locales 1138
26.6 Example Program 1138
26.6.1 Incremental Build 1139
26.6.2 All the rest 1140
26.7 Simplification strategy and dealing with tuples in L2-optimization phases 1143
26.8 Simplification of conditions (guards, loops, conditionals) . . .1147
26.9 Tricks to enforce first-order unification 1149
26.10Exception Rewriting 1150
26.10.1 Preliminary examples illustrating the usage of *rel-spec-monad*1151
26.10.2 From *L2-catch* and flat exceptions to *L2-try* and nested exceptions 1152
26.10.3 Flatten the error type of calls, aka get rid of constructor *Nonlocal* 1155
26.11Tuple optimization by analysing variable use and removing unused variables. 1156
26.12Locales, Local-Theories, Named-Targets, Morphisms and Declarations... 1159
26.13Morphisms and Declarations 1162
26.13.1 Excurse on Proof Context vs. Local Theory. 1164
26.13.2 Attributes vs. Local Theory Declarations 1165
26.14ML Antiquotations 1166
26.15Markup and Reports 1168
26.16Term Synthesize via Intro Rules 1169
26.17Pointers to Local Variables 1172
26.17.1 Design choices 1176
26.17.2 Open Ends / TODOs 1182
26.18In-Out Parameters, Abstracting Pointers to Values 1183
26.18.1 Overview 1183
26.18.2 Building Blocks 1183
26.18.3 Options 1188
26.18.4 Examples 1188
26.18.5 Handling `exit` in function calls 1196
26.18.6 Global Heap Pointers 1197
26.18.7 Disclaimer / Caution 1198
26.18.8 Implementation Aspects 1199
26.18.9 Pointer-parameters as data 1199
26.18.10 Function pointers 1200
26.18.11 Future work / Open Ends 1201

26.19 Function Pointers 1202
26.19.1 Global locales 1205
26.19.2 Function / Recursive-clique specific locales 1208
26.20 Unions 1222
26.20.1 Union support in C-Parser and Autocorres 1222
26.21 Pointers into Structures in Split Heap 1225
26.21.1 Overview 1226
26.21.2 Example Program and some Intuition 1227
26.21.3 Background 1229
26.21.4 User Level 1234
26.21.5 Simulation Proof 1239
26.21.6 Examples for normalisation of array indexes 1288
26.21.7 Essence of Heap Lifting 1289

27 C-Translation Infrastructure **1293**
27.1 Local Variables 1293
27.1.1 Basic ML primitives 1294
27.1.2 Syntax 1294
27.1.3 Simplifier setup 1295
27.2 Infrastructure for states 1297
27.3 Cached simproc examples 1298

28 Translation of the StrictC Dialect (Outdated) **1302**
28.1 Introduction 1302
28.1.1 StrictC Subset Summary 1303
28.2 Abstract Syntax 1303
28.2.1 Regions 1305
28.3 The Symbol Table 1307
28.3.1 Functional Record Updates in SML 1308
28.4 Creation of the Hoare Environment State 1309
28.4.1 Representing Values in Memory 1310
28.4.2 Pointers 1311
28.4.3 Arrays 1311
28.4.4 C `struct` Types 1312
28.5 Translating Expressions 1313
28.5.1 Undefined Behaviour 1315
28.6 Concrete Syntax: Parsing and Lexing 1316
28.6.1 Lexing and `typedef` Names 1317
28.6.2 GCC `__attribute__` Declarations 1318

Chapter 1

Introduction

AutoCorres2 is a tool to facilitate the verification of C programs within Isabelle [5]. It is a fork of AutoCorres: https://trustworthy.systems/projects/OLD/autocorres/. Here some quick links into the document:

- Quickstart guide for users (chapter 25): `doc/quickstart/Chapter1_MinMax.thy`
- Background information, internals and some history of AutoCorres (chapter 26): `doc/AutoCorresInfrastructure.thy`
- C-Parser
  - Some internals (chapter 27): `c-parser/CTranslationInfrastructure.thy`
  - Original documentation (outdated) (chapter 28): The supported subset of C is extended. Moreover, the C-parser is integrated into Isabelle/ML and no standalone C-parser is supplied. The description of the design principles is still valid: `c-parser/doc/ctranslation_body.tex`

Part I

Library

Chapter 2

Misc. Definitions and Lemmas

theory *More-Lib*
imports
Introduction-AutoCorres2
HOL−Library.Prefix-Order
Word-Lib.Word-Lib-Sumo
HOL−Eisbach.Eisbach-Tools
begin

abbreviation (*input*)
split :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c$
where
split == *case-prod*

lemma *hd-map-simp*:
$b \neq [] \Longrightarrow hd\ (map\ a\ b) = a\ (hd\ b)$
⟨*proof*⟩

lemma *tl-map-simp*:
$tl\ (map\ a\ b) = map\ a\ (tl\ b)$
⟨*proof*⟩

lemma *Collect-eq*:
$\{x.\ P\ x\} = \{x.\ Q\ x\} \longleftrightarrow (\forall x.\ P\ x = Q\ x)$
⟨*proof*⟩

lemma *iff-impI*: $[\![P \Longrightarrow Q = R]\!] \Longrightarrow (P \longrightarrow Q) = (P \longrightarrow R)$ ⟨*proof*⟩

definition
fun-app :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (**infixr** ‹$› *10*) **where**
$f\ \$\ x \equiv f\ x$

declare *fun-app-def* [*iff*]

lemma *fun-app-cong*[*fundef-cong*]:
$[\![\ f\ x = f'\ x'\ ]\!] \Longrightarrow (f\ \$\ x) = (f'\ \$\ x')$
⟨*proof*⟩

lemma *fun-app-apply-cong*[*fundef-cong*]:
$f\ x\ y = f'\ x'\ y' \Longrightarrow (f\ \$\ x)\ y = (f'\ \$\ x')\ y'$
⟨*proof*⟩

lemma *if-apply-cong*[*fundef-cong*]:
$[\![\ P = P';\ x = x';\ P' \Longrightarrow f\ x' = f'\ x';\ \neg\ P' \Longrightarrow g\ x' = g'\ x'\ ]\!]$
$\Longrightarrow (\mathit{if}\ P\ \mathit{then}\ f\ \mathit{else}\ g)\ x = (\mathit{if}\ P'\ \mathit{then}\ f'\ \mathit{else}\ g')\ x'$
⟨*proof*⟩

lemma *case-prod-apply-cong*[*fundef-cong*]:
$[\![\ f\ (\mathit{fst}\ p)\ (\mathit{snd}\ p)\ s = f'\ (\mathit{fst}\ p')\ (\mathit{snd}\ p')\ s'\ ]\!] \Longrightarrow \mathit{case\text{-}prod}\ f\ p\ s = \mathit{case\text{-}prod}\ f'\ p'\ s'$
⟨*proof*⟩

lemma *prod-injects*:
$(x,y) = p \Longrightarrow x = \mathit{fst}\ p \wedge y = \mathit{snd}\ p$
$p = (x,y) \Longrightarrow x = \mathit{fst}\ p \wedge y = \mathit{snd}\ p$
⟨*proof*⟩

definition
pred-imp :: $('a \Rightarrow \mathit{bool}) \Rightarrow ('a \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$
where
$\mathit{pred\text{-}imp}\ P\ Q \equiv \forall x.\ P\ x \longrightarrow Q\ x$

lemma *pred-impI*: $(\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \mathit{pred\text{-}imp}\ P\ Q$
⟨*proof*⟩

definition
pred-conj :: $('a \Rightarrow \mathit{bool}) \Rightarrow ('a \Rightarrow \mathit{bool}) \Rightarrow ('a \Rightarrow \mathit{bool})$ (**infixl** ‹*and*› *35*)
where
$\mathit{pred\text{-}conj}\ P\ Q \equiv \lambda x.\ P\ x \wedge Q\ x$

definition
pred-disj :: $('a \Rightarrow \mathit{bool}) \Rightarrow ('a \Rightarrow \mathit{bool}) \Rightarrow ('a \Rightarrow \mathit{bool})$ (**infixl** ‹*or*› *30*)
where
$\mathit{pred\text{-}disj}\ P\ Q \equiv \lambda x.\ P\ x \vee Q\ x$

definition
pred-neg :: (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *bool*) (‹(‹*open-block notation*=‹*prefix pred-neg*››*not* -)› [*40*] *40*)
where
pred-neg P ≡ λ*x*. ¬ *P x*

lemma *pred-neg-simp*[*simp*]:
(*not P*) *s* ⟷ ¬ (*P s*)
⟨*proof*⟩

definition *K* ≡ λ*x y*. *x*

definition
zipWith :: (*'a* ⇒ *'b* ⇒ *'c*) ⇒ *'a list* ⇒ *'b list* ⇒ *'c list* **where**
zipWith f xs ys ≡ *map* (*case-prod f*) (*zip xs ys*)

primrec
delete :: *'a* ⇒ *'a list* ⇒ *'a list*
where
delete y [] = []
| *delete y* (*x*#*xs*) = (*if y*=*x then xs else x* # *delete y xs*)

definition
swp f ≡ λ*x y*. *f y x*

lemma *swp-apply*[*simp*]: *swp f y x* = *f x y*
⟨*proof*⟩

primrec (*nonexhaustive*)
theRight :: *'a* + *'b* ⇒ *'b* **where**
theRight (*Inr x*) = *x*

primrec (*nonexhaustive*)
theLeft :: *'a* + *'b* ⇒ *'a* **where**
theLeft (*Inl x*) = *x*

definition
isLeft x ≡ (∃ *y*. *x* = *Inl y*)

definition
isRight x ≡ (∃ *y*. *x* = *Inr y*)

definition
const x ≡ λ*y*. *x*

primrec
opt-rel :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a option* ⇒ *'b option* ⇒ *bool*
where
opt-rel f None y = (*y* = *None*)

| *opt-rel f* (*Some x*) *y* = (∃ *y'*. *y* = *Some y'* ∧ *f x y'*)

lemma *opt-rel-None-rhs*[*simp*]:
opt-rel f x None = (*x* = *None*)
⟨*proof*⟩

lemma *opt-rel-Some-rhs*[*simp*]:
opt-rel f x (*Some y*) = (∃ *x'*. *x* = *Some x'* ∧ *f x' y*)
⟨*proof*⟩

lemma *tranclD2*:
(*x*, *y*) ∈ R^+ ⟹ ∃ *z*. (*x*, *z*) ∈ R^* ∧ (*z*, *y*) ∈ *R*
⟨*proof*⟩

lemma *linorder-min-same1* [*simp*]:
(*min y x* = *y*) = (*y* ≤ (*x*::'*a*::*linorder*))
⟨*proof*⟩

lemma *linorder-min-same2* [*simp*]:
(*min x y* = *y*) = (*y* ≤ (*x*::'*a*::*linorder*))
⟨*proof*⟩

A combinator for pairing up well-formed relations. The divisor function splits the population in halves, with the True half greater than the False half, and the supplied relations control the order within the halves.

definition
wf-sum :: ('*a* ⇒ *bool*) ⇒ ('*a* × '*a*) *set* ⇒ ('*a* × '*a*) *set* ⇒ ('*a* × '*a*) *set*
where
wf-sum divisor r r' ≡
(
{(*x*, *y*). ¬ *divisor x* ∧ ¬ *divisor y*} ∩ *r'*)
∪ {(*x*, *y*). ¬ *divisor x* ∧ *divisor y*}
∪ ({(*x*, *y*). *divisor x* ∧ *divisor y*} ∩ *r*)

lemma *wf-sum-wf*:
⟦ *wf r*; *wf r'* ⟧ ⟹ *wf* (*wf-sum divisor r r'*)
⟨*proof*⟩

abbreviation(*input*)
option-map == *map-option*

lemmas *option-map-def* = *map-option-case*

lemma *False-implies-equals* [*simp*]:
((*False* ⟹ *P*) ⟹ *PROP Q*) ≡ *PROP Q*
⟨*proof*⟩

lemma *split-paired-Ball*:
(∀ *x* ∈ *A*. *P x*) = (∀ *x y*. (*x*,*y*) ∈ *A* ⟶ *P* (*x*,*y*))

⟨*proof*⟩

lemma *split-paired-Bex*:
(∃ *x* ∈ *A*. *P x*) = (∃ *x y*. (*x*,*y*) ∈ *A* ∧ *P* (*x*,*y*))
⟨*proof*⟩

lemma *bexI-minus*:
⟦ *P x*; *x* ∈ *A*; *x* ∉ *B* ⟧ ⟹ ∃ *x* ∈ *A* − *B*. *P x*
⟨*proof*⟩

lemma *delete-remove1*:
delete x xs = *remove1 x xs*
⟨*proof*⟩

lemma *ignore-if*:
(*y and z*) *s* ⟹ (*if x then y else z*) *s*
⟨*proof*⟩

lemma *zipWith-Nil2* :
zipWith f xs [] = []
⟨*proof*⟩

lemma *isRight-right-map*:
isRight (*case-sum Inl* (*Inr o f*) *v*) = *isRight v*
⟨*proof*⟩

lemma *zipWith-nth*:
⟦ *n* < *min* (*length xs*) (*length ys*) ⟧ ⟹ *zipWith f xs ys* ! *n* = *f* (*xs* ! *n*) (*ys* ! *n*)
⟨*proof*⟩

lemma *length-zipWith* [*simp*]:
length (*zipWith f xs ys*) = *min* (*length xs*) (*length ys*)
⟨*proof*⟩

lemma *first-in-uptoD*:
a ≤ *b* ⟹ (*a*::'*a*::*order*) ∈ {*a*..*b*}
⟨*proof*⟩

lemma *construct-singleton*:
⟦ *S* ≠ {}; ∀ *s*∈*S*. ∀ *s'*. *s* ≠ *s'* ⟶ *s'* ∉ *S* ⟧ ⟹ ∃ *x*. *S* = {*x*}
⟨*proof*⟩

lemmas *insort-com* = *insort-left-comm*

lemma *bleeding-obvious*:
(*P* ⟹ *True*) ≡ (*Trueprop True*)
⟨*proof*⟩

lemma *Some-helper*:
$x = Some\ y \Longrightarrow x \neq None$
⟨*proof*⟩

lemma *in-empty-interE*:
$[\![\ A \cap B = \{\};\ x \in A;\ x \in B\ ]\!] \Longrightarrow False$
⟨*proof*⟩

lemma *None-upd-eq*:
$g\ x = None \Longrightarrow g(x := None) = g$
⟨*proof*⟩

lemma *exx* [*iff*]: $\exists\, x.\ x$ ⟨*proof*⟩
lemma *ExNot* [*iff*]: *Ex Not* ⟨*proof*⟩

lemma *cases-simp2* [*simp*]:
$((\neg\ P \longrightarrow Q) \wedge (P \longrightarrow Q)) = Q$
⟨*proof*⟩

lemma *a-imp-b-imp-b*:
$((a \longrightarrow b) \longrightarrow b) = (a \vee b)$
⟨*proof*⟩

lemma *length-neq*:
$length\ as \neq length\ bs \Longrightarrow as \neq bs$ ⟨*proof*⟩

lemma *take-neq-length*:
$[\![\ x \neq y;\ x \leq length\ as;\ y \leq length\ bs]\!] \Longrightarrow take\ x\ as \neq take\ y\ bs$
⟨*proof*⟩

lemma *eq-concat-lenD*:
$xs = ys\ @\ zs \Longrightarrow length\ xs = length\ ys + length\ zs$
⟨*proof*⟩

lemma *map-upt-reindex'*: $map\ f\ [a\ ..<\ b] = map\ (\lambda n.\ f\ (n + a - x))\ [x\ ..<\ x + b - a]$
⟨*proof*⟩

lemma *map-upt-reindex*: $map\ f\ [a\ ..<\ b] = map\ (\lambda n.\ f\ (n + a))\ [0\ ..<\ b - a]$
⟨*proof*⟩

lemma *notemptyI*:
$x \in S \Longrightarrow S \neq \{\}$
⟨*proof*⟩

lemma *setcomp-Max-has-prop*:
assumes *a*: $P\ x$
shows $P\ (Max\ \{(x::'a::\{finite,linorder\}).\ P\ x\})$
⟨*proof*⟩

lemma *cons-set-intro*:
lst = *x* # *xs* ⟹ *x* ∈ *set lst*
⟨*proof*⟩

lemma *list-all2-conj-nth*:
assumes *lall*: *list-all2 P as cs*
and *rl*: ⋀*n*. ⟦*P* (*as* ! *n*) (*cs* ! *n*); *n* < *length as*⟧ ⟹ *Q* (*as* ! *n*) (*cs* ! *n*)
shows *list-all2* (λ*a b*. *P a b* ∧ *Q a b*) *as cs*
⟨*proof*⟩

lemma *list-all2-conj*:
assumes *lall1*: *list-all2 P as cs*
and *lall2*: *list-all2 Q as cs*
shows *list-all2* (λ*a b*. *P a b* ∧ *Q a b*) *as cs*
⟨*proof*⟩

lemma *all-set-into-list-all2*:
assumes *lall*: ∀ *x* ∈ *set ls*. *P x*
and *length ls* = *length ls*′
shows *list-all2* (λ*a b*. *P a*) *ls ls*′
⟨*proof*⟩

lemma *GREATEST-lessE*:
fixes *x* :: ′*a* :: *order*
assumes *gts*: (*GREATEST x*. *P x*) < *X*
and *px*: *P x*
and *gtst*: ∃ *max*. *P max* ∧ (∀ *z*. *P z* ⟶ (*z* ≤ *max*))
shows *x* < *X*
⟨*proof*⟩

lemma *set-has-max*:
fixes *ls* :: (′*a* :: *linorder*) *list*
assumes *ls*: *ls* ≠ []
shows ∃ *max* ∈ *set ls*. ∀ *z* ∈ *set ls*. *z* ≤ *max*
⟨*proof*⟩

lemma *True-notin-set-replicate-conv*:
True ∉ *set ls* = (*ls* = *replicate* (*length ls*) *False*)
⟨*proof*⟩

lemma *Collect-singleton-eqI*:
(⋀*x*. *P x* = (*x* = *v*)) ⟹ {*x*. *P x*} = {*v*}
⟨*proof*⟩

lemma *exEI*:
⟦ ∃ *y*. *P y*; ⋀*x*. *P x* ⟹ *Q x* ⟧ ⟹ ∃ *z*. *Q z*
⟨*proof*⟩

lemma *allEI*:
assumes $\forall x.\ P\ x$
assumes $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows $\forall x.\ Q\ x$
⟨*proof*⟩

General lemmas that should be in the library

lemma *dom-ran*:
$x \in dom\ f \Longrightarrow the\ (f\ x) \in ran\ f$
⟨*proof*⟩

lemma *orthD1*:
$[\![\ S \cap S' = \{\};\ x \in S]\!] \Longrightarrow x \notin S'$ ⟨*proof*⟩

lemma *orthD2*:
$[\![\ S \cap S' = \{\};\ x \in S']\!] \Longrightarrow x \notin S$ ⟨*proof*⟩

lemma *distinct-element*:
$[\![\ b \cap d = \{\};\ a \in b;\ c \in d]\!] \Longrightarrow a \neq c$
⟨*proof*⟩

lemma *ball-reorder*:
$(\forall x \in A.\ \forall y \in B.\ P\ x\ y) = (\forall y \in B.\ \forall x \in A.\ \ P\ x\ y)$
⟨*proof*⟩

lemma *hd-map*: $ls \neq [] \Longrightarrow hd\ (map\ f\ ls) = f\ (hd\ ls)$
⟨*proof*⟩

lemma *tl-map*: $tl\ (map\ f\ ls) = map\ f\ (tl\ ls)$
⟨*proof*⟩

lemma *not-NilE*:
$[\![\ xs \neq [];\ \bigwedge x\ xs'.\ xs = x\ \#\ xs' \Longrightarrow R\ ]\!] \Longrightarrow R$
⟨*proof*⟩

lemma *length-SucE*:
$[\![\ length\ xs = Suc\ n;\ \bigwedge x\ xs'.\ xs = x\ \#\ xs' \Longrightarrow R\ ]\!] \Longrightarrow R$
⟨*proof*⟩

lemma *map-upt-unfold*:
assumes *ab*: $a < b$
shows $map\ f\ [a\ ..<\ b] = f\ a\ \#\ map\ f\ [Suc\ a\ ..<\ b]$
⟨*proof*⟩

lemma *tl-nat-list-simp*:
$tl\ [a..<b] = [a\ +\ 1\ ..<b]$
⟨*proof*⟩

lemma *image-Collect2*:

case-prod f ‘ {x. P (fst x) (snd x)} = {f x y |x y. P x y}
⟨*proof*⟩

lemma *image-id′*:
id ‘ Y = Y
⟨*proof*⟩

lemma *image-invert*:
assumes *r*: *f ∘ g = id*
and *g*: *B = g ‘ A*
shows *A = f ‘ B*
⟨*proof*⟩

lemma *Collect-image-fun-cong*:
assumes *rl*: *⋀a. P a ⟹ f a = g a*
shows *{f x | x. P x} = {g x | x. P x}*
⟨*proof*⟩

lemma *inj-on-take*:
shows *inj-on (take n) {x. drop n x = k}*
⟨*proof*⟩

lemma *foldr-upd-dom*:
dom (foldr (λp ps. ps (p ↦ f p)) as g) = dom g ∪ set as
⟨*proof*⟩

lemma *foldr-upd-app*:
assumes *xin*: *x ∈ set as*
shows *(foldr (λp ps. ps (p ↦ f p)) as g) x = Some (f x)*
(**is** *(?f as g) x = Some (f x)*)
⟨*proof*⟩

lemma *foldr-upd-app-other*:
assumes *xin*: *x ∉ set as*
shows *(foldr (λp ps. ps (p ↦ f p)) as g) x = g x*
(**is** *(?f as g) x = g x*)
⟨*proof*⟩

lemma *foldr-upd-app-if*:
foldr (λp ps. ps(p ↦ f p)) as g = (λx. if x ∈ set as then Some (f x) else g x)
⟨*proof*⟩

lemma *foldl-fun-upd-value*:
⋀Y. foldl (λf p. f(p := X p)) Y e p = (if p∈set e then X p else Y p)
⟨*proof*⟩

lemma *foldr-fun-upd-value*:
⋀Y. foldr (λp f. f(p := X p)) e Y p = (if p∈set e then X p else Y p)
⟨*proof*⟩

lemma *foldl-fun-upd-eq-foldr*:
!!m. $foldl\ (\lambda f\ p.\ f(p := g\ p))\ m\ xs = foldr\ (\lambda p\ f.\ f(p := g\ p))\ xs\ m$
⟨*proof*⟩

lemma *Cons-eq-neq*:
$[\![\ y = x;\ x\ \#\ xs \neq y\ \#\ ys\ ]\!] \Longrightarrow xs \neq ys$
⟨*proof*⟩

lemma *map-upt-append*:
assumes *lt*: $x \leq y$
and *lt2*: $a \leq x$
shows $map\ f\ [a\ ..<\ y] = map\ f\ [a\ ..<\ x]\ @\ map\ f\ [x\ ..<\ y]$
⟨*proof*⟩

lemma *Min-image-distrib*:
assumes *minf*: $\bigwedge x\ y.\ [\![\ x \in A;\ y \in A\ ]\!] \Longrightarrow min\ (f\ x)\ (f\ y) = f\ (min\ x\ y)$
and *fa*: *finite A*
and *ane*: $A \neq \{\}$
shows $Min\ (f\ `\ A) = f\ (Min\ A)$
⟨*proof*⟩

lemma *min-of-mono'*:
assumes $(f\ a \leq f\ c) = (a \leq c)$
shows $min\ (f\ a)\ (f\ c) = f\ (min\ a\ c)$
⟨*proof*⟩

lemma *nat-diff-less*:
fixes $x :: nat$
shows $[\![\ x < y + z;\ z \leq x]\!] \Longrightarrow x - z < y$
⟨*proof*⟩

lemma *take-map-Not*:
$(take\ n\ (map\ Not\ xs) = take\ n\ xs) = (n = 0 \vee xs = [])$
⟨*proof*⟩

lemma *union-trans*:
assumes *SR*: $\bigwedge x\ y\ z.\ [\![\ (x,y) \in S;\ (y,z) \in R\ ]\!] \Longrightarrow (x,z) \in S\widehat{}*$
shows $(R \cup S)\widehat{}* = R\widehat{}* \cup R\widehat{}*\ O\ S\widehat{}*$
⟨*proof*⟩

lemma *trancl-trancl*:
$(R^+)^+ = R^+$
⟨*proof*⟩

Some rules for showing that the reflexive transitive closure of a relation/predicate doesn't add much if it was already transitively closed.

lemma *rtrancl-eq-reflc-trans*:

assumes *trans*: *trans X*
shows *rtrancl X* = *X* ∪ *Id*
⟨*proof*⟩

lemma *rtrancl-id*:
assumes *refl*: *Id* ⊆ *X*
assumes *trans*: *trans X*
shows *rtrancl X* = *X*
⟨*proof*⟩

lemma *rtranclp-eq-reflcp-transp*:
assumes *trans*: *transp X*
shows *rtranclp X* = (λ*x y*. *X x y* ∨ *x* = *y*)
⟨*proof*⟩

lemma *rtranclp-id*:
shows *reflp X* ⟹ *transp X* ⟹ *rtranclp X* = *X*
⟨*proof*⟩

lemmas *rtranclp-id2* = *rtranclp-id*[*unfolded reflp-def transp-relcompp le-fun-def*]

lemma *if-1-0-0*:
((*if P then 1 else 0*) = (*0* :: (*'a* :: *zero-neq-one*))) = (¬ *P*)
⟨*proof*⟩

lemma *neq-Nil-lengthI*:
Suc 0 ≤ *length xs* ⟹ *xs* ≠ []
⟨*proof*⟩

lemmas *ex-with-length* = *Ex-list-of-length*

lemma *in-singleton*:
S = {*x*} ⟹ *x* ∈ *S*
⟨*proof*⟩

lemma *singleton-set*:
x ∈ *set* [*a*] ⟹ *x* = *a*
⟨*proof*⟩

lemma *take-drop-eqI*:
assumes *t*: *take n xs* = *take n ys*
assumes *d*: *drop n xs* = *drop n ys*
shows *xs* = *ys*
⟨*proof*⟩

lemma *append-len2*:
zs = *xs* @ *ys* ⟹ *length xs* = *length zs* − *length ys*
⟨*proof*⟩

lemma *if-flip*:
$(if\ \neg P\ then\ T\ else\ F) = (if\ P\ then\ F\ else\ T)$
⟨*proof*⟩

lemma *not-in-domIff*:$f\ x = None = (x \notin dom\ f)$
⟨*proof*⟩

lemma *not-in-domD*:
$x \notin dom\ f \Longrightarrow f\ x = None$
⟨*proof*⟩

definition
$graph\text{-}of\ f \equiv \{(x,y).\ f\ x = Some\ y\}$

lemma *graph-of-None-update*:
$graph\text{-}of\ (f\ (p := None)) = graph\text{-}of\ f - \{p\} \times UNIV$
⟨*proof*⟩

lemma *graph-of-Some-update*:
$graph\text{-}of\ (f\ (p \mapsto v)) = (graph\text{-}of\ f - \{p\} \times UNIV) \cup \{(p,v)\}$
⟨*proof*⟩

lemma *graph-of-restrict-map*:
$graph\text{-}of\ (m\ |`\ S) \subseteq graph\text{-}of\ m$
⟨*proof*⟩

lemma *graph-ofD*:
$(x,y) \in graph\text{-}of\ f \Longrightarrow f\ x = Some\ y$
⟨*proof*⟩

lemma *graph-ofI*:
$m\ x = Some\ y \Longrightarrow (x,\ y) \in graph\text{-}of\ m$
⟨*proof*⟩

lemma *graph-of-empty* :
$graph\text{-}of\ Map.empty = \{\}$
⟨*proof*⟩

lemma *graph-of-in-ranD*: $\forall\ y \in ran\ f.\ P\ y \Longrightarrow (x,y) \in graph\text{-}of\ f \Longrightarrow P\ y$
⟨*proof*⟩

lemma *graph-of-SomeD*:
$[\![\ graph\text{-}of\ f \subseteq graph\text{-}of\ g;\ f\ x = Some\ y\ ]\!] \Longrightarrow g\ x = Some\ y$
⟨*proof*⟩

lemma *graph-of-comp*:
$[\![\ g\ x = y;\ f\ y = Some\ z\ ]\!] \Longrightarrow (x,z) \in graph\text{-}of\ (f \circ g)$
⟨*proof*⟩

lemma *in-set-zip-refl* :
(*x*,*y*) ∈ *set* (*zip xs xs*) = (*y* = *x* ∧ *x* ∈ *set xs*)
⟨*proof*⟩

lemma *map-conv-upd*:
m v = *None* ⟹ *m o* (*f* (*x* := *v*)) = (*m o f*) (*x* := *None*)
⟨*proof*⟩

lemma *sum-all-ex* [*simp*]:
(∀ *a*. *x* ≠ *Inl a*) = (∃ *a*. *x* = *Inr a*)
(∀ *a*. *x* ≠ *Inr a*) = (∃ *a*. *x* = *Inl a*)
⟨*proof*⟩

lemma *split-distrib*: *case-prod* (λ*a b*. *T* (*f a b*)) = (λ*x*. *T* (*case-prod* (λ*a b*. *f a b*) *x*))
⟨*proof*⟩

lemma *case-sum-triv* [*simp*]:
(*case x of Inl x* ⇒ *Inl x* | *Inr x* ⇒ *Inr x*) = *x*
⟨*proof*⟩

lemma *set-eq-UNIV*: ({*a*. *P a*} = *UNIV*) = (∀ *a*. *P a*)
⟨*proof*⟩

lemma *allE2*:
⟦∀ *x y*. *P x y*; *P x y* ⟹ *R*⟧ ⟹ *R*
⟨*proof*⟩

lemma *allE3*: ⟦ ∀ *x y z*. *P x y z*; *P x y z* ⟹ *R* ⟧ ⟹ *R*
⟨*proof*⟩

lemma *my-BallE*: ⟦ ∀ *x* ∈ *A*. *P x*; *y* ∈ *A*; *P y* ⟹ *Q* ⟧ ⟹ *Q*
⟨*proof*⟩

lemma *unit-Inl-or-Inr* [*simp*]:
(*a* ≠ *Inl* ()) = (*a* = *Inr* ())
(*a* ≠ *Inr* ()) = (*a* = *Inl* ())
⟨*proof*⟩

lemma *disjE-L*: ⟦ *a* ∨ *b*; *a* ⟹ *R*; ⟦ ¬ *a*; *b* ⟧ ⟹ *R* ⟧ ⟹ *R*
⟨*proof*⟩

lemma *disjE-R*: ⟦ *a* ∨ *b*; ⟦ ¬ *b*; *a* ⟧ ⟹ *R*; ⟦ *b* ⟧ ⟹ *R* ⟧ ⟹ *R*
⟨*proof*⟩

lemma *int-max-thms*:
(*a* :: *int*) ≤ *max a b*
(*b* :: *int*) ≤ *max a b*
⟨*proof*⟩

lemma *sgn-negation* [*simp*]:
$sgn\ (-(x::int)) = -\ sgn\ x$
⟨*proof*⟩

lemma *sgn-sgn-nonneg* [*simp*]:
$sgn\ (a :: int) * sgn\ a \neq -1$
⟨*proof*⟩

lemma *inj-inj-on*:
$inj\ f \Longrightarrow inj\text{-}on\ f\ A$
⟨*proof*⟩

lemma *ex-eqI*:
$[\![\bigwedge x.\ f\ x = g\ x]\!] \Longrightarrow (\exists\, x.\ f\ x) = (\exists\, x.\ g\ x)$
⟨*proof*⟩

lemma *pre-post-ex*:
$[\![\exists\, x.\ P\ x;\ \bigwedge x.\ P\ x \Longrightarrow Q\ x]\!] \Longrightarrow \exists\, x.\ Q\ x$
⟨*proof*⟩

lemma *ex-conj-increase*:
$((\exists\, x.\ P\ x) \wedge Q) = (\exists\, x.\ P\ x \wedge Q)$
$(R \wedge (\exists\, x.\ S\ x)) = (\exists\, x.\ R \wedge S\ x)$
⟨*proof*⟩

lemma *all-conj-increase*:
$((\ \forall\, x.\ P\ x) \wedge Q) = (\forall\, x.\ P\ x \wedge Q)$
$(R \wedge (\forall\, x.\ S\ x)) = (\forall\, x.\ R \wedge S\ x)$
⟨*proof*⟩

lemma *Ball-conj-increase*:
$xs \neq \{\} \Longrightarrow ((\ \forall\, x \in xs.\ P\ x) \wedge Q) = (\forall\, x \in xs.\ P\ x \wedge Q)$
$xs \neq \{\} \Longrightarrow (R \wedge (\forall\, x \in xs.\ S\ x)) = (\forall\, x \in xs.\ R \wedge S\ x)$
⟨*proof*⟩

lemma *disjoint-subset*:
assumes $A' \subseteq A$ **and** $A \cap B = \{\}$
shows $A' \cap B = \{\}$
⟨*proof*⟩

lemma *disjoint-subset2*:
assumes $B' \subseteq B$ **and** $A \cap B = \{\}$
shows $A \cap B' = \{\}$
⟨*proof*⟩

lemma *UN-nth-mem*:
$i < length\ xs \Longrightarrow f\ (xs\ !\ i) \subseteq (\bigcup x{\in}set\ xs.\ f\ x)$
⟨*proof*⟩

lemma *Union-equal*:
$f\ `\ A = f\ `\ B \Longrightarrow (\bigcup x \in A.\ f\ x) = (\bigcup x \in B.\ f\ x)$
⟨*proof*⟩

lemma *UN-Diff-disjoint*:
$i < length\ xs \Longrightarrow (A - (\bigcup x{\in}set\ xs.\ f\ x)) \cap f\ (xs\ !\ i) = \{\}$
⟨*proof*⟩

lemma *image-list-update*:
$f\ a = f\ (xs\ !\ i)$
$\Longrightarrow f\ `\ set\ (xs\ [i := a]) = f\ `\ set\ xs$
⟨*proof*⟩

lemma *Union-list-update-id*:
$f\ a = f\ (xs\ !\ i) \Longrightarrow (\bigcup x{\in}set\ (xs\ [i := a]).\ f\ x) = (\bigcup x{\in}set\ xs.\ f\ x)$
⟨*proof*⟩

lemma *Union-list-update-id′*:
$[\![i < length\ xs; \bigwedge x.\ g\ (f\ x) = g\ x]\!]$
$\Longrightarrow (\bigcup x{\in}set\ (xs\ [i := f\ (xs\ !\ i)]).\ g\ x) = (\bigcup x{\in}set\ xs.\ g\ x)$
⟨*proof*⟩

lemma *Union-subset*:
$[\![\bigwedge x.\ x \in A \Longrightarrow (f\ x) \subseteq (g\ x)]\!] \Longrightarrow (\bigcup x \in A.\ f\ x) \subseteq (\bigcup x \in A.\ g\ x)$
⟨*proof*⟩

lemma *UN-sub-empty*:
$[\![list\text{-}all\ P\ xs; \bigwedge x.\ P\ x \Longrightarrow f\ x = g\ x]\!] \Longrightarrow (\bigcup x{\in}set\ xs.\ f\ x) - (\bigcup x{\in}set\ xs.\ g\ x)$
$= \{\}$
⟨*proof*⟩

lemma *bij-betw-fun-updI*:
$[\![x \notin A; y \notin B; bij\text{-}betw\ f\ A\ B]\!] \Longrightarrow bij\text{-}betw\ (f(x := y))\ (insert\ x\ A)\ (insert\ y\ B)$
⟨*proof*⟩

definition
$bij\text{-}betw\text{-}map\ f\ A\ B \equiv bij\text{-}betw\ f\ A\ (Some\ `\ B)$

lemma *bij-betw-map-fun-updI*:
$[\![x \notin A; y \notin B; bij\text{-}betw\text{-}map\ f\ A\ B]\!]$
$\Longrightarrow bij\text{-}betw\text{-}map\ (f(x \mapsto y))\ (insert\ x\ A)\ (insert\ y\ B)$
⟨*proof*⟩

lemma *bij-betw-map-imp-inj-on*:
bij-betw-map f A B $\Longrightarrow$ *inj-on f A*
⟨*proof*⟩

lemma *bij-betw-empty-dom-exists*:
$r = \{\} \Longrightarrow \exists\, t.$ *bij-betw t* $\{\}$ *r*
⟨*proof*⟩

lemma *bij-betw-map-empty-dom-exists*:
$r = \{\} \Longrightarrow \exists\, t.$ *bij-betw-map t* $\{\}$ *r*
⟨*proof*⟩

lemma *funpow-add* [*simp*]:
fixes $f :: 'a \Rightarrow 'a$
shows $(f \hat{\ }\hat{\ } a)\ ((f \hat{\ }\hat{\ } b)\ s) = (f \hat{\ }\hat{\ } (a + b))\ s$
⟨*proof*⟩

lemma *funpow-unfold*:
fixes $f :: 'a \Rightarrow 'a$
assumes $n > 0$
shows $f \hat{\ }\hat{\ } n = (f \hat{\ }\hat{\ } (n - 1)) \circ f$
⟨*proof*⟩

lemma *relpow-unfold*: $n > 0 \Longrightarrow S \hat{\ }\hat{\ } n = (S \hat{\ }\hat{\ } (n - 1))\ O\ S$
⟨*proof*⟩

definition
equiv-of :: $('s \Rightarrow 't) \Rightarrow ('s \times 's)$ *set*
where
equiv-of proj $\equiv \{(a, b).$ *proj a* $=$ *proj b*$\}$

lemma *equiv-of-is-equiv-relation* [*simp*]:
equiv UNIV (*equiv-of proj*)
⟨*proof*⟩

lemma *in-equiv-of* [*simp*]:
$((a, b) \in$ *equiv-of f*$) \longleftrightarrow (f\ a = f\ b)$
⟨*proof*⟩

lemma *equiv-relation-to-projection*:
fixes $R :: ('a \times 'a)$ *set*
assumes *equiv*: *equiv UNIV R*

shows $\exists f :: 'a \Rightarrow 'a\ set.\ \forall x\ y.\ f\ x = f\ y \longleftrightarrow (x,\ y) \in R$
⟨*proof*⟩

lemma *range-constant* [*simp*]:
$range\ (\lambda\text{-}.\ k) = \{k\}$
⟨*proof*⟩

lemma *dom-unpack*:
$dom\ (map\text{-}of\ (map\ (\lambda x.\ (f\ x,\ g\ x))\ xs)) = set\ (map\ (\lambda x.\ f\ x)\ xs)$
⟨*proof*⟩

lemma *fold-to-disj*:
$fold\ (++)\ ms\ a\ x = Some\ y \Longrightarrow (\exists\, b \in set\ ms.\ b\ x = Some\ y) \vee a\ x = Some\ y$
⟨*proof*⟩

lemma *fold-ignore1*:
$a\ x = Some\ y \Longrightarrow fold\ (++)\ ms\ a\ x = Some\ y$
⟨*proof*⟩

lemma *fold-ignore2*:
$fold\ (++)\ ms\ a\ x = None \Longrightarrow a\ x = None$
⟨*proof*⟩

lemma *fold-ignore3*:
$fold\ (++)\ ms\ a\ x = None \Longrightarrow (\forall\, b \in set\ ms.\ b\ x = None)$
⟨*proof*⟩

lemma *fold-ignore4*:
$b \in set\ ms \Longrightarrow b\ x = Some\ y \Longrightarrow \exists\, y.\ fold\ (++)\ ms\ a\ x = Some\ y$
⟨*proof*⟩

lemma *dom-unpack2*:
$dom\ (fold\ (++)\ ms\ Map.empty) = \bigcup (set\ (map\ dom\ ms))$
⟨*proof*⟩

lemma *fold-ignore5*:$fold\ (++)\ ms\ a\ x = Some\ y \Longrightarrow a\ x = Some\ y \vee (\exists\, b \in set\ ms.\ b\ x = Some\ y)$
⟨*proof*⟩

lemma *dom-inter-nothing*:$dom\ f \cap dom\ g = \{\} \Longrightarrow \forall x.\ f\ x = None \vee g\ x = None$
⟨*proof*⟩

lemma *fold-ignore6*:
$f\ x = None \Longrightarrow fold\ (++)\ ms\ f\ x = fold\ (++)\ ms\ Map.empty\ x$
⟨*proof*⟩

lemma *fold-ignore7*:
$m\ x = m'\ x \Longrightarrow fold\ (++)\ ms\ m\ x = fold\ (++)\ ms\ m'\ x$
⟨*proof*⟩

lemma *fold-ignore8*:
$fold\ (++)\ ms\ [x \mapsto y] = (fold\ (++)\ ms\ Map.empty)(x \mapsto y)$
⟨*proof*⟩

lemma *fold-ignore9*:
$[\![fold\ (++)\ ms\ [x \mapsto y]\ x' = Some\ z;\ x = x']\!] \Longrightarrow y = z$
⟨*proof*⟩

lemma *fold-to-map-of*:
$fold\ (++)\ (map\ (\lambda x.\ [f\ x \mapsto g\ x])\ xs)\ Map.empty = map\text{-}of\ (map\ (\lambda x.\ (f\ x,\ g\ x))\ xs)$
⟨*proof*⟩

lemma *if-n-0-0*:
$((if\ P\ then\ n\ else\ 0) \neq 0) = (P \wedge n \neq 0)$
⟨*proof*⟩

lemma *insert-dom*:
assumes *fx*: $f\ x = Some\ y$
shows $insert\ x\ (dom\ f) = dom\ f$
⟨*proof*⟩

lemma *map-comp-subset-dom*:
$dom\ (prj \circ_m f) \subseteq dom\ f$
⟨*proof*⟩

lemmas *map-comp-subset-domD* = *subsetD* [*OF map-comp-subset-dom*]

lemma *dom-map-comp*:
$x \in dom\ (prj \circ_m f) = (\exists\, y\ z.\ f\ x = Some\ y \wedge prj\ y = Some\ z)$
⟨*proof*⟩

lemma *map-option-Some-eq2*:
$(Some\ y = map\text{-}option\ f\ x) = (\exists\, z.\ x = Some\ z \wedge f\ z = y)$
⟨*proof*⟩

lemma *map-option-eq-dom-eq*:
assumes *ome*: $map\text{-}option\ f \circ g = map\text{-}option\ f \circ g'$
shows $dom\ g = dom\ g'$
⟨*proof*⟩

lemma *cart-singleton-image*:
$S \times \{s\} = (\lambda v.\ (v,\ s))\ `\ S$
⟨*proof*⟩

lemma *singleton-eq-o2s*:
$(\{x\} = set\text{-}option\ v) = (v = Some\ x)$

⟨*proof*⟩

lemma *option-set-singleton-eq*:
(*set-option opt* = {*v*}) = (*opt* = *Some v*)
⟨*proof*⟩

lemmas *option-set-singleton-eqs*
= *option-set-singleton-eq*
trans[*OF eq-commute option-set-singleton-eq*]

lemma *map-option-comp2*:
map-option (*f o g*) = *map-option f o map-option g*
⟨*proof*⟩

lemma *compD*:
⟦*f* ∘ *g* = *f* ∘ *g*′; *g x* = *v* ⟧ ⟹ *f* (*g*′ *x*) = *f v*
⟨*proof*⟩

lemma *map-option-comp-eqE*:
assumes *om*: *map-option f* ∘ *mp* = *map-option f* ∘ *mp*′
and *p1*: ⟦ *mp x* = *None*; *mp*′ *x* = *None* ⟧ ⟹ *P*
and *p2*: ⋀*v v*′. ⟦ *mp x* = *Some v*; *mp*′ *x* = *Some v*′; *f v* = *f v*′ ⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *Some-the*:
x ∈ *dom f* ⟹ *f x* = *Some* (*the* (*f x*))
⟨*proof*⟩

lemma *map-comp-update*:
f ∘$_m$ (*g*(*x* ↦ *v*)) = (*f* ∘$_m$ *g*)(*x* := *f v*)
⟨*proof*⟩

lemma *restrict-map-eqI*:
assumes *req*: *A* |‘ *S* = *B* |‘ *S*
and *mem*: *x* ∈ *S*
shows *A x* = *B x*
⟨*proof*⟩

lemma *map-comp-eqI*:
assumes *dm*: *dom g* = *dom g*′
and *fg*: ⋀*x*. *x* ∈ *dom g*′ ⟹ *f* (*the* (*g*′ *x*)) = *f* (*the* (*g x*))
shows *f* ∘$_m$ *g* = *f* ∘$_m$ *g*′
⟨*proof*⟩

definition
modify-map m p f ≡ *m* (*p* := *map-option f* (*m p*))

lemma *modify-map-id*:
modify-map m p id = m
⟨*proof*⟩

lemma *modify-map-addr-com*:
assumes *com*: *x ≠ y*
shows *modify-map (modify-map m x g) y f = modify-map (modify-map m y f) x g*
⟨*proof*⟩

lemma *modify-map-dom* :
dom (modify-map m p f) = dom m
⟨*proof*⟩

lemma *modify-map-None*:
m x = None ⟹ modify-map m x f = m
⟨*proof*⟩

lemma *modify-map-ndom* :
x ∉ dom m ⟹ modify-map m x f = m
⟨*proof*⟩

lemma *modify-map-app*:
(modify-map m p f) q = (if p = q then map-option f (m p) else m q)
⟨*proof*⟩

lemma *modify-map-apply*:
m p = Some x ⟹ modify-map m p f = m (p ↦ f x)
⟨*proof*⟩

lemma *modify-map-com*:
assumes *com*: *⋀x. f (g x) = g (f x)*
shows *modify-map (modify-map m x g) y f = modify-map (modify-map m y f) x g*
⟨*proof*⟩

lemma *modify-map-comp*:
modify-map m x (f o g) = modify-map (modify-map m x g) x f
⟨*proof*⟩

lemma *modify-map-exists-eq*:
(∃ cte. modify-map m p′ f p= Some cte) = (∃ cte. m p = Some cte)
⟨*proof*⟩

lemma *modify-map-other*:
p ≠ q ⟹ (modify-map m p f) q = (m q)
⟨*proof*⟩

lemma *modify-map-same*:

modify-map m p f p = map-option f (m p)
⟨*proof*⟩

lemma *next-update-is-modify*:
⟦ *m p = Some cte'*; *cte = f cte'* ⟧ ⟹ *(m(p ↦ cte)) = modify-map m p f*
⟨*proof*⟩

lemma *nat-power-minus-less*:
a < 2 ^ (x − n) ⟹ *(a :: nat) < 2 ^ x*
⟨*proof*⟩

lemma *neg-rtranclI*:
⟦ *x ≠ y*; *(x, y) ∉ R⁺* ⟧ ⟹ *(x, y) ∉ R**
⟨*proof*⟩

lemma *neg-rtrancl-into-trancl*:
¬ *(x, y) ∈ R** ⟹ ¬ *(x, y) ∈ R⁺*
⟨*proof*⟩

lemma *set-neqI*:
⟦ *x ∈ S*; *x ∉ S'* ⟧ ⟹ *S ≠ S'*
⟨*proof*⟩

lemma *set-pair-UN*:
{x. P x} = ⋃ ((λxa. {xa} × {xb. P (xa, xb)}) ‘ {xa. ∃xb. P (xa, xb)})
⟨*proof*⟩

lemma *singleton-elemD*: *S = {x}* ⟹ *x ∈ S*
⟨*proof*⟩

lemma *singleton-eqD*: *A = {x}* ⟹ *x ∈ A*
⟨*proof*⟩

lemma *ball-ran-fun-updI*:
⟦ *∀ v ∈ ran m. P v*; *∀ v. y = Some v ⟶ P v* ⟧ ⟹ *∀ v ∈ ran (m (x := y)). P v*
⟨*proof*⟩

lemma *ball-ran-eq*:
(∀ y ∈ ran m. P y) = (∀ x y. m x = Some y ⟶ P y)
⟨*proof*⟩

lemma *cart-helper*:
({} = {x} × S) = (S = {})
⟨*proof*⟩

lemmas *converse-trancl-induct'* = *converse-trancl-induct* [*consumes 1*, *case-names base step*]

lemma *disjCI2*: *(¬ P ⟹ Q) ⟹ P ∨ Q* ⟨*proof*⟩

lemma *insert-UNIV* :
$\mathit{insert}\ x\ \mathit{UNIV} = \mathit{UNIV}$
⟨*proof*⟩

lemma *not-singletonE*:
$[\![\forall p.\ S \neq \{p\};\ S \neq \{\};\ \bigwedge p\ p'.\ [\![p \neq p';\ p \in S;\ p' \in S]\!] \Longrightarrow R]\!] \Longrightarrow R$
⟨*proof*⟩

lemma *not-singleton-oneE*:
$[\![\forall p.\ S \neq \{p\};\ p \in S;\ \bigwedge p'.\ [\![p \neq p';\ p' \in S]\!] \Longrightarrow R]\!] \Longrightarrow R$
⟨*proof*⟩

lemma *ball-ran-modify-map-eq*:
$[\![\forall v.\ m\ x = \mathit{Some}\ v \longrightarrow P\ (f\ v) = P\ v]\!]$
$\Longrightarrow (\forall v \in \mathit{ran}\ (\mathit{modify\text{-}map}\ m\ x\ f).\ P\ v) = (\forall v \in \mathit{ran}\ m.\ P\ v)$
⟨*proof*⟩

lemma *eq-singleton-redux*:
$[\![S = \{x\}]\!] \Longrightarrow x \in S$
⟨*proof*⟩

lemma *if-eq-elem-helperE*:
$[\![x \in (\mathit{if}\ P\ \mathit{then}\ S\ \mathit{else}\ S');\ [\![P;\quad x \in S]\!] \Longrightarrow a = b;\ [\![\neg P; x \in S']\!] \Longrightarrow a = c]\!]$
$\Longrightarrow a = (\mathit{if}\ P\ \mathit{then}\ b\ \mathit{else}\ c)$
⟨*proof*⟩

lemma *if-option-Some*:
$((\mathit{if}\ P\ \mathit{then}\ \mathit{None}\ \mathit{else}\ \mathit{Some}\ x) = \mathit{Some}\ y) = (\neg P \wedge x = y)$
⟨*proof*⟩

lemma *insert-minus-eq*:
$x \notin A \Longrightarrow A - S = (A - (S - \{x\}))$
⟨*proof*⟩

lemma *modify-map-K-D*:
$\mathit{modify\text{-}map}\ m\ p\ (\lambda x.\ y)\ p' = \mathit{Some}\ v \Longrightarrow (m\ (p \mapsto y))\ p' = \mathit{Some}\ v$
⟨*proof*⟩

lemma *tranclE2*:
assumes *trancl*: $(a,\ b) \in r^+$
and *base*: $(a,\ b) \in r \Longrightarrow P$
and *step*: $\bigwedge c.\ [\![(a,\ c) \in r;\ (c,\ b) \in r^+]\!] \Longrightarrow P$
shows P
⟨*proof*⟩

lemmas $\mathit{tranclE2}' = \mathit{tranclE2}$ [*consumes 1*, *case-names base trancl*]

lemma *weak-imp-cong*:
$[\![P = R;\ Q = S]\!] \Longrightarrow (P \longrightarrow Q) = (R \longrightarrow S)$
⟨*proof*⟩

lemma *Collect-Diff-restrict-simp*:
$T - \{x \in T.\ Q\ x\} = T - \{x.\ Q\ x\}$
⟨*proof*⟩

lemma *Collect-Int-pred-eq*:
$\{x \in S.\ P\ x\} \cap \{x \in T.\ P\ x\} = \{x \in (S \cap T).\ P\ x\}$
⟨*proof*⟩

lemma *Collect-restrict-predR*:
$\{x.\ P\ x\} \cap T = \{\} \Longrightarrow \{x.\ P\ x\} \cap \{x \in T.\ Q\ x\} = \{\}$
⟨*proof*⟩

lemma *Diff-Un2*:
assumes *emptyad*: $A \cap D = \{\}$
and *emptybc*: $B \cap C = \{\}$
shows $(A \cup B) - (C \cup D) = (A - C) \cup (B - D)$
⟨*proof*⟩

lemma *ballEI*:
$[\![\forall x \in S.\ Q\ x;\ \bigwedge x.\ [\![x \in S;\ Q\ x]\!] \Longrightarrow P\ x]\!] \Longrightarrow \forall x \in S.\ P\ x$
⟨*proof*⟩

lemma *dom-if-None*:
$dom\ (\lambda x.\ if\ P\ x\ then\ None\ else\ f\ x) = dom\ f - \{x.\ P\ x\}$
⟨*proof*⟩

lemma *restrict-map-Some-iff*:
$((m\ |`\ S)\ x = Some\ y) = (m\ x = Some\ y \wedge x \in S)$
⟨*proof*⟩

lemma *context-case-bools*:
$[\![\bigwedge v.\ P\ v \Longrightarrow R\ v;\ [\![\neg\ P\ v;\ \bigwedge v.\ P\ v \Longrightarrow R\ v]\!] \Longrightarrow R\ v]\!] \Longrightarrow R\ v$
⟨*proof*⟩

lemma *inj-on-fun-upd-strongerI*:
$[\![inj\text{-}on\ f\ A;\ y \notin f\ `\ (A - \{x\})]\!] \Longrightarrow inj\text{-}on\ (f(x := y))\ A$
⟨*proof*⟩

lemma *less-handy-casesE*:
$[\![m < n;\ m = 0 \Longrightarrow R;\ \bigwedge m'\ n'.\ [\![n = Suc\ n';\ m = Suc\ m';\ m < n]\!] \Longrightarrow R]\!]$
$\Longrightarrow R$
⟨*proof*⟩

lemma *subset-drop-Diff-strg*:
$(A \subseteq C) \longrightarrow (A - B \subseteq C)$

⟨*proof*⟩

lemma *inj-case-bool*:
$inj\ (case\text{-}bool\ a\ b) = (a \neq b)$
⟨*proof*⟩

lemma *foldl-fun-upd*:
$foldl\ (\lambda s\ r.\ s\ (r := g\ r))\ f\ rs = (\lambda x.\ if\ x \in set\ rs\ then\ g\ x\ else\ f\ x)$
⟨*proof*⟩

lemma *all-rv-choice-fn-eq-pred*:
$[\![\bigwedge rv.\ P\ rv \Longrightarrow \exists fn.\ f\ rv = g\ fn\ ]\!] \Longrightarrow \exists fn.\ \forall rv.\ P\ rv \longrightarrow f\ rv = g\ (fn\ rv)$
⟨*proof*⟩

lemma *ex-const-function*:
$\exists f.\ \forall s.\ f\ (f'\ s) = v$
⟨*proof*⟩

lemma *if-Const-helper*:
$If\ P\ (Con\ x)\ (Con\ y) = Con\ (If\ P\ x\ y)$
⟨*proof*⟩

lemmas *if-Some-helper* = *if-Const-helper*[**where** *Con*=*Some*]

lemma *expand-restrict-map-eq*:
$(m\ |`\ S = m'\ |`\ S) = (\forall x.\ x \in S \longrightarrow m\ x = m'\ x)$
⟨*proof*⟩

lemma *disj-imp-rhs*:
$(P \Longrightarrow Q) \Longrightarrow (P \vee Q) = Q$
⟨*proof*⟩

lemma *remove1-filter*:
$distinct\ xs \Longrightarrow remove1\ x\ xs = filter\ (\lambda y.\ x \neq y)\ xs$
⟨*proof*⟩

lemma *Int-Union-empty*:
$(\bigwedge x.\ x \in S \Longrightarrow A \cap P\ x = \{\}) \Longrightarrow A \cap (\bigcup x \in S.\ P\ x) = \{\}$
⟨*proof*⟩

lemma *UN-Int-empty*:
$(\bigwedge x.\ x \in S \Longrightarrow P\ x \cap T = \{\}) \Longrightarrow (\bigcup x \in S.\ P\ x) \cap T = \{\}$
⟨*proof*⟩

lemma *disjointI*:
$[\![\bigwedge x\ y.\ [\![x \in A;\ y \in B]\!] \Longrightarrow x \neq y]\!] \Longrightarrow A \cap B = \{\}$
⟨*proof*⟩

lemma *UN-disjointI*:

assumes *rl*: $\bigwedge x\ y.\ [\![\ x \in A;\ y \in B\ ]\!] \Longrightarrow P\ x \cap Q\ y = \{\}$
shows $(\bigcup x \in A.\ P\ x) \cap (\bigcup x \in B.\ Q\ x) = \{\}$
⟨*proof*⟩

lemma *UN-set-member*:
assumes *sub*: $A \subseteq (\bigcup x \in S.\ P\ x)$
and *nz*: $A \neq \{\}$
shows $\exists x \in S.\ P\ x \cap A \neq \{\}$
⟨*proof*⟩

lemma *append-Cons-cases* [*consumes 1*, *case-names pre mid post*]:
$[\![(x,\ y) \in set\ (as\ @\ b\ \#\ bs);$
$(x,\ y) \in set\ as \Longrightarrow R;$
$[\![(x,\ y) \notin set\ as;\ (x,\ y) \notin set\ bs;\ (x,\ y) = b]\!] \Longrightarrow R;$
$(x,\ y) \in set\ bs \Longrightarrow R]\!] \Longrightarrow R$
⟨*proof*⟩

lemma *cart-singletons*:
$\{a\} \times \{b\} = \{(a,\ b)\}$
⟨*proof*⟩

lemma *disjoint-subset-neg1*:
$[\![\ B \cap C = \{\};\ A \subseteq B;\ A \neq \{\}\ ]\!] \Longrightarrow \neg\ A \subseteq C$
⟨*proof*⟩

lemma *disjoint-subset-neg2*:
$[\![\ B \cap C = \{\};\ A \subseteq C;\ A \neq \{\}\ ]\!] \Longrightarrow \neg\ A \subseteq B$
⟨*proof*⟩

lemma *iffE2*:
$[\![\ P = Q;\ [\![\ P;\ Q\ ]\!] \Longrightarrow R;\ [\![\ \neg\ P;\ \neg\ Q\ ]\!] \Longrightarrow R\ ]\!] \Longrightarrow R$
⟨*proof*⟩

lemma *list-case-If*:
$(case\ xs\ of\ []\ \Rightarrow P\ |\ -\ \Rightarrow Q) = (if\ xs = []\ then\ P\ else\ Q)$
⟨*proof*⟩

lemma *remove1-Nil-in-set*:
$[\![\ remove1\ x\ xs = []; xs \neq []\ ]\!] \Longrightarrow x \in set\ xs$
⟨*proof*⟩

lemma *remove1-empty*:
$(remove1\ v\ xs = []) = (xs = [v] \vee xs = [])$
⟨*proof*⟩

lemma *set-remove1*:
$x \in set\ (remove1\ y\ xs) \Longrightarrow x \in set\ xs$
⟨*proof*⟩

lemma *If-rearrage*:
$(\mathit{if}\ P\ \mathit{then}\ \mathit{if}\ Q\ \mathit{then}\ x\ \mathit{else}\ y\ \mathit{else}\ z) = (\mathit{if}\ P \wedge Q\ \mathit{then}\ x\ \mathit{else}\ \mathit{if}\ P\ \mathit{then}\ y\ \mathit{else}\ z)$
⟨*proof*⟩

lemma *disjI2-strg*:
$Q \longrightarrow (P \vee Q)$
⟨*proof*⟩

lemma *eq-imp-strg*:
$P\ t \longrightarrow (t = s \longrightarrow P\ s)$
⟨*proof*⟩

lemma *if-both-strengthen*:
$P \wedge Q \longrightarrow (\mathit{if}\ G\ \mathit{then}\ P\ \mathit{else}\ Q)$
⟨*proof*⟩

lemma *if-both-strengthen2*:
$P\ s \wedge Q\ s \longrightarrow (\mathit{if}\ G\ \mathit{then}\ P\ \mathit{else}\ Q)\ s$
⟨*proof*⟩

lemma *if-swap*:
$(\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ R) = (\mathit{if}\ \neg P\ \mathit{then}\ R\ \mathit{else}\ Q)$ ⟨*proof*⟩

lemma *imp-consequent*:
$P \longrightarrow Q \longrightarrow P$ ⟨*proof*⟩

lemma *list-case-helper*:
$xs \neq [] \Longrightarrow \mathit{case\text{-}list}\ f\ g\ xs = g\ (\mathit{hd}\ xs)\ (\mathit{tl}\ xs)$
⟨*proof*⟩

lemma *list-cons-rewrite*:
$(\forall x\ xs.\ L = x \#\ xs \longrightarrow P\ x\ xs) = (L \neq [] \longrightarrow P\ (\mathit{hd}\ L)\ (\mathit{tl}\ L))$
⟨*proof*⟩

lemma *list-not-Nil-manip*:
$[\![\ xs = y \#\ ys;\ \mathit{case}\ xs\ \mathit{of}\ [] \Rightarrow \mathit{False} \mid (y \#\ ys) \Rightarrow P\ y\ ys\ ]\!] \Longrightarrow P\ y\ ys$
⟨*proof*⟩

lemma *ran-ball-triv*:
$\bigwedge P\ m\ S.\ [\![\ \forall x \in (\mathit{ran}\ S).\ P\ x\ ;\ m \in (\mathit{ran}\ S)\ ]\!] \Longrightarrow P\ m$
⟨*proof*⟩

lemma *singleton-tuple-cartesian*:
$(\{(a,\ b)\} = S \times T) = (\{a\} = S \wedge \{b\} = T)$
$(S \times T = \{(a,\ b)\}) = (\{a\} = S \wedge \{b\} = T)$
⟨*proof*⟩

lemma *strengthen-ignore-if*:
$A\ s \wedge B\ s \longrightarrow (\mathit{if}\ P\ \mathit{then}\ A\ \mathit{else}\ B)\ s$

⟨*proof*⟩

lemma *case-sum-True* :
(*case r of Inl a ⇒ True | Inr b ⇒ f b*) = (∀ *b*. *r* = *Inr b* ⟶ *f b*)
⟨*proof*⟩

lemma *sym-ex-elim*:
F x = *y* ⟹ ∃ *x*. *y* = *F x*
⟨*proof*⟩

lemma *tl-drop-1* :
tl xs = *drop 1 xs*
⟨*proof*⟩

lemma *upt-lhs-sub-map*:
[*x* ..< *y*] = *map* ((+) *x*) [*0* ..< *y* − *x*]
⟨*proof*⟩

lemma *upto-0-to-4*:
[*0*..<*4*] = *0* # [*1*..<*4*]
⟨*proof*⟩

lemma *disjEI*:
⟦ *P* ∨ *Q*; *P* ⟹ *R*; *Q* ⟹ *S* ⟧
⟹ *R* ∨ *S*
⟨*proof*⟩

lemma *dom-fun-upd2*:
s x = *Some z* ⟹ *dom* (*s* (*x* ↦ *y*)) = *dom s*
⟨*proof*⟩

lemma *foldl-True* :
foldl (∨) *True bs*
⟨*proof*⟩

lemma *image-set-comp*:
f ‘ {*g x* | *x*. *Q x*} = (*f* ∘ *g*) ‘ {*x*. *Q x*}
⟨*proof*⟩

lemma *mutual-exE*:
⟦ ∃ *x*. *P x*; ⋀*x*. *P x* ⟹ *Q x* ⟧ ⟹ ∃ *x*. *Q x*
⟨*proof*⟩

lemma *nat-diff-eq*:
fixes *x* :: *nat*
shows ⟦ *x* − *y* = *x* − *z*; *y* < *x*⟧ ⟹ *y* = *z*
⟨*proof*⟩

lemma *comp-upd-simp*:

$(f \circ (g\ (x := y))) = ((f \circ g)\ (x := f\ y))$
⟨*proof*⟩

lemma *dom-option-map*:
$dom\ (map\text{-}option\ f\ o\ m) = dom\ m$
⟨*proof*⟩

lemma *drop-imp*:
$P \Longrightarrow (A \longrightarrow P) \wedge (B \longrightarrow P)$ ⟨*proof*⟩

lemma *inj-on-fun-updI2*:
$[\![\ inj\text{-}on\ f\ A;\ y \notin f\ `\ (A - \{x\})\ ]\!] \Longrightarrow inj\text{-}on\ (f(x := y))\ A$
⟨*proof*⟩

lemma *inj-on-fun-upd-elsewhere*:
$x \notin S \Longrightarrow inj\text{-}on\ (f\ (x := y))\ S = inj\text{-}on\ f\ S$
⟨*proof*⟩

lemma *not-Some-eq-tuple*:
$(\forall\, y\ z.\ x \neq Some\ (y,\ z)) = (x = None)$
⟨*proof*⟩

lemma *ran-option-map*:
$ran\ (map\text{-}option\ f\ o\ m) = f\ `\ ran\ m$
⟨*proof*⟩

lemma *All-less-Ball*:
$(\forall\, x < n.\ P\ x) = (\forall\, x{\in}\{..<\ n\}.\ P\ x)$
⟨*proof*⟩

lemma *Int-image-empty*:
$[\![\ \bigwedge x\ y.\ f\ x \neq g\ y\ ]\!]$
$\quad \Longrightarrow f\ `\ S \cap g\ `\ T = \{\}$
⟨*proof*⟩

lemma *Max-prop*:
$[\![\ Max\ S \in S \Longrightarrow P\ (Max\ S);\ (S :: ('a :: \{finite,\ linorder\})\ set) \neq \{\}\ ]\!] \Longrightarrow P\ (Max\ S)$
⟨*proof*⟩

lemma *Min-prop*:
$[\![\ Min\ S \in S \Longrightarrow P\ (Min\ S);\ (S :: ('a :: \{finite,\ linorder\})\ set) \neq \{\}\ ]\!] \Longrightarrow P\ (Min\ S)$
⟨*proof*⟩

lemma *findSomeD*:
$find\ P\ xs = Some\ x \Longrightarrow P\ x \wedge x \in set\ xs$
⟨*proof*⟩

lemma *findNoneD*:
find P xs = None $\Longrightarrow \forall x \in set\ xs.\ \neg P\ x$
⟨*proof*⟩

lemma *dom-upd*:
dom ($\lambda x.$ *if* $x = y$ *then None else f x*) = *dom f* − {*y*}
⟨*proof*⟩

definition
is-inv :: (′*a* ⇀ ′*b*) ⇒ (′*b* ⇀ ′*a*) ⇒ *bool* **where**
is-inv f g ≡ *ran f* = *dom g* ∧ ($\forall x\ y.$ *f x* = *Some y* ⟶ *g y* = *Some x*)

lemma *is-inv-NoneD*:
assumes *g x* = *None*
assumes *is-inv f g*
shows *x* ∉ *ran f*
⟨*proof*⟩

lemma *is-inv-SomeD*:
⟦ *f x* = *Some y*; *is-inv f g* ⟧ ⟹ *g y* = *Some x*
⟨*proof*⟩

lemma *is-inv-com*:
is-inv f g ⟹ *is-inv g f*
⟨*proof*⟩

lemma *is-inv-inj*:
is-inv f g ⟹ *inj-on f* (*dom f*)
⟨*proof*⟩

lemma *ran-upd′*:
⟦*inj-on f* (*dom f*); *f y* = *Some z*⟧ ⟹ *ran* (*f* (*y* := *None*)) = *ran f* − {*z*}
⟨*proof*⟩

lemma *is-inv-None-upd*:
⟦ *is-inv f g*; *g x* = *Some y*⟧ ⟹ *is-inv* (*f*(*y* := *None*)) (*g*(*x* := *None*))
⟨*proof*⟩

lemma *is-inv-inj2*:
is-inv f g ⟹ *inj-on g* (*dom g*)
⟨*proof*⟩

Map inversion (implicitly assuming injectivity).

definition
the-inv-map m = ($\lambda s.$ *if s*∈*ran m then Some* (*THE x. m x* = *Some s*) *else None*)

Map inversion can be expressed by function inversion.

lemma *the-inv-map-def2*:

the-inv-map m = (*Some* ∘ *the-inv-into* (*dom m*) (*the* ∘ *m*)) |‘ (*ran m*)
⟨*proof*⟩

The domain of a function composition with Some is the universal set.

lemma *dom-comp-Some*[*simp*]: *dom* (*comp Some f*) = *UNIV* ⟨*proof*⟩

Assuming injectivity, map inversion produces an inversive map.

lemma *is-inv-the-inv-map*:
inj-on m (*dom m*) ⟹ *is-inv m* (*the-inv-map m*)
⟨*proof*⟩

lemma *the-the-inv-mapI*:
inj-on m (*dom m*) ⟹ *m x* = *Some y* ⟹ *the* (*the-inv-map m y*) = *x*
⟨*proof*⟩

lemma *eq-restrict-map-None*:
restrict-map m A x = *None* ⟷ *x* ~: (*A* ∩ *dom m*)
⟨*proof*⟩

lemma *eq-the-inv-map-None*[*simp*]: *the-inv-map m x* = *None* ⟷ *x*∉*ran m*
⟨*proof*⟩

lemma *is-inv-unique*:
is-inv f g ⟹ *is-inv f h* ⟹ *g*=*h*
⟨*proof*⟩

lemma *range-convergence1*:
⟦ ∀ *z*. *x* < *z* ∧ *z* ≤ *y* ⟶ *P z*; ∀ *z* > *y*. *P* (*z* :: *'a* :: *linorder*) ⟧ ⟹ ∀ *z* > *x*. *P z*
⟨*proof*⟩

lemma *range-convergence2*:
⟦ ∀ *z*. *x* < *z* ∧ *z* ≤ *y* ⟶ *P z*; ∀ *z*. *z* > *y* ∧ *z* < *w* ⟶ *P* (*z* :: *'a* :: *linorder*) ⟧
⟹ ∀ *z*. *z* > *x* ∧ *z* < *w* ⟶ *P z*
⟨*proof*⟩

lemma *zip-upt-Cons*:
a < *b* ⟹ *zip* [*a* ..< *b*] (*x* # *xs*) = (*a*, *x*) # *zip* [*Suc a* ..< *b*] *xs*
⟨*proof*⟩

lemma *map-comp-eq*:
f ∘$_m$ *g* = *case-option None f* ∘ *g*
⟨*proof*⟩

lemma *dom-If-Some*:
dom (λ*x*. *if x* ∈ *S then Some v else f x*) = (*S* ∪ *dom f*)
⟨*proof*⟩

lemma *foldl-fun-upd-const*:
foldl (λ*s x*. *s*(*f x* := *v*)) *s xs*

$= (\lambda x.\ \mathit{if}\ x \in f\ `\ \mathit{set}\ xs\ \mathit{then}\ v\ \mathit{else}\ s\ x)$
⟨*proof*⟩

lemma *foldl-id*:
$\mathit{foldl}\ (\lambda s\ x.\ s)\ s\ xs = s$
⟨*proof*⟩

lemma *SucSucMinus*: $2 \leq n \Longrightarrow \mathit{Suc}\ (\mathit{Suc}\ (n - 2)) = n$ ⟨*proof*⟩

lemma *ball-to-all*:
$(\bigwedge x.\ (x \in A) = (P\ x)) \Longrightarrow (\forall x \in A.\ B\ x) = (\forall x.\ P\ x \longrightarrow B\ x)$
⟨*proof*⟩

lemma *case-option-If*:
$\mathit{case\text{-}option}\ P\ (\lambda x.\ Q)\ v = (\mathit{if}\ v = \mathit{None}\ \mathit{then}\ P\ \mathit{else}\ Q)$
⟨*proof*⟩

lemma *case-option-If2*:
$\mathit{case\text{-}option}\ P\ Q\ v = \mathit{If}\ (v \neq \mathit{None})\ (Q\ (\mathit{the}\ v))\ P$
⟨*proof*⟩

lemma *if3-fold*:
$(\mathit{if}\ P\ \mathit{then}\ x\ \mathit{else}\ \mathit{if}\ Q\ \mathit{then}\ y\ \mathit{else}\ x) = (\mathit{if}\ P \vee \neg\ Q\ \mathit{then}\ x\ \mathit{else}\ y)$
⟨*proof*⟩

lemma *rtrancl-insert*:
assumes *x-new*: $\bigwedge y.\ (x,y) \notin R$
shows $R\hat{}* ``\ \mathit{insert}\ x\ S = \mathit{insert}\ x\ (R\hat{}* ``\ S)$
⟨*proof*⟩

lemma *ran-del-subset*:
$y \in \mathit{ran}\ (f\ (x := \mathit{None})) \Longrightarrow y \in \mathit{ran}\ f$
⟨*proof*⟩

lemma *trancl-sub-lift*:
assumes *sub*: $\bigwedge p\ p'.\ (p,p') \in r \Longrightarrow (p,p') \in r'$
shows $(p,p') \in r\hat{}+ \Longrightarrow (p,p') \in r'\hat{}+$
⟨*proof*⟩

lemma *trancl-step-lift*:
assumes *x-step*: $\bigwedge p\ p'.\ (p,p') \in r' \Longrightarrow (p,p') \in r \vee (p = x \wedge p' = y)$
assumes *y-new*: $\bigwedge p'.\ \neg(y,p') \in r$
shows $(p,p') \in r'\hat{}+ \Longrightarrow (p,p') \in r\hat{}+ \vee ((p,x) \in r\hat{}+ \wedge p' = y) \vee (p = x \wedge p' = y)$
⟨*proof*⟩

lemma *rtrancl-simulate-weak*:
assumes *r*: $(x,z) \in R^*$
assumes *s*: $\bigwedge y.\ (x,y) \in R \Longrightarrow (y,z) \in R^* \Longrightarrow (x,y) \in R' \wedge (y,z) \in R'^*$

shows $(x,z) \in R'^*$
⟨*proof*⟩

lemma *list-case-If2*:
$case\text{-}list\ f\ g\ xs = If\ (xs = [])\ f\ (g\ (hd\ xs)\ (tl\ xs))$
⟨*proof*⟩

lemma *length-ineq-not-Nil*:
$length\ xs > n \Longrightarrow xs \neq []$
$length\ xs \geq n \Longrightarrow n \neq 0 \longrightarrow xs \neq []$
$\neg\ length\ xs < n \Longrightarrow n \neq 0 \longrightarrow xs \neq []$
$\neg\ length\ xs \leq n \Longrightarrow xs \neq []$
⟨*proof*⟩

lemma *numeral-eqs*:
$2 = Suc\ (Suc\ 0)$
$3 = Suc\ (Suc\ (Suc\ 0))$
$4 = Suc\ (Suc\ (Suc\ (Suc\ 0)))$
$5 = Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0))))$
$6 = Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0)))))$
⟨*proof*⟩

lemma *psubset-singleton*:
$(S \subset \{x\}) = (S = \{\})$
⟨*proof*⟩

lemma *length-takeWhile-ge*:
$length\ (takeWhile\ f\ xs) = n \Longrightarrow length\ xs = n \vee (length\ xs > n \wedge \neg\ f\ (xs\ !\ n))$
⟨*proof*⟩

lemma *length-takeWhile-le*:
$\neg\ f\ (xs\ !\ n) \Longrightarrow length\ (takeWhile\ f\ xs) \leq n$
⟨*proof*⟩

lemma *length-takeWhile-gt*:
$n < length\ (takeWhile\ f\ xs)$
$\Longrightarrow (\exists\ ys\ zs.\ length\ ys = Suc\ n \wedge xs = ys\ @\ zs \wedge takeWhile\ f\ xs = ys\ @\ takeWhile\ f\ zs)$
⟨*proof*⟩

lemma *hd-drop-conv-nth2*:
$n < length\ xs \Longrightarrow hd\ (drop\ n\ xs) = xs\ !\ n$
⟨*proof*⟩

lemma *map-upt-eq-vals-D*:
$[\![\ map\ f\ [0\ ..<\ n] = ys;\ m < length\ ys\ ]\!] \Longrightarrow f\ m = ys\ !\ m$
⟨*proof*⟩

lemma *length-le-helper*:

$[\![n \leq \mathit{length}\ \mathit{xs};\ n \neq 0]\!] \Longrightarrow \mathit{xs} \neq [] \wedge n - 1 \leq \mathit{length}\ (\mathit{tl}\ \mathit{xs})$
⟨*proof*⟩

lemma *all-ex-eq-helper*:
$(\forall v.\ (\exists v'.\ v = f\ v' \wedge P\ v\ v') \longrightarrow Q\ v)$
$\quad = (\forall v'.\ P\ (f\ v')\ v' \longrightarrow Q\ (f\ v'))$
⟨*proof*⟩

lemma *nat-less-cases'*:
$(x::nat) < y \Longrightarrow x = y - 1 \vee x < y - 1$
⟨*proof*⟩

lemma *less-numeral-nat-iff-disj*:
$(n::nat) < \mathit{numeral}\ m \longleftrightarrow n = \mathit{numeral}\ m - 1 \vee n < \mathit{numeral}\ m - 1$
⟨*proof*⟩

lemma *filter-to-shorter-upto*:
$n \leq m \Longrightarrow \mathit{filter}\ (\lambda x.\ x < n)\ [0\ ..<\ m] = [0\ ..<\ n]$
⟨*proof*⟩

lemma *in-emptyE*: $[\![A = \{\};\ x \in A]\!] \Longrightarrow P$ ⟨*proof*⟩

lemma *Ball-emptyI*:
$S = \{\} \Longrightarrow (\forall x \in S.\ P\ x)$
⟨*proof*⟩

lemma *allfEI*:
$[\![\forall x.\ P\ x;\ \bigwedge x.\ P\ (f\ x) \Longrightarrow Q\ x]\!] \Longrightarrow \forall x.\ Q\ x$
⟨*proof*⟩

lemma *cart-singleton-empty2*:
$(\{x\} \times S = \{\}) = (S = \{\})$
$(\{\} = S \times \{e\}) = (S = \{\})$
⟨*proof*⟩

lemma *cases-simp-conj*:
$((P \longrightarrow Q) \wedge (\neg\ P \longrightarrow Q) \wedge R) = (Q \wedge R)$
⟨*proof*⟩

lemma *domE* :
$[\![x \in \mathit{dom}\ m;\ \bigwedge r.\ [\![m\ x = \mathit{Some}\ r]\!] \Longrightarrow P]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *dom-eqD*:
$[\![f\ x = \mathit{Some}\ v;\ \mathit{dom}\ f = S]\!] \Longrightarrow x \in S$
⟨*proof*⟩

lemma *exception-set-finite1*:
$\mathit{finite}\ \{x.\ P\ x\} \Longrightarrow \mathit{finite}\ \{x.\ (x = y \longrightarrow Q\ x) \wedge P\ x\}$

⟨*proof*⟩

lemma *exception-set-finite2*:
finite {*x*. *P x*} ⟹ *finite* {*x*. *x* ≠ *y* ⟶ *P x*}
⟨*proof*⟩

lemmas *exception-set-finite* = *exception-set-finite1 exception-set-finite2*

lemma *exfEI*:
⟦ ∃ *x*. *P x*; ⋀*x*. *P x* ⟹ *Q* (*f x*) ⟧ ⟹ ∃ *x*. *Q x*
⟨*proof*⟩

lemma *Collect-int-vars*:
{*s*. *P rv s*} ∩ {*s*. *rv* = *xf s*} = {*s*. *P* (*xf s*) *s*} ∩ {*s*. *rv* = *xf s*}
⟨*proof*⟩

lemma *if-0-1-eq*:
((*if P then 1 else 0*) = (*case Q of True* ⇒ *of-nat 1* | *False* ⇒ *of-nat 0*)) = (*P* = *Q*)
⟨*proof*⟩

lemma *modify-map-exists-cte* :
(∃ *cte*. *modify-map m p f p'* = *Some cte*) = (∃ *cte*. *m p'* = *Some cte*)
⟨*proof*⟩

lemma *dom-eqI*:
assumes *c1*: ⋀*x y*. *P x* = *Some y* ⟹ ∃ *y*. *Q x* = *Some y*
and *c2*: ⋀*x y*. *Q x* = *Some y* ⟹ ∃ *y*. *P x* = *Some y*
shows *dom P* = *dom Q*
⟨*proof*⟩

lemma *dvd-reduce-multiple*:
fixes *k* :: *nat*
shows (*k dvd k* ∗ *m* + *n*) = (*k dvd n*)
⟨*proof*⟩

lemma *image-iff2*:
inj f ⟹ *f x* ∈ *f* ‘ *S* = (*x* ∈ *S*)
⟨*proof*⟩

lemma *map-comp-restrict-map-Some-iff*:
((*g* ∘*m* (*m* |‘ *S*)) *x* = *Some y*) = ((*g* ∘*m m*) *x* = *Some y* ∧ *x* ∈ *S*)
⟨*proof*⟩

lemma *range-subsetD*:
fixes *a* :: ′*a* :: *order*
shows ⟦ {*a*..*b*} ⊆ {*c*..*d*}; *a* ≤ *b* ⟧ ⟹ *c* ≤ *a* ∧ *b* ≤ *d*
⟨*proof*⟩

lemma *case-option-dom*:
(*case f x of None ⇒ a | Some v ⇒ b v*) = (*if x ∈ dom f then b* (*the* (*f x*)) *else a*)
⟨*proof*⟩

lemma *contrapos-imp*:
P ⟶ *Q* ⟹ ¬ *Q* ⟶ ¬ *P*
⟨*proof*⟩

lemma *filter-eq-If*:
distinct xs ⟹ *filter* (λ*v*. *v* = *x*) *xs* = (*if x* ∈ *set xs then* [*x*] *else* [])
⟨*proof*⟩

lemma (**in** *semigroup-add*) *foldl-assoc*:
shows *foldl* (+) (*x*+*y*) *zs* = *x* + (*foldl* (+) *y zs*)
⟨*proof*⟩

lemma (**in** *monoid-add*) *foldl-absorb0*:
shows *x* + (*foldl* (+) *0 zs*) = *foldl* (+) *x zs*
⟨*proof*⟩

lemma *foldl-conv-concat*:
foldl (@) *xs xss* = *xs* @ *concat xss*
⟨*proof*⟩

lemma *foldl-concat-concat*:
foldl (@) [] (*xs* @ *ys*) = *foldl* (@) [] *xs* @ *foldl* (@) [] *ys*
⟨*proof*⟩

lemma *foldl-does-nothing*:
⟦ ⋀*x*. *x* ∈ *set xs* ⟹ *f s x* = *s* ⟧ ⟹ *foldl f s xs* = *s*
⟨*proof*⟩

lemma *foldl-use-filter*:
⟦ ⋀*v x*. ⟦ ¬ *g x*; *x* ∈ *set xs* ⟧ ⟹ *f v x* = *v* ⟧ ⟹ *foldl f v xs* = *foldl f v* (*filter g xs*)
⟨*proof*⟩

lemma *map-comp-update-lift*:
assumes *fv*: *f v* = *Some v′*
shows (*f* ∘$_m$ (*g*(*ptr* ↦ *v*))) = ((*f* ∘$_m$ *g*)(*ptr* ↦ *v′*))
⟨*proof*⟩

lemma *restrict-map-cong*:
assumes *sv*: *S* = *S′*
and *rl*: ⋀*p*. *p* ∈ *S′* ⟹ *mp p* = *mp′ p*
shows *mp* |‘ *S* = *mp′* |‘ *S′*
⟨*proof*⟩

lemma *case-option-over-if*:

case-option P Q (if G then None else Some v)
= (if G then P else Q v)
case-option P Q (if G then Some v else None)
= (if G then Q v else P)
⟨*proof*⟩

lemma *map-length-cong*:
⟦ *length xs = length ys*; ⋀*x y*. *(x, y) ∈ set (zip xs ys)* ⟹ *f x = g y* ⟧
⟹ *map f xs = map g ys*
⟨*proof*⟩

lemma *take-min-len*:
take (min (length xs) n) xs = take n xs
⟨*proof*⟩

lemmas *interval-empty = atLeastatMost-empty-iff*

lemma *fold-and-false*[*simp*]:
¬(*fold (∧) xs False*)
⟨*proof*⟩

lemma *fold-and-true*:
fold (∧) xs True ⟹ ∀ *i < length xs*. *xs ! i*
⟨*proof*⟩

lemma *fold-or-true*[*simp*]:
fold (∨) xs True
⟨*proof*⟩

lemma *fold-or-false*:
¬(*fold (∨) xs False*) ⟹ ∀ *i < length xs*. ¬(*xs ! i*)
⟨*proof*⟩

2.1 Take, drop, zip, *list-all* etc rules

method *two-induct* **for** *xs ys* =
((*induct xs arbitrary*: *ys*; *simp?*), (*case-tac ys*; *simp*)*?*)

lemma *map-fst-zip-prefix*:
map fst (zip xs ys) ≤ xs
⟨*proof*⟩

lemma *map-snd-zip-prefix*:
map snd (zip xs ys) ≤ ys
⟨*proof*⟩

lemma *nth-upt-0* [*simp*]:
i < length xs ⟹ *[0..<length xs] ! i = i*
⟨*proof*⟩

lemma *take-insert-nth*:
i < *length xs*⟹ *insert* (*xs* ! *i*) (*set* (*take i xs*)) = *set* (*take* (*Suc i*) *xs*)
⟨*proof*⟩

lemma *zip-take-drop*:
⟦*n* < *length xs*; *length ys* = *length xs*⟧ ⟹
zip xs (*take n ys* @ *a* # *drop* (*Suc n*) *ys*) =
zip (*take n xs*) (*take n ys*) @ (*xs* ! *n*, *a*) # *zip* (*drop* (*Suc n*) *xs*) (*drop* (*Suc n*) *ys*)
⟨*proof*⟩

lemma *take-nth-distinct*:
⟦*distinct xs*; *n* < *length xs*; *xs* ! *n* ∈ *set* (*take n xs*)⟧ ⟹ *False*
⟨*proof*⟩

lemma *take-drop-append*:
drop a xs = *take b* (*drop a xs*) @ *drop* (*a* + *b*) *xs*
⟨*proof*⟩

lemma *drop-take-drop*:
drop a (*take* (*b* + *a*) *xs*) @ *drop* (*b* + *a*) *xs* = *drop a xs*
⟨*proof*⟩

lemma *not-prefixI*:
⟦ *xs* ≠ *ys*; *length xs* = *length ys*⟧ ⟹ ¬ *xs* ≤ *ys*
⟨*proof*⟩

lemma *map-fst-zip′*:
length xs ≤ *length ys* ⟹ *map fst* (*zip xs ys*) = *xs*
⟨*proof*⟩

lemma *zip-take-triv*:
n ≥ *length bs* ⟹ *zip* (*take n as*) *bs* = *zip as bs*
⟨*proof*⟩

lemma *zip-take-triv2*:
length as ≤ *n* ⟹ *zip as* (*take n bs*) = *zip as bs*
⟨*proof*⟩

lemma *zip-take-length*:
zip xs (*take* (*length xs*) *ys*) = *zip xs ys*
⟨*proof*⟩

lemma *zip-singleton*:
ys ≠ [] ⟹ *zip* [*a*] *ys* = [(*a*, *ys* ! *0*)]
⟨*proof*⟩

lemma *zip-append-singleton*:

⟦*i* = *length xs*; *length xs* < *length ys*⟧ ⟹ *zip* (*xs* @ [*a*]) *ys* = (*zip xs ys*) @ [(*a*,*ys* ! *i*)]
⟨*proof*⟩

lemma *ranE*:
⟦ *v* ∈ *ran f*; ⋀*x*. *f x* = *Some v* ⟹ *R*⟧ ⟹ *R*
⟨*proof*⟩

lemma *ran-map-option-restrict-eq*:
⟦ *x* ∈ *ran* (*map-option f o g*); *x* ∉ *ran* (*map-option f o* (*g* |` (− {*y*}))) ⟧
⟹ ∃ *v*. *g y* = *Some v* ∧ *f v* = *x*
⟨*proof*⟩

lemma *map-of-zip-range*:
⟦*length xs* = *length ys*; *distinct xs*⟧ ⟹ (λ*x*. (*the* (*map-of* (*zip xs ys*) *x*))) ` *set xs* = *set ys*
⟨*proof*⟩

lemma *map-zip-fst*:
length xs = *length ys* ⟹ *map* (λ(*x*, *y*). *f x*) (*zip xs ys*) = *map f xs*
⟨*proof*⟩

lemma *map-zip-fst'*:
length xs ≤ *length ys* ⟹ *map* (λ(*x*, *y*). *f x*) (*zip xs ys*) = *map f xs*
⟨*proof*⟩

lemma *map-zip-snd*:
length xs = *length ys* ⟹ *map* (λ(*x*, *y*). *f y*) (*zip xs ys*) = *map f ys*
⟨*proof*⟩

lemma *map-zip-snd'*:
length ys ≤ *length xs* ⟹ *map* (λ(*x*, *y*). *f y*) (*zip xs ys*) = *map f ys*
⟨*proof*⟩

lemma *map-of-zip-tuple-in*:
⟦(*x*, *y*) ∈ *set* (*zip xs ys*); *distinct xs*⟧ ⟹ *map-of* (*zip xs ys*) *x* = *Some y*
⟨*proof*⟩

lemma *in-set-zip1*:
(*x*, *y*) ∈ *set* (*zip xs ys*) ⟹ *x* ∈ *set xs*
⟨*proof*⟩

lemma *in-set-zip2*:
(*x*, *y*) ∈ *set* (*zip xs ys*) ⟹ *y* ∈ *set ys*
⟨*proof*⟩

lemma *map-zip-snd-take*:
map (λ(*x*, *y*). *f y*) (*zip xs ys*) = *map f* (*take* (*length xs*) *ys*)
⟨*proof*⟩

lemma *map-of-zip-is-index*:
⟦*length xs* = *length ys*; *x* ∈ *set xs*⟧ ⟹ ∃ *i*. (*map-of* (*zip xs ys*)) *x* = *Some* (*ys* ! *i*)
⟨*proof*⟩

lemma *map-of-zip-take-update*:
⟦*i* < *length xs*; *length xs* ≤ *length ys*; *distinct xs*⟧
⟹ (*map-of* (*zip* (*take i xs*) *ys*))(*xs* ! *i* ↦ (*ys* ! *i*)) = *map-of* (*zip* (*take* (*Suc i*) *xs*) *ys*)
⟨*proof*⟩

lemma *map-of-zip-is-Some'*:
length xs ≤ *length ys* ⟹ (*x* ∈ *set xs*) = (∃ *y*. *map-of* (*zip xs ys*) *x* = *Some y*)
⟨*proof*⟩

lemma *map-of-zip-inj*:
⟦*distinct xs*; *distinct ys*; *length xs* = *length ys*⟧
⟹ *inj-on* (λ*x*. (*the* (*map-of* (*zip xs ys*) *x*))) (*set xs*)
⟨*proof*⟩

lemma *map-of-zip-inj'*:
⟦*distinct xs*; *distinct ys*; *length xs* ≤ *length ys*⟧
⟹ *inj-on* (λ*x*. (*the* (*map-of* (*zip xs ys*) *x*))) (*set xs*)
⟨*proof*⟩

lemma *list-all-nth*:
⟦*list-all P xs*; *i* < *length xs*⟧ ⟹ *P* (*xs* ! *i*)
⟨*proof*⟩

lemma *list-all-update*:
⟦*list-all P xs*; *i* < *length xs*; ⋀*x*. *P x* ⟹ *P* (*f x*)⟧
⟹ *list-all P* (*xs* [*i* := *f* (*xs* ! *i*)])
⟨*proof*⟩

lemma *list-allI*:
⟦*list-all P xs*; ⋀*x*. *P x* ⟹ *P'* *x*⟧ ⟹ *list-all P'* *xs*
⟨*proof*⟩

lemma *list-all-imp-filter*:
list-all (λ*x*. *f x* ⟶ *g x*) *xs* = *list-all* (λ*x*. *g x*) [*x*←*xs* . *f x*]
⟨*proof*⟩

lemma *list-all-imp-filter2*:
list-all (λ*x*. *f x* ⟶ *g x*) *xs* = *list-all* (λ*x*. ¬*f x*) [*x*←*xs* . (λ*x*. ¬*g x*) *x*]
⟨*proof*⟩

lemma *list-all-imp-chain*:

⟦*list-all* (λ*x*. *f x* ⟶ *g x*) *xs*; *list-all* (λ*x*. *f′ x* ⟶ *f x*) *xs*⟧
⟹ *list-all* (λ*x*. *f′ x* ⟶ *g x*) *xs*
⟨*proof*⟩

lemma *inj-Pair*:
inj-on (*Pair x*) *S*
⟨*proof*⟩

lemma *inj-on-split*:
inj-on f S ⟹ *inj-on* (λ*x*. (*z*, *f x*)) *S*
⟨*proof*⟩

lemma *split-state-strg*:
(∃ *x*. *f s* = *x* ∧ *P x s*) ⟶ *P* (*f s*) *s* ⟨*proof*⟩

lemma *theD*:
⟦*the* (*f x*) = *y*; *x* ∈ *dom f* ⟧ ⟹ *f x* = *Some y*
⟨*proof*⟩

lemma *bspec-split*:
⟦ ∀ (*a*, *b*) ∈ *S*. *P a b*; (*a*, *b*) ∈ *S* ⟧ ⟹ *P a b*
⟨*proof*⟩

lemma *set-zip-same*:
set (*zip xs xs*) = *Id* ∩ (*set xs* × *set xs*)
⟨*proof*⟩

lemma *ball-ran-updI*:
(∀ *x* ∈ *ran m*. *P x*) ⟹ *P v* ⟹ (∀ *x* ∈ *ran* (*m* (*y* ↦ *v*)). *P x*)
⟨*proof*⟩

lemma *not-psubset-eq*:
⟦ ¬ *A* ⊂ *B*; *A* ⊆ *B* ⟧ ⟹ *A* = *B*
⟨*proof*⟩

lemma *set-as-imp*:
(*A* ∩ *P* ∪ *B* ∩ −*P*) = {*s*. (*s* ∈ *P* ⟶ *s* ∈ *A*) ∧ (*s* ∉ *P* ⟶ *s* ∈ *B*)}
⟨*proof*⟩

lemma *in-image-op-plus*:
(*x* + *y* ∈ (+) *x* ‘ *S*) = ((*y* :: *′a* :: *ring*) ∈ *S*)
⟨*proof*⟩

lemma *insert-subtract-new*:

$x \notin S \Longrightarrow (insert\ x\ S - S) = \{x\}$
⟨*proof*⟩

lemmas *zip-is-empty = zip-eq-Nil-iff*

lemma *minus-Suc-0-lt*:
$a \neq 0 \Longrightarrow a - Suc\ 0 < a$
⟨*proof*⟩

lemma *fst-last-zip-upt*:
$zip\ [0\ ..<\ m]\ xs \neq [] \Longrightarrow$
$fst\ (last\ (zip\ [0\ ..<\ m]\ xs)) = (if\ length\ xs < m\ then\ length\ xs - 1\ else\ m - 1)$
⟨*proof*⟩

lemma *neq-into-nprefix*:
$[\![\ x \neq take\ (length\ x)\ y\ ]\!] \Longrightarrow \neg\ x \leq y$
⟨*proof*⟩

lemma *suffix-eqI*:
$[\![\ suffix\ xs\ as;\ suffix\ xs\ bs;\ length\ as = length\ bs;$
$take\ (length\ as - length\ xs)\ as \leq take\ (length\ bs - length\ xs)\ bs]\!] \Longrightarrow as = bs$
⟨*proof*⟩

lemma *suffix-Cons-mem*:
$suffix\ (x\ \#\ xs)\ as \Longrightarrow x \in set\ as$
⟨*proof*⟩

lemma *distinct-imply-not-in-tail*:
$[\![\ distinct\ list;\ suffix\ (y\ \#\ ys)\ list]\!] \Longrightarrow y \notin set\ ys$
⟨*proof*⟩

lemma *list-induct-suffix* [*case-names Nil Cons*]:
assumes *nilr*: $P\ []$
and *consr*: $\bigwedge x\ xs.\ [\![P\ xs;\ suffix\ (x\ \#\ xs)\ as\ ]\!] \Longrightarrow P\ (x\ \#\ xs)$
shows $P\ as$
⟨*proof*⟩

Parallel etc. and lemmas for list prefix

lemma *prefix-induct* [*consumes 1*, *case-names Nil Cons*]:
fixes *prefix*
assumes *np*: $prefix \leq lst$
and *base*: $\bigwedge xs.\ P\ []\ xs$
and *rl*: $\bigwedge x\ xs\ y\ ys.\ [\![\ x = y;\ xs \leq ys;\ P\ xs\ ys\ ]\!] \Longrightarrow P\ (x\#xs)\ (y\#ys)$
shows *P prefix lst*
⟨*proof*⟩

lemma *not-prefix-cases*:
fixes *prefix*
assumes *pfx*: $\neg\ prefix \leq lst$

and *c1*: $[\![\ prefix \neq [];\ lst = []\ ]\!] \Longrightarrow R$
and *c2*: $\bigwedge a\ as\ x\ xs.\ [\![\ prefix = a\#as;\ lst = x\#xs;\ x = a;\ \neg\ as \leq xs]\!] \Longrightarrow R$
and *c3*: $\bigwedge a\ as\ x\ xs.\ [\![\ prefix = a\#as;\ lst = x\#xs;\ x \neq a]\!] \Longrightarrow R$
shows *R*
⟨*proof*⟩

lemma *not-prefix-induct* [*consumes 1*, *case-names Nil Neq Eq*]:
fixes *prefix*
assumes *np*: $\neg\ prefix \leq lst$
and *base*: $\bigwedge x\ xs.\ P\ (x\#xs)\ []$
and *r1*: $\bigwedge x\ xs\ y\ ys.\ x \neq y \Longrightarrow P\ (x\#xs)\ (y\#ys)$
and *r2*: $\bigwedge x\ xs\ y\ ys.\ [\![\ x = y;\ \neg\ xs \leq ys;\ P\ xs\ ys\ ]\!] \Longrightarrow P\ (x\#xs)\ (y\#ys)$
shows *P prefix lst*
⟨*proof*⟩

lemma *rsubst*:
$[\![\ P\ s;\ s = t\ ]\!] \Longrightarrow P\ t$
⟨*proof*⟩

lemma *ex-impE*: $((\exists\, x.\ P\ x) \longrightarrow Q) \Longrightarrow P\ x \Longrightarrow Q$
⟨*proof*⟩

lemma *option-Some-value-independent*:
$[\![\ f\ x = Some\ v;\ \bigwedge v'.\ f\ x = Some\ v' \Longrightarrow f\ y = Some\ v'\ ]\!] \Longrightarrow f\ y = Some\ v$
⟨*proof*⟩

Some int bitwise lemmas. Helpers for proofs about `NatBitwise.thy`

lemma *int-2p-eq-shiftl*:
$(2{::}int)\hat{}x = 1 << x$
⟨*proof*⟩

lemma *nat-int-mul*:
$nat\ (int\ a * b) = a * nat\ b$
⟨*proof*⟩

lemma *int-shiftl-less-cancel*:
$n \leq m \Longrightarrow ((x :: int) << n < y << m) = (x < y << (m - n))$
⟨*proof*⟩

lemma *int-shiftl-lt-2p-bits*:
$0 \leq (x{::}int) \Longrightarrow x < 1 << n \Longrightarrow \forall\, i \geq n.\ \neg\ x\ !!\ i$
⟨*proof*⟩

lemmas *int-eq-test-bit = bin-eq-iff*
lemmas *int-eq-test-bitI = int-eq-test-bit*[*THEN iffD2*, *rule-format*]

lemma *le-nat-shrink-left*:
$y \leq z \Longrightarrow y = Suc\ x \Longrightarrow x < z$
⟨*proof*⟩

lemma *length-ge-split*:
$n < length\ xs \Longrightarrow \exists\, x\ xs'.\ xs = x \# xs' \wedge n \leq length\ xs'$
⟨*proof*⟩

Support for defining enumerations on datatypes derived from enumerations

lemma *distinct-map-enum*:
$[\![\ (\forall\ x\ y.\ (F\ x = F\ y \longrightarrow x = y\ ))\ ]\!]$
$\Longrightarrow distinct\ (map\ F\ (enum\text{-}class.enum :: 'a :: enum\ list))$
⟨*proof*⟩

lemma *if-option-None-eq*:
$((if\ P\ then\ None\ else\ Some\ x) = None) = P$
⟨*proof*⟩

lemma *not-in-ran-None-upd*:
$x \notin ran\ m \Longrightarrow x \notin ran\ (m(y := None))$
⟨*proof*⟩

Prevent clarsimp and others from creating Some from not None by folding this and unfolding again when safe.

definition
$not\text{-}None\ x = (x \neq None)$

lemma *sorted-wrt-hd-min*: $[\![\ \bigwedge x.\ P\ x\ x;\ sorted\text{-}wrt\ P\ xs\ ]\!] \Longrightarrow (\forall\, x \in set\ xs.\ P\ (hd\ xs)\ x)$
⟨*proof*⟩

lemma *disjoint-no-subset*: $A \cap B = \{\} \Longrightarrow A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow False$
⟨*proof*⟩

lemma *map-prod-split-case*: $map\text{-}prod\ f\ g\ x = (case\ x\ of\ (a,\ b) => (f\ a,\ g\ b))$
⟨*proof*⟩

lemma *map-prod-split-prj*: $map\text{-}prod\ f\ g\ x = \ (f\ (fst\ x),\ g\ (snd\ x))$
⟨*proof*⟩

end

Chapter 3

ML Antiquotations

3.1 Building terms: *mk-term*

theory *MkTermAntiquote*
imports
Pure
begin

mk-term: ML antiquotation for building and splicing terms.
See `MkTermAntiquote_Tests.thy` for examples and tests.

⟨*ML*⟩

end
mk-term: ML antiquotation for building and splicing terms.

theory *MkTermAntiquote-Tests*
imports
MkTermAntiquote
Main
begin

Basic usage: @{*mk-term pattern*... (*vars*, ...)} (*args*, ...)
The vars should match schematic vars in the pattern, and they are substituted with the given arguments.
Template vars can include type vars, and they should be applied to type arguments.

⟨*ML*⟩
Note that *mk-term* does not perform full type inference automatically. If some argument types are mismatched or too general, the output may be inconsistent. This may change in future versions.

⟨*ML*⟩
Besides static checking, using *mk-term* also gives a reusable template:

⟨*ML*⟩

end

3.2 Term pattern: *term-pat*

theory *TermPatternAntiquote* **imports**
Pure
begin

term-pat: ML antiquotation for pattern matching on terms.
See `TermPatternAntiquote_Tests.thy` for examples and tests.

⟨*ML*⟩

end

theory *TermPatternAntiquote-Tests*
imports
TermPatternAntiquote
Main
begin

Term pattern matching utility.
Instead of writing monstrosities such as

```
case t of
  Const ("Pure.imp", _) $
    P $
    Const (@{const_name Trueprop}, _) $
      (Const ("HOL.eq", _) $
        (Const (@{const_name "my_func"}, _) $ x) $ y)
  => (P, x, y)
```

simply use a term pattern with variables:

```
case t of
  @{term_pat "PROP ?P \<Longrightarrow> my_func ?x = ?y"}
=> (P, x, y)
```

Each *term-pat* generates an ML pattern that can be used in any case-expression or name binding. The ML pattern matches directly on the term datatype; it does not perform matching in the Isabelle logic.

Schematic variables in the pattern generate ML variables. The variables must obey ML's pattern matching rules, i.e. each can appear only once.

Due to the difficulty of enforcing this rule for type variables, schematic type variables are ignored and not checked.

Example: evaluate arithmetic expressions in ML.

⟨*ML*⟩
Regression test: backslash handling

⟨*ML*⟩
Regression test: special-casing for dummy vars

⟨*ML*⟩

end

theory *Print-Annotated*
imports *Main*
keywords *print-annotated-thm* :: *diag*
begin

⟨*ML*⟩

end

theory *ML-Fun-Cache*
imports *Pure*
begin

Extension of the basic cache in `Cache`:

- Include some statistics on cache hits / misses
- Retrieval of existing caches
- Support garbage collected caches (via weak references)

Combination of synchronized variables and weak references. Stored value might get garbage collected. Note the difference in the signature (option value at some positions) compared to `Synchronized`.

⟨*ML*⟩

3.2.1 Example

Cache without garbage collection

⟨*ML*⟩
Cache with garbage collection

⟨*ML*⟩

end

theory *AutoCorres-Utils*
imports
Print-Annotated
ML-Fun-Cache
keywords *lazy-named-theorems*::*thy-decl*

begin

definition *CONV-ID*:: *'a*::{} ⇒ *'a* **where**
CONV-ID x ≡ *x*

lemma *CONV-ID-intro*: (*x*::*'a*::{}) ≡ *CONV-ID x*
⟨*proof*⟩

⟨*ML*⟩

definition *FALSE* ≡ (⋀*P*. *PROP P*)
lemma *ex-falso-quodlibet*: *PROP FALSE* ⟹ *PROP P*
⟨*proof*⟩

⟨*ML*⟩

definition *sim-set* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a set* ⇒ *'b set* ⇒ *bool* **where**
sim-set R A B ⟷ (∀ *a*∈*A*. ∃ *b*∈*B*. *R a b*)

lemma *sim-set-empty*: *sim-set R* {} *B*
⟨*proof*⟩

lemma *sim-set-insert*: *R a b* ⟹ *sim-set R A B* ⟹ *sim-set R* (*insert a A*) (*insert b B*)
⟨*proof*⟩

lemma *sim-set-Collect-Ex*:
(⋀*a*. *sim-set R* {*x*. *P a x*} {*x*. *Q a x*}) ⟹ *sim-set R* {*x*. ∃ *a*. *P a x*} {*x*. ∃ *a*. *Q a x*}
⟨*proof*⟩

lemma *sim-set-Collect-conj*:
(*A* ⟹ *sim-set R* {*x*. *P x*} {*x*. *Q x*}) ⟹ *sim-set R* {*x*. *A* ∧ *P x*} {*x*. *A* ∧ *Q x*}
⟨*proof*⟩

lemma *sim-set-Collect-eq*:
R a b ⟹ *sim-set R* {*x*. *x* = *a*} {*x*. *x* = *b*}
⟨*proof*⟩

lemma *sim-set-eq-rel-set*: *sim-set R* = *rel-set R OO* (⊆)

⟨*proof*⟩

end

3.3 ML Antiquotation to Match and Instantiate (recombine) cterms and terms

theory *Match-Cterm*
imports
AutoCorres-Utils
TermPatternAntiquote
begin

⟨*ML*⟩

The antiquotation yields a function that matches the schematic variables of a pattern with a `cterm`. The matched parts are returned as `cterm` in a record, where the field name is derived from the original schematic variable name. Moreover a function *instantiate* is returned that can be used to instantiate the matched pattern with other `cterm`s.

The antiquotation makes use of the kernel operations `Thm.match` / `Thm.first_order_match`. These operations are efficient in the sense that no costly re-certification of subterms has to be performed.

⟨*ML*⟩

Dummy pattern can also be supplied. The matched values for the dummy patterns are not part of the matching result but still considered in *instantiate*.

⟨*ML*⟩

There is also a variant for first-order-matching.

⟨*ML*⟩

There is also the corresponding antiquotations for plain terms.

⟨*ML*⟩

There you can also see the different workings of first-order-matching. Note the eta expanded match result in the higher-order variant before.

⟨*ML*⟩

There is a variant for ML-pattern-matching. This means, that before using first-order-matching it is tested whether the term actually matches the underlying ML-pattern with ML-pattern matching.

⟨*ML*⟩

Note that as with all ML-Patterns a variable might only appear once. So the following example is a valid first-order or higher-order pattern but not a valid ML-pattern

⟨*ML*⟩
This is what happens conceptually in the expanded antiquotation.

⟨*ML*⟩
With the verbose flag you can see the generated function.

declare [[*verbose=5*]]

⟨*ML*⟩

declare [[*verbose=0*]]

end

3.4 Record Antiquotation

theory *ML-Record-Antiquotation*
imports *Main*
begin

3.4.1 Motivation

A shortcoming of ML records is the lack of proper update / map functions for the record fields. Manually defining those update functions is considered as painful, as it requires 'quadratic' editing when adding a new field: - add the new field to every record pattern (in every already defined update / map function) - add a new update / map function for the field.

Various workarounds have been proposed. E.g. - Using mutable references as fields to allow for destructive updates - Fancy higher order function-combinators (Fold): http://mlton.org/FunctionalRecordUpdate

Here we develop yet another solution. We provide a ML-Antiquotation that generates the definitions.

For a record specification as datatype `datatype 'a foo = Foo {f1:'a, f2:bool}` it generates the datatype as specified with one constructor `Foo` wrapping the record and additionally all the 'canonical' functions:

- `make_foo`
- `dest_foo`
- `get_f1`
- `get_f2`
- `map_f1`
- `map_f2`

3.4.2 Example

⟨*ML*⟩

3.4.3 Implementation

⟨*ML*⟩

Test the parser and its output.

⟨*ML*⟩

3.4.4 Examples

⟨*ML*⟩

Note that the markup of the datatype declaration is limited to basic Lex-markup. Experiments with explicitly evaluating the datatype with e.g. the `ML` function failed. The issue there is that the `ML` function is not evaluated at the point (aka context) exactly in the text position, but with the context of the surrounding ML command.

The following examples try to illustrate the issues.

Works as expected, as every type referred to in the datatype spec is already known at the beginnig of the **ML-val** command.

⟨*ML*⟩

The following fails as 'bar' is not known at the beginning of the context.

⟨*ML*⟩

Adding a semicolon after the definition of bar helps. There the semicolon marks the end of an "evaluation / compilation chunk". So `ML` is evaluated within an already augmented ML-context that knows about *bar*.

⟨*ML*⟩

In the context of a structure the semicolon is not enough.

⟨*ML*⟩

end

theory *Misc-Antiquotation*
imports *Main*
begin

3.5 Various Antiquotations

3.5.1 Antiquotations for terms with schematic variables

⟨*ML*⟩

3.5.2 Antiquotation for types with schematic variables

⟨*ML*⟩

end

Chapter 4

Proof Tools

theory *Tuple-Tools*

imports *Main*
Misc-Antiquotation
TermPatternAntiquote
ML-Fun-Cache

begin

4.1 Tools for handling Tuples

The tools are supposed to help simplifiying expressions containing *case-prod* constructions, like $\lambda(w,\ x,\ y,\ z).\ f\ w\ x\ y\ z$.

- Simprocs for single step splitting and simplification of tuples / *case-prods* instead of repeated application of the built-in variants for pairs
- Looper for the simplifier to instantiate f in $f\ (a,\ b,\ c)$ with $\lambda(a,\ b,\ c).\ g\ a\ b\ c$

4.1.1 Antiquotations for terms with schematic variables

⟨*ML*⟩

Unification only works modulo ordinary beta reduction but does not succeed on tupled-beta reduction. For example the simplifier will not find an instatiation for the variable *?c*

schematic-goal $\bigwedge a\ b.\ (case\ (a,\ b)\ of\ (x,\ y) \Rightarrow f\ x\ y) \equiv ?c\ (a,\ b)$
⟨*proof*⟩

Such equations can typically occur in congruence rules when a pair is splitted by $(\bigwedge x.\ PROP\ ?P\ x) \equiv (\bigwedge a\ b.\ PROP\ ?P\ (a,\ b))$.

lemma *tuple-up-eq-trivial*: $f\ r\ \equiv (\lambda x.\ f\ x)\ r$

⟨*proof*⟩

thm *tuple-up-eq-trivial*
lemma *tuple-up-eq2*: *f* (*fst r*) (*snd r*) ≡ (*case r of* (*x1*, *x2*) ⇒ *f x1 x2*)
⟨*proof*⟩

Constant to explictly trigger splitting by simproc `SPLIT_simproc` below

definition *SPLIT* :: *'a*::{} ⇒ *'a*
where *SPLIT P* ≡ *P*

lemma *SPLIT-cong*: *PROP SPLIT P* ≡ *PROP SPLIT P*
⟨*proof*⟩

lemma *case-prod-out*: (λ*r*. *f* ((λ(*a*,*b*). (*g a b*)) *r*)) = (λ(*a*, *b*). *f* (*g a b*))
⟨*proof*⟩

lemma *case-prod-eta-contract*: (λ*x*. (*case-prod s*) *x*) ≡ (*case-prod s*)
⟨*proof*⟩

definition *ETA-TUPLED* :: (*'a* ⇒ *'b*) ⇒ (*'a* ⇒ *'b*)
where *ETA-TUPLED f* ≡ *f*

lemma *ETA-TUPLED-trans*: *f* ≡ *g* ⟹ *ETA-TUPLED f* ≡ *g*
⟨*proof*⟩

Add the congruence rule *SPLIT PROP ?P* ≡ *SPLIT PROP ?P* to the simpset together with the simproc to avoid descending into *P* before the split.

⟨*ML*⟩
declare [[*simproc del*: *ETA-TUPLED*]]

Set up the theorem cache. We could use "dynamic programming" here and rewrite "thm n" with one step of "thm (n - 1)" and *case-prod* = (λ*c p*. *c* (*fst p*) (*snd p*))

⟨*ML*⟩

thm *HOL.ext* [*split-tuple f* **and** *g arity*: *3*]

⟨*ML*⟩

Especially in the case of congruence rules we prefer the more restrictive instantiation, where the freshly introduced variable does not depend on 'non-bound' tuple components

⟨*ML*⟩

declare [[*simp-trace*=*false*]]

Variable names should be preserved here, since all abstractions stay in place.

lemma $\bigwedge r.\ (\lambda(x,y).\ (x::nat) < y \wedge y < 200)\ r$
⟨*proof*⟩

Variable names are not preserved due to eta contraction: $(\lambda(x,\ y).\ x < y) = (\lambda(x,\ y).\ x < y)$

lemma $\bigwedge r.\ (\lambda(x,y).\ (x::nat) < y\ )\ r$
⟨*proof*⟩

lemma $(\lambda(x,y,p,z,f).\ (x::nat) < y)\ (a,b,c) = (case\ c\ of\ (p,z,f) \Rightarrow a < b)$
⟨*proof*⟩

lemma $\bigwedge r.\ (\lambda(x,y,p,z,f).\ (p::nat) < f)\ r$
⟨*proof*⟩

lemma $(\lambda(x,y,p).\ (x::nat) < y)\ (a,b,c,d,e) = (a < b)$
⟨*proof*⟩

⟨*ML*⟩

lemma *SPLIT* $(\bigwedge r.\ ((\lambda z.\ Q\ z \wedge (Q\ z \longrightarrow P\ z))\ r))$
$\equiv$ *XXX*
⟨*proof*⟩

lemma *PROP SPLIT* $(\bigwedge r.\ ((\lambda(x,y,z).\ y < z \wedge z=s)\ r) \Longrightarrow P\ r)$
$\equiv (\bigwedge x\ y\ z.\ y < z \wedge z = s \Longrightarrow P\ (x,\ y,\ s))$
⟨*proof*⟩

schematic-goal *ETA-TUPLED* $(\lambda x::('a \times 'b \times 'c).\ f\ x) = ?XXX$
⟨*proof*⟩

context
fixes $f::('a \times 'b \times 'c \times 'd) \Rightarrow nat$
fixes $c::('a \times 'b \times 'c \times 'd) \Rightarrow 's \Rightarrow bool$
begin

⟨*ML*⟩
end

end

theory *Subgoal-Methods*
imports *Main*
begin

⟨*ML*⟩

end

4.2 Tools for intro-rule based term synthesis *synthesize-rules*

theory *Synthesize*
imports
Tuple-Tools
AutoCorres-Utils
MkTermAntiquote
keywords
synthesize-rules::*thy-decl* **and**
add-synthesize-pattern::*thy-decl* % *ML* **and**
print-synthesize-rules::*diag*
begin

⟨*ML*⟩

4.2.1 Commands

⟨*ML*⟩

4.2.2 ML Antiquotations

⟨*ML*⟩

4.2.3 Attributes

⟨*ML*⟩

end

Chapter 5

Rule by Method

theory *Rule-By-Method*
imports
  Main
  HOL−Eisbach.Eisbach-Tools
begin

⟨*ML*⟩

experiment begin

⟨*ML*⟩

lemmas *baz* = [[@‹*erule thin-rl*, *rule revcut-rl*[*of P* ⟶ *P* ∧ *P*], *simp*›]] **for** *P*

lemmas *bazz*[*THEN impE*] = *TrueI*[@‹*erule thin-rl*, *rule revcut-rl*[*of P* ⟶ *P* ∧ *P*], *simp*›] **for** *P*

lemma *Q* ⟶ *Q* ∧ *Q* ⟨*proof*⟩

method *silly-rule* **for** *P* :: *bool* **uses** *rule* =
  (*rule* [[@‹*erule thin-rl*, *cut-tac rule*, *drule asm-rl*[*of P*]›]])

lemma assumes *A* **shows** *A* ⟨*proof*⟩

lemma assumes *A*[*simp*]: *A* **shows** *A*
  ⟨*proof*⟩

end

end

theory *Option-Scanner*
imports *ML-Record-Antiquotation*

begin

The purpose of the option scanner is to provide an interface to scan lists of options (key / value) for toplevel commands. As values may have different types the idea is that the collection of options is represented by an record of optional values. This record can be provided individually for each command. The initial option-record "empty" has every component assigned to `NONE`. To be able to specify all options in a single list, we represent a single parser / scanner for a field as an update function for the option-record, which is composed by the map-functions for the record fields. So we can preserve different types for parsing individual fields (and do not need an universal value type), while the specification list is a monotype.

⟨*ML*⟩

end

theory *Named-Rules*
imports *Main*
keywords *named-rules*::*thy-decl*
begin
⟨*ML*⟩
end

theory *Subgoals*
imports *Main*
keywords *prefers* :: *prf-script* % *proof* **and** *subgoals* :: *prf-script-goal* % *proof*
begin

definition *protected-conjunction* :: *prop* ⇒ *prop* ⇒ *prop* (**infixr** ‹&^&› *2*) **where**
protected-conjunction A B ≡ (*PROP A* &&& *PROP B*)

definition
protected-prop A ≡ *PROP A*

lemma *protect-prop*:
PROP A **if** *PROP protected-prop A*
⟨*proof*⟩

⟨*ML*⟩

Usage examples **lemma**
x > *2* ⟹ *x* > *0* ∧ *x* > *1* **for** *x* :: *nat*
⟨*proof*⟩

lemma
x = *2* ⟹ *x* > *0* ∧ *x* ≤ *3* ∧ *x* ≤ *2* **for** *x* :: *nat*

⟨*proof*⟩

end

5.1 Tagging

theory *Tagging*
imports *Main Subgoals HOL−Eisbach.Eisbach*
keywords *preferT prefersT* :: *prf-script* % *proof*
and *subgoalT subgoalsT* :: *prf-script-goal* % *proof*
begin

5.1.1 Basic Definitions and Theorems

definition *ASM-TAG* (‹¶›) **where**
ASM-TAG t ≡ *True*

definition *TAG* :: *'b* ⇒ *'a* :: {} ⇒ *'a* (‹(‹*open-block notation*=‹*infix TAG*››- | -)› [*13*, *13*] *14*)
where ‹*TAG t x* ≡ *x*›

abbreviation *tag-prop* (‹(‹*open-block notation*=‹*infix TAG*››- || -)› [*0*, *0*] *0*)
where *tag-prop t x* ≡ *PROP TAG t x*

lemma *TAG-cong*[*cong*]: *x* ≡ *y* ⟹ (*tag* | *x*) ≡ *tag* | (*y*::*'a*::{})
⟨*proof*⟩

lemma *ASM-TAG-cong*[*cong*]: ¶ *tag* ⟷ ¶ *tag*
⟨*proof*⟩

lemma *ASM-TAG-I*[*intro*!]: ¶ *x* ⟨*proof*⟩

lemma *TAG-TrueI*[*intro*!, *simp*]: *tag* | *True*
⟨*proof*⟩

lemma *TAG-False*[*simp*]: (*tag* | *False*) ⟷ *False*
⟨*proof*⟩

lemma *TAG-false*[*elim*!]: (*tag* | *False*) ⟹ *P*
⟨*proof*⟩

lemma *ASM-TAG-aux1*:
PROP P ≡ (*True* ⟹ *PROP P*)
⟨*proof*⟩

lemma *ASM-TAG-CONV1*:
PROP TAG t P ≡ (*ASM-TAG t* ⟹ *PROP P*)
⟨*proof*⟩

lemma *disjE-tagged*:
$(P \vee Q) \Longrightarrow (\P\ ''l'' \Longrightarrow P \Longrightarrow R) \Longrightarrow (\P\ ''r'' \Longrightarrow Q \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

lemma *conjI-tagged*:
$(\P\ ''l'' \Longrightarrow P) \Longrightarrow (\P\ ''r'' \Longrightarrow Q) \Longrightarrow P \wedge Q$
⟨*proof*⟩
lemma *assm-tagE*[*elim*!]:
assumes $\P\ t \mid A$
assumes $\P\ t \Longrightarrow t \mid A \Longrightarrow P$
shows P
⟨*proof*⟩

⟨*ML*⟩

Subgoals Test

lemma
assumes *False*
shows
$\bigwedge a\ b.\ A \Longrightarrow B \Longrightarrow C \Longrightarrow ''foo'' \mid AA\ a\ b$
$A \Longrightarrow B \Longrightarrow C2 \Longrightarrow ''bar'' \mid B$
$A \Longrightarrow B \Longrightarrow C \Longrightarrow ''bar'' \mid C$
$A2 \Longrightarrow B \Longrightarrow C22 \Longrightarrow ''bar'' \mid C$
$\bigwedge b.\ A2 \Longrightarrow B \Longrightarrow C22 \Longrightarrow ''foo'' \mid CC\ b$
$\bigwedge a\ b.\ A2 \Longrightarrow B \Longrightarrow C22 \Longrightarrow f\ a\ b$
$\bigwedge c\ d.\ A2 \Longrightarrow \P\ ''foobar'' \Longrightarrow B \Longrightarrow C22 \Longrightarrow D\ c\ d$
⟨*proof*⟩

5.1.2 Conversions

⟨*ML*⟩

lemma *norm-tag*:
Trueprop $(t \mid P) \equiv (t \mid\mid P)$
⟨*proof*⟩

⟨*ML*⟩

method *tidy-tags* = *changed* ‹*tidy-tags-tac*›

method *tidy-tags-assm* = *changed* ‹*tidy-tags-assm-tac*›

lemma
$n > 0$ **if** $n > 1 \vee n > 2$ **for** $n :: nat$
⟨*proof*⟩

lemma
$n > 0$ **if** $(\P\ ''l'' \mid n > 1) \vee (\P\ ''r'' \mid n > 2)$ **for** $n :: nat$
⟨*proof*⟩

lemma *rm-ASM-TAG*:
(*ASM-TAG A* $\Longrightarrow$ *PROP B*) $\equiv$ *PROP B*
⟨*proof*⟩

⟨*ML*⟩

thm *conjI*[*tags l r*] *disjE*[*tags - l r*] *conjI*[*tags l r*, *untag*] *disjE*[*tags - l r*, *untag*]

thm *conjI*[*case-names A B*, *tags*] *disjE*[*case-names l* [*P*] *r* [*Q*], *consumes 1*, *tags*] *nat-induct*[*tags*]
conjI[*case-names A B*, *tags*, *untag*] *disjE*[*case-names l* [*P*] *r* [*Q*], *consumes 1*, *tags*, *untag*]
nat-induct[*tags*, *untag*]

— the *assm-names* slot is meant to retain assumption names from theorem statements:
thm *conjI*[*tags*] *disjE*[*tagged assm-names* ;*l*;*r*, *tags*]

thm *conjI*[*tag*] *conjI*[*tag conj*]

5.1.3 Globbing

⟨*ML*⟩

5.1.4 Reordering Subgoals

⟨*ML*⟩

lemma
TAG ″*p*″ *P TAG* ″*q*″ *Q PROP TAG* ″*r*″ *R TAG* ″*p*″ *P2*
⟨*proof*⟩

lemma
$\bigwedge t$. *TAG* ″*p*″ *P* $\wedge$ *TAG* ″*q*″ *Q*
⟨*proof*⟩

lemma
P $\wedge$ *Q*
⟨*proof*⟩

5.1.5 *subgoalT* and *subgoalsT*, and *prefersT*

⟨*ML*⟩

lemma
¶ ″*tag*″ $\Longrightarrow$ *TAG* ″*p*″ *P* $\Longrightarrow$ *TAG* ″*q*″ *Q* $\Longrightarrow$ *P* $\wedge$ *Q*
⟨*proof*⟩

lemma

assumes *TAG ''p'' P Q False*
shows *TAG [''x''] True TAG [''y''] False TAG [''x'', ''y''] False TAG [''a'', ''b''] True*
⟨*proof*⟩

lemma
assumes *TAG ''p'' P Q False*
shows *TAG [''x''] True TAG [''y''] False TAG [''x'', ''y''] False TAG [''a'', ''b''] True*
⟨*proof*⟩

lemma
assumes *TAG ''p'' P TAG [''q'', ''r''] Q TAG [''q''] QQ False*
shows *TAG [''x''] True TAG [''y''] False TAG [''x'', ''y''] False TAG [''a'', ''b''] True*
⟨*proof*⟩

lemma
assumes *TAG ''p'' P TAG [''q'', ''r''] Q False*
shows
TAG [''x''] True TAG [''y''] False P TAG [''x'', ''y''] False TAG [''a'', ''b''] True
⟨*proof*⟩

lemma
n > 0 **if** (¶ *''l''* | *n > 1*) ∨ (¶ *''r''* | *n > 2*) **for** *n* :: *nat*
⟨*proof*⟩

lemma
n > 0 **if** (¶ *''l''* | *n > 9*) ∨ (¶ *''s''* | *n > 2*) ∨ (¶ *''s''* | *n > 0*) **for** *n* :: *nat*
⟨*proof*⟩

5.1.6 Syntax

⟨*ML*⟩

syntax *-ASM-TAG* :: ‹*cartouche-position* ⇒ *string*› (¶-)

⟨*ML*⟩

syntax *-TAG* :: ‹*cartouche-position* ⇒ *'a* ⇒ *string*› (- | - [*13*, *13*] *14*)

⟨*ML*⟩

syntax *-TAG* :: ‹*cartouche-position* ⇒ *'a* ⇒ *string*› (- | - [*13*, *13*] *14*)

⟨*ML*⟩

term ‹‹› | *a*›

term ‹‹ab› | a›
term ‹‹a-b',c› | abc›
term ¶‹AB› | $a = b$
term ‹$[''a'', ''b'']$ | t›
term ‹$(‹a›$ | $b)$ x›
term ‹$(t$ | $b)$ x›
term ‹t | x›
term ‹¶ t›

end

Chapter 6

Verification Condition Generator *runs-to-vcg*

theory *Basic-Runs-To-VCG*
imports
Named-Rules
HOL−Eisbach.Eisbach-Tools
Tagging
begin

6.1 Marked Assumptions

⟨*ML*⟩

named-theorems *remove-ASSMs Remove the markers for marked assumptions*

method *cleanup-ASSMs* = *simp-all only*: *remove-ASSMs*

definition *SIMP-ASSM* :: *bool* ⇒ *bool* **where** [*remove-ASSMs*]: *SIMP-ASSM P* ⟷ *P*
lemma *SIMP-ASSM-D*: *SIMP-ASSM P* ⟹ *P* ⟨*proof*⟩
⟨*ML*⟩

6.2 *THEN-ALL-NEW-FORWARD*

⟨*ML*⟩

6.3 Basic VCG

named-theorems *runs-to-vcg-cong-state-only*
named-theorems *runs-to-vcg-weaken*
named-theorems *runs-to-vcg-cong-program-only*
named-rules (*intro*) *runs-to-vcg*

named-theorems *runs-to-vcg-post-elims*
named-theorems *runs-to-vcg-post-intros*

declare *conjI*[*runs-to-vcg-post-intros*]
declare *disjE*[*runs-to-vcg-post-elims*]

⟨*ML*⟩

6.4 Setup for Tagging

lemma *push-tag-to-assm*:
  $t \mid P$ **if** ¶ $t \Longrightarrow P$
  ⟨*proof*⟩

lemma *tagE*:
  $tag \mid Q \Longrightarrow (Q \Longrightarrow P) \Longrightarrow P$
  ⟨*proof*⟩

bundle *basic-vcg-tagging-setup*
begin

lemmas [*runs-to-vcg-post-intros*] = *push-tag-to-assm TAG-TrueI ASM-TAG-I*
lemmas [*runs-to-vcg-post-elims*] = *tagE*

end

end

theory *Runs-To-VCG*
  imports
    Basic-Runs-To-VCG
begin

lemma *transfer-conj-imp*: *rel-fun* (=) (*rel-fun* (⟶) (⟶)) (∧) (∧)
  ⟨*proof*⟩

lemma *transfer-all-imp*: *rel-fun* (*rel-set R*) (*rel-fun* (*rel-fun R* (⟶)) (⟶)) (*Ball*)
(*Ball*)
  ⟨*proof*⟩

6.5 *runs-to-vcg*

⟨*ML*⟩

6.6 Check

thm *runs-to-vcg* [*no-vars*]
end

Chapter 7

Proof Methods

theory *Eisbach-Methods*
imports
Subgoal-Methods
HOL−Eisbach.Eisbach-Tools
Rule-By-Method
begin

7.1 Debugging methods

method *print-concl* = (*match* **conclusion in** *P* **for** *P* ⇒ ‹*print-term P*›)

⟨*ML*⟩

7.2 Simple Combinators

⟨*ML*⟩

method *repeat-new* **methods** *m* = (*m* ; (*repeat-new* ‹*m*›)*?*)

The following *fails* and *succeeds* methods protect the goal from the effect of a method, instead simply determining whether or not it can be applied to the current goal. The *fails* method inverts success, only succeeding if the given method would fail.

⟨*ML*⟩

This method wraps up the "focus" mechanic of match without actually doing any matching. We need to consider whether or not there are any assumptions in the goal, as premise matching fails if there are none.

If the *fails* method is removed here, then backtracking will produce a set of invalid results, where only the conclusion is focused despite the presence of subgoal premises.

method *focus-concl* **methods** *m* =
((*fails* ‹*erule thin-rl*›, *match* **conclusion in** - ⇒ ‹*m*›)

| *match* **premises** (*local*) **in** *H*:- (*multi*) ⇒ ‹*m*›)

repeat applies a method a specific number of times, like a bounded version of the '+' combinator.

usage: apply (repeat n *text*)

- Applies the method *text* to the current proof state n times. - Fails if *text* can't be applied n times.

⟨*ML*⟩

notepad begin
⟨*proof*⟩

end

Literally a copy of the parser for *subgoal-tac* composed with an analogue of **prefer**.

Useful if you find yourself introducing many new facts via *subgoal-tac*, but prefer to prove them immediately rather than after they're used.

⟨*ML*⟩

notepad begin ⟨*proof*⟩
end

7.3 Advanced combinators

7.3.1 Protecting goal elements (assumptions or conclusion) from methods

context
begin

private definition *protect-concl* $x \equiv \neg\ x$
private definition *protect-false* ≡ *False*

private lemma *protect-start*: (*protect-concl P* ⟹ *protect-false*) ⟹ *P*
⟨*proof*⟩ **lemma** *protect-end*: *protect-concl P* ⟹ *P* ⟹ *protect-false*
⟨*proof*⟩

method *only-asm* **methods** *m* =
(*match* **premises in** *H*[*thin*]:- (*multi*,*cut*) ⇒
‹*rule protect-start*,
match premises in H'[*thin*]:*protect-concl* - ⇒
‹*insert H*,*m*;*rule protect-end*[*OF H'*]››)

method *only-concl* **methods** *m* = (*focus-concl* ‹*m*›)

end

notepad begin
⟨*proof*⟩

end

7.3.2 Safe subgoal folding (avoids expanding meta-conjuncts)

Isabelle's goal mechanism wants to aggressively expand meta-conjunctions if they are the top-level connective. This means that *fold-subgoals* will immediately be unfolded if there are no common assumptions to lift over.

To avoid this we simply wrap conjunction inside of conjunction' to hide it from the usual facilities.

context begin

definition
conjunction′ :: *prop* ⇒ *prop* ⇒ *prop* (**infixr** ‹&^&› *2*) **where**
conjunction′ A B ≡ (*PROP A* &&& *PROP B*)

In general the context antiquotation does not work in method definitions. Here it is fine because `Conv.top_sweep_conv` is just over-specified to need a Proof.context when anything would do.

method *safe-meta-conjuncts* =
raw-tactic
‹*REPEAT-DETERM*
(*CHANGED-PROP*
(*PRIMITIVE*
(*Conv.gconv-rule* ((*Conv.top-sweep-conv* (*K* (*Conv.rewr-conv* @{*thm conjunction′-def*[*symmetric*]})) @{*context*})) *1*)))›

method *safe-fold-subgoals* = (*fold-subgoals* (*prefix*), *safe-meta-conjuncts*)

lemma *atomize-conj′* [*atomize*]: (*A* &^& *B*) == *Trueprop* (*A* & *B*)
⟨*proof*⟩

lemma *context-conjunction′I*:
PROP P ⟹ (*PROP P* ⟹ *PROP Q*) ⟹ *PROP P* &^& *PROP Q*
⟨*proof*⟩

lemma *conjunction′I*:
PROP P ⟹ *PROP Q* ⟹ *PROP P* &^& *PROP Q*
⟨*proof*⟩

lemma *conjunction′E*:
assumes *PQ*: *PROP P* &^& *PROP Q*
assumes *PQR*: *PROP P* ⟹ *PROP Q* ⟹ *PROP R*
shows
PROP R
⟨*proof*⟩

end

notepad begin
⟨*proof*⟩

end

7.4 Utility methods

7.4.1 Finding a goal based on successful application of a method

⟨*ML*⟩

Ensure that the proof state is in a certain case of a case distinction:

method *in-case* **for** x = *match* **premises in** $t = x$ **for** $t \Rightarrow$ *succeed*

Focus on a case in a case distinction:

method *find-case* **for** x = *find-goal* ‹*in-case* x›

notepad begin

⟨*proof*⟩

end

7.4.2 Remove redundant subgoals

Tries to solve subgoals by assuming the others and then using the given method. Backtracks over all possible re-orderings of the subgoals.

context begin

definition *protect* (*PROP P*) ≡ *P*

lemma *protectE*: *PROP protect P* ⟹ (*PROP P* ⟹ *PROP R*) ⟹ *PROP R*
⟨*proof*⟩ **lemmas** *protect-thin* = *thin-rl*[**where** *V*=*PROP protect P* **for** *P*]

private lemma *context-conjunction′I-protected*:
assumes *P*: *PROP P*
assumes *PQ*: *PROP protect* (*PROP P*) ⟹ *PROP Q*
shows
PROP P &^& *PROP Q*
⟨*proof*⟩ **lemma** *conjunction′-sym*: *PROP P* &^& *PROP Q* ⟹ *PROP Q* &^& *PROP P*
⟨*proof*⟩ **lemmas** *context-conjuncts′I* =
context-conjunction′I-protected
context-conjunction′I-protected[*THEN conjunction′-sym*]

method *distinct-subgoals-strong* **methods** m =
(*safe-fold-subgoals*,
(*intro context-conjuncts'I*;
(((*elim protectE conjunction'E*)*?*, *solves* ‹*m*›)
| (*elim protect-thin*)*?*)))*?*

end

method *forward-solve* **methods** *fwd* m =
(*fwd*, *prefer-last*, *fold-subgoals*, *safe-meta-conjuncts*, *rule conjunction'I*,
defer-tac, ((*intro conjunction'I*)*?*; *solves* ‹*m*›))[*1*]

method *frule-solve* **methods** m **uses** *rule* = (*forward-solve* ‹*frule rule*› ‹*m*›)
method *drule-solve* **methods** m **uses** *rule* = (*forward-solve* ‹*drule rule*› ‹*m*›)

notepad begin
⟨*proof*⟩
end

notepad begin
⟨*proof*⟩
end

7.5 Attribute methods (for use with *rule-by-method* attributes)

method *prove-prop-raw* **for** P :: *prop* **methods** m =
(*erule thin-rl*, *rule revcut-rl*[*of PROP P*],
solves ‹*match conclusion in* - ⇒ ‹*m*››)

method *prove-prop* **for** P :: *prop* = (*prove-prop-raw PROP P* ‹*auto*›)

experiment begin

lemma assumes A[*simp*]:A **shows** A ⟨*proof*⟩

end

7.6 Shortcuts for *prove-prop*.

Note these are less efficient than using the raw syntax because the facts are re-proven every time.

method *ruleP* **for** P :: *prop* = (*catch* ‹*rule* [[@‹*prove-prop PROP P*›]]› ‹*fail*›)
method *insertP* **for** P :: *prop* = (*catch* ‹*insert* [[@‹*prove-prop PROP P*›]]› ‹*fail*›)[*1*]

experiment begin

lemma assumes *A*[*simp*]:*A* **shows** *A* ⟨*proof*⟩
lemma assumes *A*:*A* **shows** *A* ⟨*proof*⟩

end

context begin

private definition *bool-protect* (*b*::*bool*) ≡ *b*

lemma *bool-protectD*:
bool-protect P ⟹ *P*
⟨*proof*⟩

lemma *bool-protectI*:
P ⟹ *bool-protect P*
⟨*proof*⟩

When you want to apply a rule/tactic to transform a potentially complex goal into another one manually, but want to indicate that any fresh emerging goals are solved by a more brutal method. E.g. *apply* (*solves-emerging frule x=... in my-rulefastforce simp*: ... *intro*!: ...

method *solves-emerging* **methods** *m1 m2* = (*rule bool-protectD*, (*m1* ; (*rule bool-protectI* | (*m2*; *fail*))))

end

end

theory *Less-Monad-Syntax*
imports *HOL−Library.Monad-Syntax*
begin

no-syntax
-thenM :: ['*a*, '*b*] ⇒ '*c* (**infixr** ‹>>› *54*)

no-notation
Monad-Syntax.bind (**infixr** ‹>>=› *54*)

notation (**output**)
bind-do (**infixl** ‹>>=› *54*)

translations

CONST bind-do <= CONST bind

end

Chapter 8

Option Monad (State Reader)

theory *Reader-Monad*
imports
More-Lib
Less-Monad-Syntax
begin

type-synonym $('s,'a)$ *lookup* $= 's \Rightarrow 'a$ *option*

Similar to *map-option* but the second function returns option as well

definition
opt-map :: $('s,'a)$ *lookup* $\Rightarrow$ $('a \Rightarrow 'b$ *option*$) \Rightarrow ('s,'b)$ *lookup* (**infixl** ‹|>› *54*)
where
f |> $g \equiv \lambda s.$ *case* $f\ s$ *of None* $\Rightarrow$ *None* | *Some* $x \Rightarrow g\ x$

abbreviation *opt-map-Some* :: $('s \rightharpoonup 'a) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 's \rightharpoonup 'b$ (**infixl** ‹||>› *54*)
where
f ||> $g \equiv f$ |> (*Some* $\circ$ g)
lemmas *opt-map-Some-def* = *opt-map-def*

lemma *opt-map-cong* [*fundef-cong*]:
$[\![f = f'; \bigwedge v\ s.\ f\ s = \textit{Some}\ v \Longrightarrow g\ v = g'\ v]\!] \Longrightarrow f$ |> $g = f'$ |> g'
⟨*proof*⟩

lemma *in-opt-map-eq*:
$((f$ |> $g)\ s = \textit{Some}\ v) = (\exists v'.\ f\ s = \textit{Some}\ v' \wedge g\ v' = \textit{Some}\ v)$
⟨*proof*⟩

lemma *opt-mapE*:
$[\![(f$ |> $g)\ s = \textit{Some}\ v; \bigwedge v'.\ [\![f\ s = \textit{Some}\ v'; g\ v' = \textit{Some}\ v\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *opt-map-upd-None*:

$f(x := None) \mid> g = (f \mid> g)(x := None)$
⟨*proof*⟩

lemma *opt-map-upd-Some*:
$f(x \mapsto v) \mid> g = (f \mid> g)(x := g\ v)$
⟨*proof*⟩

lemmas *opt-map-upd*[*simp*] = *opt-map-upd-None opt-map-upd-Some*

declare *None-upd-eq*[*simp*]

lemma ⟦ $(f \mid> g)\ x = None$; $g\ v = None$ ⟧ ⟹ $f(x \mapsto v) \mid> g = f \mid> g$
⟨*proof*⟩

definition
obind :: $('s,'a)$ *lookup* ⇒ $('a \Rightarrow ('s,'b)$ *lookup*$)$ ⇒ $('s,'b)$ *lookup* (**infixl** ‹|>>› *53*)
where
$f \mid>> g \equiv \lambda s.$ *case* $f\ s$ *of None* ⇒ *None* | *Some* x ⇒ $g\ x\ s$

adhoc-overloading
Monad-Syntax.bind ⇌ *obind*

definition
ofail = *K None*

definition
oreturn = *K o Some*

definition
oassert $P \equiv$ *if* P *then oreturn* () *else ofail*

definition *oapply* :: $'a \Rightarrow ('a \Rightarrow 'b$ *option*$) \Rightarrow 'b$ *option*
where
oapply $x \equiv \lambda s.\ s\ x$

If the result can be an exception. Corresponding bindE would be analogous to lifting in NonDetMonad.

definition
oreturnOk x = *K* (*Some* (*Inr* x))

definition
othrow e = *K* (*Some* (*Inl* e))

definition
oguard $G \equiv (\lambda s.$ *if* $G\ s$ *then Some* () *else None*)

definition

ocondition c L R ≡ (λ*s*. *if c s then L s else R s*)

definition
oskip ≡ *oreturn* ()

Monad laws

lemma *oreturn-bind* [*simp*]: (*oreturn x* |>> *f*) = *f x*
⟨*proof*⟩

lemma *obind-return* [*simp*]: (*m* |>> *oreturn*) = *m*
⟨*proof*⟩

lemma *obind-assoc*:
(*m* |>> *f*) |>> *g* = *m* |>> (λ*x*. *f x* |>> *g*)
⟨*proof*⟩

Binding fail

lemma *obind-fail* [*simp*]:
f |>> (λ-. *ofail*) = *ofail*
⟨*proof*⟩

lemma *ofail-bind* [*simp*]:
ofail |>> *m* = *ofail*
⟨*proof*⟩

Function package setup

lemma *opt-bind-cong* [*fundef-cong*]:
⟦ *f* = *f'*; ⋀*v s*. *f' s* = *Some v* ⟹ *g v s* = *g' v s* ⟧ ⟹ *f* |>> *g* = *f'* |>> *g'*
⟨*proof*⟩

lemma *opt-bind-cong-apply* [*fundef-cong*]:
⟦ *f s* = *f' s*; ⋀*v*. *f' s* = *Some v* ⟹ *g v s* = *g' v s* ⟧ ⟹ (*f* |>> *g*) *s* = (*f'* |>> *g'*) *s*
⟨*proof*⟩

lemma *oassert-bind-cong* [*fundef-cong*]:
⟦ *P* = *P'*; *P'* ⟹ *m* = *m'* ⟧ ⟹ *oassert P* |>> *m* = *oassert P'* |>> *m'*
⟨*proof*⟩

lemma *oassert-bind-cong-apply* [*fundef-cong*]:
⟦ *P* = *P'*; *P'* ⟹ *m* () *s* = *m'* () *s* ⟧ ⟹ (*oassert P* |>> *m*) *s* = (*oassert P'* |>> *m'*) *s*
⟨*proof*⟩

lemma *oreturn-bind-cong* [*fundef-cong*]:
⟦ *x* = *x'*; *m x'* = *m' x'* ⟧ ⟹ *oreturn x* |>> *m* = *oreturn x'* |>> *m'*
⟨*proof*⟩

lemma *oreturn-bind-cong-apply* [*fundef-cong*]:

$[\![x = x'; m\ x' \ s = m'\ x'\ s]\!] \Longrightarrow (oreturn\ x \mid>>\ m)\ s = (oreturn\ x' \mid>>\ m')\ s$
⟨*proof*⟩

lemma *oreturn-bind-cong2* [*fundef-cong*]:
$[\![x = x'; m\ x' = m'\ x']\!] \Longrightarrow (oreturn\ \$\ x) \mid>>\ m = (oreturn\ \$\ x') \mid>>\ m'$
⟨*proof*⟩

lemma *oreturn-bind-cong2-apply* [*fundef-cong*]:
$[\![x = x'; m\ x' \ s = m'\ x'\ s]\!] \Longrightarrow ((oreturn\ \$\ x) \mid>>\ m)\ s = ((oreturn\ \$\ x') \mid>>\ m')\ s$
⟨*proof*⟩

lemma *ocondition-cong* [*fundef-cong*]:
$[\![c = c'; \bigwedge s.\ c'\ s \Longrightarrow l\ s = l'\ s; \bigwedge s.\ \neg c'\ s \Longrightarrow r\ s = r'\ s]\!]$
$\Longrightarrow ocondition\ c\ l\ r = ocondition\ c'\ l'\ r'$
⟨*proof*⟩

Decomposition

lemma *ocondition-K-true* [*simp*]:
$ocondition\ (\lambda\text{-}.\ True)\ T\ F = T$
⟨*proof*⟩

lemma *ocondition-K-false* [*simp*]:
$ocondition\ (\lambda\text{-}.\ False)\ T\ F = F$
⟨*proof*⟩

lemma *ocondition-False*:
$[\![\bigwedge s.\ \neg\ P\ s]\!] \Longrightarrow ocondition\ P\ L\ R = R$
⟨*proof*⟩

lemma *ocondition-True*:
$[\![\bigwedge s.\ P\ s]\!] \Longrightarrow ocondition\ P\ L\ R = L$
⟨*proof*⟩

lemma *in-oreturn* [*simp*]:
$(oreturn\ x\ s = Some\ v) = (v = x)$
⟨*proof*⟩

lemma *oreturnE*:
$[\![oreturn\ x\ s = Some\ v;\ v = x \Longrightarrow P]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *in-ofail* [*simp*]:
$ofail\ s \neq Some\ v$
⟨*proof*⟩

lemma *ofailE*:
$ofail\ s = Some\ v \Longrightarrow P$
⟨*proof*⟩

lemma *in-oassert-eq* [*simp*]:
(*oassert P s = Some v*) = *P*
⟨*proof*⟩

lemma *oassert-True* [*simp*]:
oassert True = oreturn ()
⟨*proof*⟩

lemma *oassert-False* [*simp*]:
oassert False = ofail
⟨*proof*⟩

lemma *oassertE*:
⟦ *oassert P s = Some v*; *P* ⟹ *Q* ⟧ ⟹ *Q*
⟨*proof*⟩

lemma *in-obind-eq*:
((*f* |>> *g*) *s = Some v*) = (∃ *v'*. *f s = Some v'* ∧ *g v' s = Some v*)
⟨*proof*⟩

lemma *obind-eqI*:
⟦ *f s = f s'* ; ⋀*x*. *f s = Some x* ⟹ *g x s = g' x s'* ⟧ ⟹ *obind f g s = obind f g' s'*
⟨*proof*⟩

lemma *obind-eqI-full*:
⟦ *f s = f s'* ; ⋀*x*. ⟦ *f s = Some x*; *f s' = f s* ⟧ ⟹ *g x s = g' x s'* ⟧
⟹ *obind f g s = obind f g' s'*
⟨*proof*⟩

lemma *obindE*:
⟦ (*f* |>> *g*) *s = Some v*;
⋀*v'*. ⟦*f s = Some v'*; *g v' s = Some v*⟧ ⟹ *P*⟧ ⟹ *P*
⟨*proof*⟩

lemma *in-othrow-eq* [*simp*]:
(*othrow e s = Some v*) = (*v = Inl e*)
⟨*proof*⟩

lemma *othrowE*:
⟦*othrow e s = Some v*; *v = Inl e* ⟹ *P*⟧ ⟹ *P*
⟨*proof*⟩

lemma *in-oreturnOk-eq* [*simp*]:
(*oreturnOk x s = Some v*) = (*v = Inr x*)
⟨*proof*⟩

lemma *oreturnOkE*:
⟦*oreturnOk x s* = *Some v*; *v* = *Inr x* ⟹ *P*⟧ ⟹ *P*
⟨*proof*⟩

lemmas *omonadE* [*elim*!] =
opt-mapE obindE oreturnE ofailE othrowE oreturnOkE oassertE

lemma *in-opt-map-Some-eq*:
((*f* ||> *g*) *x* = *Some y*) = (∃ *v*. *f x* = *Some v* ∧ *g v* = *y*)
⟨*proof*⟩

lemma *in-opt-map-None-eq*[*simp*]:
((*f* ||> *g*) *x* = *None*) = (*f x* = *None*)
⟨*proof*⟩

lemma *oreturn-comp*[*simp*]:
oreturn x ∘ *f* = *oreturn x*
⟨*proof*⟩

lemma *ofail-comp*[*simp*]:
ofail ∘ *f* = *ofail*
⟨*proof*⟩

lemma *oassert-comp*[*simp*]:
oassert P ∘ *f* = *oassert P*
⟨*proof*⟩

lemma *fail-apply*[*simp*]:
ofail s = *None*
⟨*proof*⟩

lemma *oassert-apply*[*simp*]:
oassert P s = (*if P then Some* () *else None*)
⟨*proof*⟩

lemma *oreturn-apply*[*simp*]:
oreturn x s = *Some x*
⟨*proof*⟩

lemma *oapply-apply*[*simp*]:
oapply x s = *s x*
⟨*proof*⟩

lemma *obind-comp-dist*:
obind f g o h = *obind* (*f o h*) (λ*x*. *g x o h*)
⟨*proof*⟩

lemma *if-comp-dist*:
(*if P then f else g*) *o h* = (*if P then f o h else g o h*)

⟨*proof*⟩

8.1 "While" loops over option monad.

This is an inductive definition of a while loop over the plain option monad (without passing through a state)

inductive-set
  option-while′ :: (′*a* ⇒ *bool*) ⇒ (′*a* ⇒ ′*a option*) ⇒ ′*a option rel*
  for *C B*
where
    final: ¬ *C r* ⟹ (*Some r*, *Some r*) ∈ *option-while′ C B*
  | *fail*: ⟦ *C r*; *B r* = *None* ⟧ ⟹ (*Some r*, *None*) ∈ *option-while′ C B*
  | *step*: ⟦ *C r*; *B r* = *Some r′*; (*Some r′*, *sr″*) ∈ *option-while′ C B* ⟧
        ⟹ (*Some r*, *sr″*) ∈ *option-while′ C B*

definition
  option-while C B r ≡
    (*if* (∃ *s*. (*Some r*, *s*) ∈ *option-while′ C B*) *then*
      (*THE s*. (*Some r*, *s*) ∈ *option-while′ C B*) *else None*)

lemma *option-while′-inj*:
  assumes (*s*,*s′*) ∈ *option-while′ C B* (*s*, *s″*) ∈ *option-while′ C B*
  shows *s′* = *s″*
  ⟨*proof*⟩

lemma *option-while′-inj-step*:
  ⟦ *C s*; *B s* = *Some s′*; (*Some s*, *t*) ∈ *option-while′ C B* ; (*Some s′*, *t′*) ∈ *option-while′ C B* ⟧ ⟹ *t* = *t′*
  ⟨*proof*⟩

lemma *option-while′-THE*:
  assumes (*Some r*, *sr′*) ∈ *option-while′ C B*
  shows (*THE s*. (*Some r*, *s*) ∈ *option-while′ C B*) = *sr′*
  ⟨*proof*⟩

lemma *option-while-simps*:
  ¬ *C s* ⟹ *option-while C B s* = *Some s*
  C s ⟹ *B s* = *None* ⟹ *option-while C B s* = *None*
  C s ⟹ *B s* = *Some s′* ⟹ *option-while C B s* = *option-while C B s′*
  (*Some s*, *ss′*) ∈ *option-while′ C B* ⟹ *option-while C B s* = *ss′*
  ⟨*proof*⟩

lemma *option-while-rule*:
  assumes *option-while C B s* = *Some s′*
  assumes *I s*
  assumes *istep*: ⋀*s s′*. *C s* ⟹ *I s* ⟹ *B s* = *Some s′* ⟹ *I s′*
  shows *I s′* ∧ ¬ *C s′*
⟨*proof*⟩

lemma *option-while′-term*:
assumes *I r*
assumes *wf M*
assumes *step-less*: $\bigwedge r\ r'.\ [\![I\ r;\ C\ r;\ B\ r = Some\ r']\!] \Longrightarrow (r',r) \in M$
assumes *step-I*: $\bigwedge r\ r'.\ [\![I\ r;\ C\ r;\ B\ r = Some\ r']\!] \Longrightarrow I\ r'$
obtains *sr′* **where** $(Some\ r,\ sr') \in option\text{-}while'\ C\ B$
⟨*proof*⟩

lemma *option-while-rule′*:
assumes *option-while C B s = ss′*
assumes *wf M*
assumes *I* (*Some s*)
assumes *less*: $\bigwedge s\ s'.\ C\ s \Longrightarrow I\ (Some\ s) \Longrightarrow B\ s = Some\ s' \Longrightarrow (s',\ s) \in M$
assumes *step*: $\bigwedge s\ s'.\ C\ s \Longrightarrow I\ (Some\ s) \Longrightarrow B\ s = Some\ s' \Longrightarrow I\ (Some\ s')$
assumes *final*: $\bigwedge s.\ C\ s \Longrightarrow I\ (Some\ s) \Longrightarrow B\ s = None \Longrightarrow I\ None$
shows $I\ ss' \wedge (case\ ss'\ of\ Some\ s' \Rightarrow \neg\ C\ s' \mid \text{-} \Rightarrow True)$
⟨*proof*⟩

8.2 Lift *option-while* to the (′*a*, ′*s*) *lookup* monad

definition
owhile :: (′*a* ⇒ ′*s* ⇒ *bool*) ⇒ (′*a* ⇒ (′*s*,′*a*) *lookup*) ⇒ ′*a* ⇒ (′*s*,′*a*) *lookup*
where
owhile c b a ≡ λ*s*. *option-while* (λ*a*. *c a s*) (λ*a*. *b a s*) *a*

lemma *owhile-unroll*:
owhile C B r = *ocondition* (*C r*) (*B r* |>> *owhile C B*) (*oreturn r*)
⟨*proof*⟩

rule for terminating loops

lemma *owhile-rule*:
assumes *I r s*
assumes *wf M*
assumes *less*: $\bigwedge r\ r'.\ [\![I\ r\ s;\ C\ r\ s;\ B\ r\ s = Some\ r']\!] \Longrightarrow (r',r) \in M$
assumes *step*: $\bigwedge r\ r'.\ [\![I\ r\ s;\ C\ r\ s;\ B\ r\ s = Some\ r']\!] \Longrightarrow I\ r'\ s$
assumes *fail*: $\bigwedge r.\ [\![I\ r\ s;\ C\ r\ s;\ B\ r\ s = None]\!] \Longrightarrow Q\ None$
assumes *final*: $\bigwedge r.\ [\![I\ r\ s;\ \neg C\ r\ s]\!] \Longrightarrow Q\ (Some\ r)$
shows *Q* (*owhile C B r s*)
⟨*proof*⟩

end

theory *Option-MonadND*
imports
Reader-Monad

begin

definition
ogets :: (*'a* ⇒ *'b*) ⇒ (*'a* ⇒ *'b option*)
where
ogets f ≡ (*λs. Some* (*f s*))

definition
ocatch :: (*'s*,(*'e* + *'a*)) *lookup* ⇒ (*'e* ⇒ (*'s*,*'a*) *lookup*) ⇒ (*'s*, *'a*) *lookup*
(**infix** ‹<*ocatch*>› *10*)
where
f <*ocatch*> *handler* ≡ *do* {
x ← *f*;
case x of Inr b ⇒ *oreturn b* | *Inl e* ⇒ *handler e*
}

definition
odrop :: (*'s*, *'e* + *'a*) *lookup* ⇒ (*'s*, *'a*) *lookup*
where
odrop f ≡ *do* {
x ← *f*;
case x of Inr b ⇒ *oreturn b* | *Inl e* ⇒ *ofail*
}

definition
osequence-x :: (*'s*, *'a*) *lookup list* ⇒ (*'s*, *unit*) *lookup*
where
osequence-x xs ≡ *foldr* (*λx y. do* { *x*; *y* }) *xs* (*oreturn* ())

definition
osequence :: (*'s*, *'a*) *lookup list* ⇒ (*'s*, *'a list*) *lookup*
where
osequence xs ≡ *let mcons* = (*λp q. p* |>> (*λx. q* |>> (*λy. oreturn* (*x*#*y*))))
in foldr mcons xs (*oreturn* [])

definition
omap :: (*'a* ⇒ (*'s*,*'b*) *lookup*) ⇒ *'a list* ⇒ (*'s*, *'b list*) *lookup*
where
omap f xs ≡ *osequence* (*map f xs*)

definition
opt-cons :: *'a option* ⇒ *'a list* ⇒ *'a list* (**infixr** ‹*o*#› *65*)
where
opt-cons x xs ≡ *case x of None* ⇒ *xs* | *Some x'* ⇒ *x'* # *xs*

end

theory *Apply-Trace*
imports
  Main
begin

⟨*ML*⟩

end

theory *Apply-Trace-Cmd*
imports *Apply-Trace*
keywords *apply-trace* :: *prf-script*
begin

⟨*ML*⟩

lemmas [*no-trace*] = *protectI protectD TrueI Eq-TrueI eq-reflection*

lemma $(a \wedge b) = (b \wedge a)$
  apply-trace *auto*
  ⟨*proof*⟩

lemma $(a \wedge b) = (b \wedge a)$
  apply-trace ‹*intro*› *auto*
  ⟨*proof*⟩

lemma
  assumes *X*: $b = a$
  assumes *Y*: $b = a$
  shows
  $b = a$
  apply-trace (*rule Y*)
  ⟨*proof*⟩

locale *Apply-Trace-foo* = **fixes** *b a*
  assumes *X*: $b = a$
begin

lemma shows $b = a$ $b = a$
⟨*proof*⟩
apply-trace (*rule Apply-Trace-foo.X*)
⟨*proof*⟩
apply-trace (*rule X*)
⟨*proof*⟩
end

experiment begin

Example of trace for grouped lemmas

definition *ex* :: *nat set* **where**
ex = {*1*,*2*,*3*,*4*}

lemma *v1*: *1* ∈ *ex* ⟨*proof*⟩
lemma *v2*: *2* ∈ *ex* ⟨*proof*⟩
lemma *v3*: *3* ∈ *ex* ⟨*proof*⟩

Group several lemmas in a single one

lemmas *vs* = *v1 v2 v3*

lemma *2* ∈ *ex*
apply-trace (*simp add*: *vs*)
⟨*proof*⟩

end
end

Chapter 9

Mutual CCPO Recursion
fixed-point

theory *Mutual-CCPO-Recursion*
  imports
    HOL−Library.Complete-Partial-Order2
    AutoCorres-Utils
  keywords *fixed-point* :: *thy-goal-stmt*
begin

9.1 Relate orders between locale and type classes

lemma *gfp-lub-fun*: *gfp.lub-fun* = *Inf*
  ⟨*proof*⟩

lemma *gfp-le-fun*: *gfp.le-fun* = (≥)
  ⟨*proof*⟩

lemma
  shows *fst-prod-lub*: *fst* (*prod-lub luba lubb S*) = *luba* (*fst* ‘ *S*)
    and *snd-prod-lub*: *snd* (*prod-lub luba lubb S*) = *lubb* (*snd* ‘ *S*)
  ⟨*proof*⟩

lemma *fun-lub-app*: *fun-lub lub S x* = *lub* ((λ*f*. *f x*) ‘ *S*)
  ⟨*proof*⟩

lemma *ccpo-Inf*: *class.ccpo* (*Inf* :: *'a*::*complete-lattice set* ⇒ -) (≥) (*mk-less* (≥))
  ⟨*proof*⟩

lemma *ccpo-Sup*: *class.ccpo* (*Sup* :: *'a*::*complete-lattice set* ⇒ -) (≤) (*mk-less* (≤))
  ⟨*proof*⟩

lemma *ccpo-flat*: *class.ccpo* (*flat-lub b*) (*flat-ord b*) (*mk-less* (*flat-ord b*))
  ⟨*proof*⟩

lemma *ccpo-ccpo-class'*: *class.ccpo* (*Sup* :: *'a::ccpo set* ⇒ -) (≤) (<)
⟨*proof*⟩

lemma *monotone-pair*[*partial-function-mono*]:
monotone R orda f ⟹ *monotone R ordb g* ⟹ *monotone R* (*rel-prod orda ordb*) (λ*x*. (*f x*, *g x*))
⟨*proof*⟩

lemma *monotone-fst'*[*partial-function-mono*]:
monotone orda (*rel-prod ordb ordc*) *f* ⟹ *monotone orda ordb* (λ*x*. *fst* (*f x*))
⟨*proof*⟩

lemma *monotone-snd'*[*partial-function-mono*]:
monotone orda (*rel-prod ordb ordc*) *f* ⟹ *monotone orda ordc* (λ*x*. *snd* (*f x*))
⟨*proof*⟩

lemma *monotone-id-fun-elim*[*partial-function-mono*]:
monotone (≥) *ord* (λ*x*. *x*) ⟹ *monotone* (≥) (*fun-ord ord*) (λ*x*. *x*)
⟨*proof*⟩

lemma *monotone-id*[*partial-function-mono*]: *monotone ord ord* (λ*x*. *x*)
⟨*proof*⟩

lemma *monotone-abs*:
(⋀*x*. *monotone R Q* (λ*f*. *F f x*)) ⟹ *monotone R* (*fun-ord Q*) (λ*f x*. *F f x*)
⟨*proof*⟩

lemma *monotone-fun-ord-applyD*:
monotone orda (*fun-ord ordb*) *f* ⟹ *monotone orda ordb* (λ*y*. *f y x*)
⟨*proof*⟩

lemma *admissible-snd*:
cont luba orda (*prod-lub lubb Inf*) (*rel-prod ordb* (≥)) *F* ⟹
ccpo.admissible luba orda (λ*x*. *snd* (*F x*))
⟨*proof*⟩

lemma *cont-apply-gfp*: *cont gfp.lub-fun gfp.le-fun Inf* (≤) (λ*x*. *x c*)
⟨*proof*⟩

lemma *mcont-mem-gfp*:
mcont gfp.lub-fun gfp.le-fun (*Inf*) (≥) *F* ⟹ *gfp.admissible* (λ*A*. *x* ∈ *F A*)
⟨*proof*⟩

lemma *mcont2mcont-call*:
mcont luba orda (*fun-lub lubb*) (*fun-ord ordb*) *F* ⟹ *mcont luba orda lubb ordb* (λ*x*. *F x c*)
⟨*proof*⟩

lemma *preorder-fun-ord* [*partial-function-mono*]: *class.preorder R* (*mk-less R*) $\Longrightarrow$ *class.preorder* (*fun-ord R*) (*mk-less* (*fun-ord R*))
⟨*proof*⟩

lemma *preorder-monotone-const'*[*partial-function-mono*]:
class.preorder leq (*mk-less leq*) $\Longrightarrow$ *monotone ord leq* (λ-. *c*)
⟨*proof*⟩

declare *preorder-rel-prodI* [*partial-function-mono*]
declare *gfp.preorder* [*partial-function-mono*]
declare *lfp.preorder* [*partial-function-mono*]

lemma *option-ord-preorder* [*partial-function-mono*]: *class.preorder option-ord* (*mk-less option-ord*)
⟨*proof*⟩

9.2 Prove admissibility of *corresXF*

lemma *not-imp-not-iff*: $(\neg A \longrightarrow \neg B) \longleftrightarrow (B \longrightarrow A)$
⟨*proof*⟩

lemma *mcont-fun-lub-call*:
mcont luba orda (*fun-lub lubb*) (*fun-ord ordb*) *f* $\Longrightarrow$
mcont luba orda lubb ordb (λ*y*. *f y x*)
⟨*proof*⟩

lemma *chain-disj-of-subsingleton*:
Complete-Partial-Order.chain ord {*r*. *P r*} $\Longrightarrow$
Complete-Partial-Order.chain ord {*r*. *Q r*} $\Longrightarrow$
($\bigwedge r\ q$. *P r* $\Longrightarrow$ *Q q* $\Longrightarrow$ *r* = *q*) $\Longrightarrow$
Complete-Partial-Order.chain ord {*r*. *P r* ∨ *Q r*}
⟨*proof*⟩

definition *subsingleton-set* :: *'a set* ⇒ *bool* **where**
subsingleton-set P $\longleftrightarrow$ (∀ *a*∈*P*. ∀ *b*∈*P*. *a* = *b*)

lemma *subsingleton-set-empty*[*iff*]: *subsingleton-set* {}
⟨*proof*⟩

lemma *subsingleton-set-singleton*[*iff*]: *subsingleton-set* {*x*}
⟨*proof*⟩

context *ccpo*
begin

lemma *subsingleton-sets-imp-chain*:
subsingleton-set ($\bigcup$ *Ps*) $\Longrightarrow$ *Complete-Partial-Order.chain* (≤) ($\bigcup$ *Ps*)
⟨*proof*⟩

lemma *Sup-of-subsingleton-sets-eq*:
assumes *Ps*: *subsingleton-set* ($\bigcup$ *Ps*) **and** *P*: $P \in Ps$ **and** *a*: $a \in P$
shows *Sup* ($\bigcup$ *Ps*) = *a*
⟨*proof*⟩

lemma *monotone-Sup-of-subsingleton-sets*:
assumes *Ps*: $\bigwedge F.$ *subsingleton-set* ($\bigcup$ (*Ps F*))
and $*$: $\bigwedge F\ G.$ *ord F G* $\Longrightarrow$ *sim-set* (*sim-set* ($\leq$)) (*Ps F*) (*Ps G*)
shows *monotone ord* ($\leq$) ($\lambda F.$ *Sup* ($\bigcup$ (*Ps F*)))
⟨*proof*⟩

lemma *ccpo-refl*: $x \leq x$ ⟨*proof*⟩

lemma *fixp-unfold-def*:
fixes *f* :: *'a* ⇒ *'a*
assumes *def*: $F \equiv$ *ccpo.fixp Sup* ($\leq$) *f*
assumes *f*: *monotone* ($\leq$) ($\leq$) *f*
shows $F = f\ F$
⟨*proof*⟩

lemma *fixp-induct-def*:
fixes *f* :: *'a* ⇒ *'a*
assumes *def*: $F \equiv$ *ccpo.fixp Sup* ($\leq$) *f*
assumes *mono*: *monotone* ($\leq$) ($\leq$) *f*
assumes *adm*: *ccpo.admissible Sup* ($\leq$) *P*
assumes *bot*: *P* (*Sup* {})
assumes *step*: $\bigwedge x.$ *P x* $\Longrightarrow$ *P* (*f x*)
shows *P F*
⟨*proof*⟩

lemma *induct-Sup-of-subsingleton-sets*:
assumes *Ps*: *subsingleton-set* ($\bigcup$ *Ps*)
and *adm*: *ccpo.admissible Sup* ($\leq$) *P*
and *bot*: *P* (*Sup* {})
and *step*: $\bigwedge p\ x.\ p \in Ps \Longrightarrow x \in p \Longrightarrow P\ x$
shows *P* (*Sup* ($\bigcup$ *Ps*))
⟨*proof*⟩

lemma *induct-Sup-of-subsingleton-sets-def*:
assumes *F*: $F \equiv$ *Sup* ($\bigcup$ *Ps*)
assumes *Ps*: *subsingleton-set* ($\bigcup$ *Ps*)
and *adm*: *ccpo.admissible Sup* ($\leq$) *P*
and *bot*: *P* (*Sup* {})
and *step*: $\bigwedge p\ x.\ p \in Ps \Longrightarrow x \in p \Longrightarrow P\ x$
shows *P F*
⟨*proof*⟩

end

lemma *flat-lub-empty*: *flat-lub x* {} = *x*
⟨*proof*⟩

lemma *monotone-bind-option*[*partial-function-mono*]:
monotone ord option-ord f ⟹ (⋀*a*. *monotone ord option-ord* (λ*x*. *g x a*)) ⟹
monotone ord option-ord (λ*x*. *Option.bind* (*f x*) (*g x*))
⟨*proof*⟩

lemma (**in** *ccpo*) *admissible-ord*: *ccpo.admissible Sup* (≤) (λ*x*. *x* ≤ *b*)
⟨*proof*⟩

named-theorems *fixed-point-cleanup-simps* ‹*Simp set for fixed-point*›

lemma *monotone-const*[*partial-function-mono*]:
monotone R (≤) (λ*x*. (*c*::′*a*::*order*))
⟨*proof*⟩

lemma *Sup-empty-ccpo*[*simp*]:
Sup {} = (*bot*::′*a*::{*ccpo*, *order-bot*})
⟨*proof*⟩

lemmas [*fixed-point-cleanup-simps*] =
Inf-empty
Sup-empty
prod-lub-empty
fun-lub-empty
flat-lub-empty
Sup-empty-ccpo
fst-conv
snd-conv
split-paired-all
prod.collapse

⟨*ML*⟩

no-notation *top* (‹⊤›)
no-notation *bot* (‹⊥›)
no-notation *sup* (**infixl** ‹⊔› *65*)
no-notation *inf* (**infixl** ‹⊓› *70*)

hide-const (**open**) *cont*

end

Part II

C-Parser

Chapter 10

Theory Variants for Target Architectures via *L4V-ARCH*

theory *Target-Architecture*
  imports *Main*
  keywords
    if-architecture-by :: *qed-global* % *proof* **and**
    if-architecture-context :: *thy-decl-block*

begin

⟨*ML*⟩

end

Chapter 11

Unified Memory Model (UMM)

11.1 More Word Lemmas

This is a holding area for Word utility lemmas that are too specific or unpolished for the AFP, but which are reusable enough to be collected together for the rest of L4V. New utility lemmas that only prove facts about words should be added here (in preference to being kept where they were first needed).

theory *Word-Lemmas-Internal*
imports
Word-Lib.Word-Lemmas
Word-Lib.More-Word-Operations
Word-Lib.Many-More
Word-Lib.Word-Syntax
Word-Lib.Syntax-Bundles
begin

unbundle *bit-operations-syntax*
unbundle *bit-projection-infix-syntax*

lemmas *shiftl-nat-def* = *push-bit-eq-mult*[*of* - *a* **for** *a*::*nat*, *folded shiftl-def*]
lemmas *shiftr-nat-def* = *drop-bit-eq-div*[*of* - *a* **for** *a*::*nat*, *folded shiftr-def*]

declare *bit-simps*[*simp*]

lemma *signed-ge-zero-scast-eq-ucast*:
0 <=*s x* ⟹ *scast x* = *ucast x*
⟨*proof*⟩

lemma *disjCI2*:
(¬ *P* ⟹ *Q*) ⟹ *P* ∨ *Q*

⟨*proof*⟩

lemma *nat-diff-diff-le-lhs*:
$a + c - b \leq d \Longrightarrow a - (b - c) \leq (d :: nat)$
⟨*proof*⟩

lemma *is-aligned-obvious-no-wrap'*:
$[\![\textit{is-aligned ptr sz}; x = 2 \hat{} sz - 1]\!]$
$\Longrightarrow ptr \leq ptr + x$
⟨*proof*⟩

lemmas *add-ge0-weak* = *add-increasing*[**where** *'a=int* **and** *b=0*]

lemmas *aligned-sub-aligned* = *Aligned.aligned-sub-aligned'*

lemma *minus-minus-swap*:
$[\![a \leq c; b \leq d; b \leq a; d \leq c ; (d :: nat) - b = c - a]\!]$
$\Longrightarrow a - b = c - d$
⟨*proof*⟩

lemma *minus-minus-swap'*:
$[\![c \leq a; d \leq b; b \leq a; d \leq c ; (b :: nat) - d = a - c]\!]$
$\Longrightarrow a - b = c - d$
⟨*proof*⟩

lemmas *word-le-mask-eq* = *le-mask-imp-and-mask*

lemma *int-and-leR*:
$0 \leq b \Longrightarrow a \ AND \ b \leq (b :: int)$
⟨*proof*⟩

lemma *int-and-leL*:
$0 \leq a \Longrightarrow a \ AND \ b \leq (a :: int)$
⟨*proof*⟩

lemma *if-then-1-else-0*:
$(\textit{if } P \textit{ then } 1 \textit{ else } 0) = (0 :: 'a :: \textit{zero-neq-one}) \longleftrightarrow \neg P$
⟨*proof*⟩

lemma *if-then-0-else-1*:
$(\textit{if } P \textit{ then } 0 \textit{ else } 1) = (0 :: 'a :: \textit{zero-neq-one}) \longleftrightarrow P$
⟨*proof*⟩

lemmas *if-then-simps* = *if-then-0-else-1 if-then-1-else-0*

lemma *createNewCaps-guard*:
fixes $x :: 'a :: len \ word$
shows $[\![unat \ x = c; b < 2 \hat{} LENGTH('a)]\!]$

$\Longrightarrow (n < \textit{of-nat}\ b \wedge n < x) = (n < \textit{of-nat}\ (\textit{min}\ (\textit{min}\ b\ c)\ c))$
⟨*proof*⟩

lemma *bits-2-subtract-ineq*:
$i < (n :: ('a :: len)\ word)$
$\Longrightarrow 2 \hat{}\ \textit{bits} + 2 \hat{}\ \textit{bits} * \textit{unat}\ (n - (1 + i)) = \textit{unat}\ (n - i) * 2 \hat{}\ \textit{bits}$
⟨*proof*⟩

lemmas *double-neg-mask* = *neg-mask-combine*

lemmas *int-unat* = *uint-nat*[*symmetric*]

lemmas *word-sub-mono3* = *word-plus-mcs-4'*

lemma *shift-distinct-helper*:
$[\![(x :: 'a :: len\ word) < \textit{bnd}; y < \textit{bnd}; x \neq y; x << n = y << n; n < \textit{LENGTH}('a);$
$\textit{bnd} - 1 \leq 2 \hat{}\ (\textit{LENGTH}('a) - n) - 1]\!]$
$\Longrightarrow P$
⟨*proof*⟩

lemma *of-nat-shift-distinct-helper*:
$[\![x < \textit{bnd}; y < \textit{bnd}; x \neq y; (\textit{of-nat}\ x :: 'a :: len\ word) << n = \textit{of-nat}\ y << n;$
$n < \textit{LENGTH}('a); \textit{bnd} \leq 2 \hat{}\ (\textit{LENGTH}('a) - n)]\!]$
$\Longrightarrow P$
⟨*proof*⟩

lemmas *pre-helper2* = *add-mult-in-mask-range*[*folded add-mask-fold*]

lemma *ptr-add-distinct-helper*:
$[\![\textit{ptr-add}\ (p :: 'a :: len\ word)\ (x * 2 \hat{}\ n) = \textit{ptr-add}\ p\ (y * 2 \hat{}\ n); x \neq y;$
$x < \textit{bnd}; y < \textit{bnd}; n < \textit{LENGTH}('a);$
$\textit{bnd} \leq 2 \hat{}\ (\textit{LENGTH}('a) - n)]\!]$
$\Longrightarrow P$
⟨*proof*⟩

lemma *unat-sub-le-strg*:
$\textit{unat}\ v \leq v2 \wedge x \leq v \wedge y \leq v \wedge y < (x :: ('a :: len)\ word)$
$\longrightarrow \textit{unat}\ (x + (-1 - y)) \leq v2$
⟨*proof*⟩

lemma *multi-lessD*:
$[\![(a :: nat) * b < c; 0 < a; 0 < b]\!]$
$\Longrightarrow a < c \wedge b < c$
⟨*proof*⟩

lemmas *leq-high-bits-shiftr-low-bits-leq-bits* =
leq-high-bits-shiftr-low-bits-leq-bits-mask[*unfolded mask-2pm1*[*of high-bits*]]

lemmas *unat-le-helper* = *word-unat-less-le*

lemmas *word-of-nat-plus* = *of-nat-add*[**where** *'a='a :: len word*]
lemmas *word-of-nat-minus* = *of-nat-diff*[**where** *'a='a :: len word*]

lemma *word-up-bound*:
(*ptr :: 'a :: len word*) ≤ *2 ^ LENGTH('a) − 1*
⟨*proof*⟩

lemma *base-length-minus-one-inequality*:
assumes *foo*: *wbase* ≤ *2 ^ sz − 1*
1 ≤ (*wlength :: ('a :: len) word*)
wlength ≤ *2 ^ sz − wbase*
sz < *LENGTH ('a)*
shows *wbase* ≤ *wbase + wlength − 1*
⟨*proof*⟩

lemmas *from-bool-to-bool-and-1* = *from-to-bool-last-bit*[**where** *x=r* **for** *r*]

lemmas *max-word-neq-0* = *max-word-not-0*

lemmas *word-le-p2m1* = *word-up-bound*[**where** *ptr=w* **for** *w*]

lemma *inj-ucast*:
⟦ *uc* = *ucast*; *is-up uc* ⟧
⟹ *inj uc*
⟨*proof*⟩

lemma *ucast-eq-0*[*OF refl*]:
⟦ *c* = *ucast*; *is-up c* ⟧
⟹ (*c x* = *0*) = (*x* = *0*)
⟨*proof*⟩

lemmas *is-up-compose'* = *is-up-compose*

lemma *uint-is-up-compose*:
fixes *uc* :: *'a :: len word* ⇒ *'b :: len word*
and *uc'* :: *'b word* ⇒ *'c :: len sword*
assumes *uc* = *ucast*
and *uc'* = *ucast*
and *uuc* = *uc'* ∘ *uc*
shows ⟦ *is-up uc*; *is-up uc'* ⟧
⟹ *uint* (*uuc b*) = *uint b*
⟨*proof*⟩

lemma *uint-is-up-compose-pred*:
fixes *uc* :: *'a :: len word* ⇒ *'b :: len word*
and *uc'* :: *'b word* ⇒ *'c :: len sword*
assumes *uc* = *ucast* **and** *uc'* = *ucast* **and** *uuc* = *uc'* ∘ *uc*

shows ⟦ *is-up uc*; *is-up uc′* ⟧
⟹ *P* (*uint* (*uuc b*)) ⟷ *P*(*uint b*)
⟨*proof*⟩

lemma *is-down-up-sword*:
fixes *uc* :: *′a* :: *len word* ⇒ *′b* :: *len sword*
shows ⟦ *uc* = *ucast*; *LENGTH*(*′a*) < *LENGTH*(*′b*) ⟧
⟹ *is-up uc* = (¬ *is-down uc*)
⟨*proof*⟩

lemma *is-not-down-compose*:
fixes *uc* :: *′a* :: *len word* ⇒ *′b* :: *len word*
and *uc′* :: *′b word* ⇒ *′c* :: *len sword*
shows ⟦ *uc* = *ucast*; *uc′* = *ucast*; *LENGTH*(*′a*) < *LENGTH*(*′c*) ⟧
⟹ ¬ *is-down* (*uc′* ∘ *uc*)
⟨*proof*⟩

lemma *sint-ucast-uint*:
fixes *uc* :: *′a* :: *len word* ⇒ *′b* :: *len word*
and *uc′* :: *′b word* ⇒ *′c* :: *len sword*
assumes *uc* = *ucast* **and** *uc′* = *ucast* **and** *uuc*=*uc′* ∘ *uc*
and *LENGTH*(*′a*) < *LENGTH*(*′c signed*)
shows ⟦ *is-up uc*; *is-up uc′*⟧
⟹ *sint* (*uuc b*) = *uint b*
⟨*proof*⟩

lemma *sint-ucast-uint-pred*:
fixes *uc* :: *′a* :: *len word* ⇒ *′b* :: *len word*
and *uc′* :: *′b word* ⇒ *′c* :: *len sword*
and *uuc* :: *′a word* ⇒ *′c sword*
assumes *uc* = *ucast* **and** *uc′* = *ucast* **and** *uuc*=*uc′* ∘ *uc*
and *LENGTH*(*′a*) < *LENGTH*(*′c*)
shows ⟦ *is-up uc*; *is-up uc′* ⟧
⟹ *P* (*uint b*) ⟷ *P* (*sint* (*uuc b*))
⟨*proof*⟩

lemma *sint-uucast-uint-uucast-pred*:
fixes *uc* :: *′a* :: *len word* ⇒ *′b* :: *len word*
and *uc′* :: *′b word* ⇒ *′c* :: *len sword*
assumes *uc* = *ucast* **and** *uc′* = *ucast* **and** *uuc*=*uc′* ∘ *uc*
and *LENGTH*(*′a*) < *LENGTH*(*′c*)
shows ⟦ *is-up uc*; *is-up uc′* ⟧
⟹ *P* (*uint*(*uuc b*)) ⟷ *P* (*sint* (*uuc b*))
⟨*proof*⟩

lemma *unat-minus′*:
fixes *x* :: *′a* :: *len word*
shows *x* ≠ *0* ⟹ *unat* (−*x*) = *2* ^ *LENGTH*(*′a*) − *unat x*
⟨*proof*⟩

lemma *word-nth-neq*:
$n < LENGTH('a) \Longrightarrow (\sim\sim\ x :: 'a :: len\ word)\ !!\ n = (\neg\ x\ !!\ n)$
⟨*proof*⟩

lemma *word-wrap-of-natD*:
fixes $x :: 'a :: len\ word$
assumes *wraps*: $\neg\ x \leq x + of\text{-}nat\ n$
shows $\exists k.\ x + of\text{-}nat\ k = 0 \wedge k \leq n$
⟨*proof*⟩

lemma *two-bits-cases*:
$[\![\ LENGTH('a) > 2; (x :: 'a :: len\ word)\ \&\&\ 3 = 0 \Longrightarrow P; x\ \&\&\ 3 = 1 \Longrightarrow P;$
$x\ \&\&\ 3 = 2 \Longrightarrow P; x\ \&\&\ 3 = 3 \Longrightarrow P\ ]\!]$
$\Longrightarrow P$
⟨*proof*⟩

lemma *zero-OR-eq*:
$y = 0 \Longrightarrow (x\ ||\ y) = x$
⟨*proof*⟩

declare *is-aligned-neg-mask-eq*[*simp*]
declare *is-aligned-neg-mask-weaken*[*simp*]

lemmas *mask-in-range* = *neg-mask-in-mask-range*[*folded add-mask-fold*]
lemmas *aligned-range-offset-mem* = *aligned-offset-in-range*[*folded add-mask-fold*]
lemmas *range-to-bl'* = *mask-range-to-bl'*[*folded add-mask-fold*]
lemmas *range-to-bl* = *mask-range-to-bl*[*folded add-mask-fold*]
lemmas *aligned-ranges-subset-or-disjoint* = *aligned-mask-range-cases*[*folded add-mask-fold*]
lemmas *aligned-range-offset-subset* = *aligned-mask-range-offset-subset*[*folded add-mask-fold*]
lemmas *aligned-diff* = *aligned-mask-diff*[*unfolded mask-2pm1*]
lemmas *aligned-ranges-subset-or-disjoint-coroll* = *aligned-mask-ranges-disjoint*[*folded add-mask-fold*]
lemmas *distinct-aligned-addresses-accumulate* = *aligned-mask-ranges-disjoint2*[*folded add-mask-fold*]

lemmas *bang-big* = *test-bit-over*

lemma *unat-and-mask-le*:
$n < LENGTH('a) \Longrightarrow unat\ (x\ \&\&\ mask\ n) \leq 2\hat{}n$ **for** $x::'a::len\ word$
⟨*proof*⟩

lemma *sign-extend-less-mask-idem*:
$[\![\ w \leq mask\ n; n < size\ w\ ]\!] \Longrightarrow sign\text{-}extend\ n\ w = w$
⟨*proof*⟩

lemma *word-and-le*:
$a \leq c \Longrightarrow (a :: 'a :: len\ word)\ \&\&\ b \leq c$
⟨*proof*⟩

lemma *le-smaller-mask*:
⟦ $x \leq mask\ n$; $n \leq m$ ⟧ ⟹ $x \leq mask\ m$ **for** $x :: 'a::len\ word$
⟨*proof*⟩

lemma *upcast-less-unat-less*:
assumes *less*: $UCAST('a \to 'b)\ x < UCAST('a \to 'b)\ (of\text{-}nat\ y)$
assumes *len*: $LENGTH('a::len) \leq LENGTH('b::len)$
assumes *bound*: $y < 2\ \hat{}\ LENGTH('a)$
shows $unat\ x < y$
⟨*proof*⟩

lemma *word-ctz-max*:
$word\text{-}ctz\ w \leq size\ w$
⟨*proof*⟩

lemma *scast-of-nat-small*:
$x < 2\ \hat{}\ (LENGTH('a) - 1) \Longrightarrow scast\ (of\text{-}nat\ x :: 'a :: len\ word) = (of\text{-}nat\ x :: 'b :: len\ word)$
⟨*proof*⟩

lemmas *casts-of-nat-small* = *ucast-of-nat-small scast-of-nat-small*

— The conditions under which ‘takeWhile P xs = take n xs‘ and ‘dropWhile P xs = drop n xs‘
definition *list-while-len* **where**
$list\text{-}while\text{-}len\ P\ n\ xs \equiv (\forall i.\ i < n \longrightarrow i < length\ xs \longrightarrow P\ (xs\ !\ i)) \wedge (n < length\ xs \longrightarrow \neg\ P\ (xs\ !\ n))$

lemma *list-while-len-iff-takeWhile-eq-take*:
$list\text{-}while\text{-}len\ P\ n\ xs \longleftrightarrow takeWhile\ P\ xs = take\ n\ xs$
⟨*proof*⟩

lemma *list-while-len-exists*:
$\exists n.\ list\text{-}while\text{-}len\ P\ n\ xs$
⟨*proof*⟩

lemma *takeWhile-truncate*:
$length\ (takeWhile\ P\ xs) \leq m \Longrightarrow takeWhile\ P\ (take\ m\ xs) = takeWhile\ P\ xs$
⟨*proof*⟩

lemma *word-clz-shiftr-1*:
fixes $z::'a::len\ word$
assumes *wordsize*: $1 < LENGTH('a)$
shows $z \neq 0 \Longrightarrow word\text{-}clz\ (z >> 1) = word\text{-}clz\ z + 1$
⟨*proof*⟩

lemma *shiftr-Suc*:
fixes *x*::*'a*::*len word*
shows *x >> (Suc n) = x >> n >> 1*
⟨*proof*⟩

lemma *word-clz-shiftr*:
fixes *z* :: *'a*::*len word*
shows *n < LENGTH('a) ⟹ mask n < z ⟹ word-clz (z >> n) = word-clz z + n*
⟨*proof*⟩

lemma *mask-to-bl-exists-True*:
x && mask n ≠ 0 ⟹ ∃ m. (rev (to-bl x)) ! m ∧ m < n
⟨*proof*⟩

lemma *word-ctz-shiftr-1*:
fixes *z*::*'a*::*len word*
assumes *wordsize*: *1 < LENGTH('a)*
shows *z ≠ 0 ⟹ 1 ≤ word-ctz z ⟹ word-ctz (z >> 1) = word-ctz z − 1*
⟨*proof*⟩

lemma *word-ctz-bound-below-helper*:
fixes *x* :: *'a*::*len word*
assumes *sz*: *n ≤ LENGTH('a)*
shows *x && mask n = 0*
⟹ to-bl x = (take (LENGTH('a) − n) (to-bl x) @ replicate n False)
⟨*proof*⟩

lemma *word-ctz-bound-below*:
fixes *x* :: *'a*::*len word*
assumes *sz*[*simp*]: *n ≤ LENGTH('a)*
shows *x && mask n = 0 ⟹ n ≤ word-ctz x*
⟨*proof*⟩

lemma *word-ctz-bound-above*:
fixes *x* :: *'a*::*len word*
shows *x && mask n ≠ 0 ⟹ word-ctz x < n*
⟨*proof*⟩

lemma *word-ctz-shiftr*:
fixes *z*::*'a*::*len word*
assumes *nz*: *z ≠ 0*
shows *n < LENGTH('a) ⟹ n ≤ word-ctz z ⟹ word-ctz (z >> n) = word-ctz z − n*
⟨*proof*⟩
lemma *word-less-max-simp*:
fixes *w* :: *'a*::*len word*
assumes *max-w = −1*
shows *w ≤ max-w*

⟨*proof*⟩

lemma *word-and-mask-word-ctz-zero*:
assumes $l = \textit{word-ctz}\ w$
shows $w\ \&\&\ \textit{mask}\ l = 0$
⟨*proof*⟩

lemma *word-ctz-len-word-and-mask-zero*:
fixes $w :: 'a{::}\textit{len word}$
shows $\textit{word-ctz}\ w = \textit{LENGTH}('a) \Longrightarrow w = 0$
⟨*proof*⟩

lemma *word-le-1*:
fixes $w :: 'a{::}\textit{len word}$
shows $w \leq 1 \longleftrightarrow w = 0 \vee w = 1$
⟨*proof*⟩

lemma *less-ucast-ucast-less'*:
$x < \textit{UCAST}('b \to 'a)\ y \Longrightarrow \textit{UCAST}('a \to 'b)\ x < y$
for $x :: 'a{::}\textit{len word}$ **and** $y :: 'b{::}\textit{len word}$
⟨*proof*⟩

lemma *ucast-up-less-bounded-implies-less-ucast-down'*:
assumes *len*: $\textit{LENGTH}('a{::}\textit{len}) < \textit{LENGTH}('b{::}\textit{len})$
assumes *bound*: $y < 2\ \hat{}\ \textit{LENGTH}('a)$
assumes *less*: $\textit{UCAST}('a \to 'b)\ x < y$
shows $x < \textit{UCAST}('b \to 'a)\ y$
⟨*proof*⟩

lemma *ucast-up-less-bounded-iff-less-ucast-down'*:
assumes *len*: $\textit{LENGTH}('a{::}\textit{len}) < \textit{LENGTH}('b{::}\textit{len})$
assumes *bound*: $y < 2\ \hat{}\ \textit{LENGTH}('a)$
shows $\textit{UCAST}('a \to 'b)\ x < y \longleftrightarrow x < \textit{UCAST}('b \to 'a)\ y$
⟨*proof*⟩

lemma *word-of-int-word-of-nat-eqD*:
$[\![\ \textit{word-of-int}\ x = (\textit{word-of-nat}\ y :: 'a :: \textit{len word});\ 0 \leq x;\ x < 2\hat{}\textit{LENGTH}('a);$
$y < 2\hat{}\textit{LENGTH}('a)\ ]\!]$
$\Longrightarrow \textit{nat}\ x = y$
⟨*proof*⟩

lemma *ucast-down-0*:
$[\![\ \textit{UCAST}('a{::}\textit{len} \to 'b{::}\textit{len})\ x = 0;\ \textit{unat}\ x < 2\hat{}\textit{LENGTH}('b)\ ]\!] \Longrightarrow x = 0$
⟨*proof*⟩

lemma *uint-minus-1-eq*:
‹$\textit{uint}\ (-\ 1 :: 'a\ \textit{word}) = 2\ \hat{}\ \textit{LENGTH}('a{::}\textit{len}) - 1$›
⟨*proof*⟩

lemma *FF-eq-minus-1*:
‹*0xFF* = (− *1* :: *8 word*)›
⟨*proof*⟩

lemma *shiftl-t2n'*:
w << *n* = *w* ∗ (*2* ^ *n*) **for** *w* :: *'a*::*len word*
⟨*proof*⟩

end

theory *Word-Lemmas-32-Internal*
imports *Word-Lib.Word-Lib-Sumo Word-Lib.Machine-Word-32 Word-Lemmas-Internal*
begin

lemmas *sint-eq-uint-32* = *sint-eq-uint-2pl*[**where** *'a=32*, *simplified*]

lemmas *sle-positive-32* = *sle-le-2pl*[**where** *'a=32*, *simplified*]

lemmas *sless-positive-32* = *sless-less-2pl*[**where** *'a=32*, *simplified*]

lemma *zero-le-sint-32*:
⟦ *0* ≤ (*a* :: *word32*); *a* < *0x80000000* ⟧
⟹ *0* ≤ *sint a*
⟨*proof*⟩

lemmas *unat-add-simple* = *iffD1*[*OF unat-add-lem*[**where** *'a* = *32*, *folded word-bits-def*]]

lemma *upto-enum-inc-1*:
a < *2* ^ *word-bits* − *1*
⟹ [(*0*:: *'a* :: *len word*) *.e.* *1* + *a*] = [*0.e.a*] @ [(*1*+*a*)]
⟨*proof*⟩

lemmas *upt-enum-offset-trivial* =
upt-enum-offset-trivial[**where** *'a=32*, *folded word-bits-def*]

lemmas *unat32-eq-of-nat* = *unat-eq-of-nat*[**where** *'a=32*, *folded word-bits-def*]

declare *mask-32-max-word*[*simp*]

lemma *le-32-mask-eq*:
(*bits* :: *word32*) ≤ *32* ⟹ *bits* && *mask 6* = *bits*
⟨*proof*⟩

lemmas *scast-1-32*[*simp*] = *scast-1*[**where** *'a=32*]

lemmas *mask-32-id*[*simp*] = *mask-len-id*[**where** *'a=32*, *folded word-bits-def*]

lemmas *t2p-shiftr-32* = *t2p-shiftr*[**where** *'a=32*, *folded word-bits-def*]

lemma *mask-eq1-nochoice*:
(*x* :: *word32*) && *1* = *x*
⟹ *x* = *0* ∨ *x* = *1*
⟨*proof*⟩

lemmas *const-le-unat-word-32* = *const-le-unat*[**where** *'a=32*, *folded word-bits-def*]

lemmas *createNewCaps-guard-helper* =
createNewCaps-guard[**where** *'a=32*, *folded word-bits-def*]

lemma *word-log2-max-word32*[*simp*]:
word-log2 (*w* :: *32 word*) < *32*
⟨*proof*⟩

lemma *mapping-two-power-16-64-inequality*:
assumes *sz*: *sz* ≤ *4* **and** *len*: *unat* (*len* :: *word32*) = *2* ^ *sz*
shows *unat* (*len* ∗ *8* − *1*) ≤ *127*
⟨*proof*⟩

lemmas *pre-helper2-32* = *pre-helper2*[**where** *'a=32*, *folded word-bits-def*]

lemmas *of-nat-shift-distinct-helper-machine* =
of-nat-shift-distinct-helper[**where** *'a=32*, *folded word-bits-def*]

lemmas *ptr-add-distinct-helper-32* =
ptr-add-distinct-helper[**where** *'a=32*, *folded word-bits-def*]

lemmas *mask-out-eq-0-32* = *mask-out-eq-0*[**where** *'a=32*, *folded word-bits-def*]

lemmas *neg-mask-mask-unat-32* = *neg-mask-mask-unat*[**where** *'a=32*, *folded word-bits-def*]

lemmas *unat-less-iff-32* = *unat-less-iff*[**where** *'a=32*, *folded word-bits-def*]

lemmas *is-aligned-no-overflow3-32* = *is-aligned-no-overflow3*[**where** *'a=32*, *folded word-bits-def*]

lemmas *unat-ucast-16-32* = *unat-signed-ucast-less-ucast*[**where** *'a=16* **and** *'b=32*, *simplified*]

lemma *scast-mask-8*:
scast (*mask 8* :: *sword32*) = (*mask 8* :: *word32*)
⟨*proof*⟩

lemmas *ucast-le-8-32-equiv* = *ucast-le-up-down-iff*[**where** *'a=8* **and** *'b=32*, *simplified*]

lemma *signed-unat-minus-one-32*:
unat (−*1* :: *32 signed word*) = *4294967295*
⟨*proof*⟩

lemmas *two-bits-cases-32* = *two-bits-cases*[**where** *'a=32*, *simplified*]

lemmas *word-ctz-not-minus-1-32* = *word-ctz-not-minus-1*[**where** *'a=32*, *simplified*]

lemmas *sint-ctz-32* = *sint-ctz*[**where** *'a=32*, *simplified*]

lemmas *scast-specific-plus32* =
scast-of-nat-signed-to-unsigned-add[**where** *'a=32* **and** *x=word-ctz x* **and** *y=0x20* **for** *x*,
simplified]
lemmas *scast-specific-plus32-signed* =
scast-of-nat-unsigned-to-signed-add[**where** *'a=32* **and** *x=word-ctz x* **and** *y=0x20* **for** *x*,
simplified]

lemma *neq-0-unat*: *x* ≠ *0* ⟹ *0* < *unat x* **for** *x*::*machine-word*
⟨*proof*⟩

end

theory *Word-Lemmas-64-Internal*
imports *Word-Lib.Word-Lib-Sumo Word-Lib.Word-64 Word-Lib.Machine-Word-64*
begin

lemmas *word-and-max-simps* = *word-and-max-simps word64-and-max-simp*

lemmas *unat-add-simple* = *iffD1*[*OF unat-add-lem*[**where** *'a* = *64*, *folded word-bits-def*]]

lemma *unat-length-4-helper*:
assumes *unat* (*l*::*machine-word*) = *length args* ¬ *l* < *4*
shows ∃ *x xa xb xc xs*. *args* = *x*#*xa*#*xb*#*xc*#*xs*
⟨*proof*⟩

lemma *ucast-drop-big-mask*:
UCAST(*64* → *16*) (*x* &&& *0xFFFF*) = *UCAST*(*64* → *16*) *x*
⟨*proof*⟩

lemma *first-port-last-port-compare*:
UCAST(*16* → *32 signed*) (*UCAST*(*64* → *16*) (*xa* &&& *0xFFFF*))
<*s UCAST*(*16* → *32 signed*) (*UCAST*(*64* → *16*) (*x* &&& *0xFFFF*))
= (*UCAST*(*64* → *16*) *xa* < *UCAST*(*64* → *16*) *x*)

⟨*proof*⟩

lemma *machine-word-and-3F-less-40*:
(*w* :: *machine-word*) && *0x3F* < *0x40*
⟨*proof*⟩

lemmas *unat64-eq-of-nat* = *unat-eq-of-nat*[**where** *'a=64*, *folded word-bits-def*]

lemma *unat-mask-3-less-8*:
unat (*p* && *mask 3* :: *word64*) < *8*
⟨*proof*⟩

lemma *scast-specific-plus64*:
scast (*of-nat* (*word-ctz x*) + *0x20* :: *64 signed word*) = *of-nat* (*word-ctz x*) + (*0x20* :: *machine-word*)
⟨*proof*⟩

lemma *scast-specific-plus64-signed*:
scast (*of-nat* (*word-ctz x*) + *0x20* :: *machine-word*) = *of-nat* (*word-ctz x*) + (*0x20* :: *64 signed word*)
⟨*proof*⟩

lemmas *mask-64-id*[*simp*] = *mask-len-id*[**where** *'a=64*, *folded word-bits-def*]
mask-len-id[**where** *'a=64*, *simplified*]

lemma *neq-0-unat*: *x* ≠ *0* ⟹ *0* < *unat x* **for** *x*::*machine-word*
⟨*proof*⟩

end

11.2 Distinct Proposition

theory *Distinct-Prop*
imports
Word-Lib.Many-More
HOL−Library.Prefix-Order
begin

primrec
distinct-prop :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'a list* ⇒ *bool*)
where
distinct-prop P [] = *True*
| *distinct-prop P* (*x* # *xs*) = ((∀ *y*∈*set xs*. *P x y*) ∧ *distinct-prop P xs*)

primrec
distinct-sets :: *'a set list* ⇒ *bool*
where
distinct-sets [] = *True*
| *distinct-sets* (*x*#*xs*) = (*x* ∩ ⋃ (*set xs*) = {} ∧ *distinct-sets xs*)

declare *distinct-sets.simps* [*simp del*]

lemma *distinct-prop-map*:
distinct-prop P (*map f xs*) = *distinct-prop* (λ*x y*. *P* (*f x*) (*f y*)) *xs*
⟨*proof*⟩

lemma *distinct-prop-append*:
distinct-prop P (*xs* @ *ys*) =
(*distinct-prop P xs* ∧ *distinct-prop P ys* ∧ (∀ *x* ∈ *set xs*. ∀ *y* ∈ *set ys*. *P x y*))
⟨*proof*⟩

lemma *distinct-prop-distinct*:
⟦ *distinct xs*; ⋀*x y*. ⟦ *x* ∈ *set xs*; *y* ∈ *set xs*; *x* ≠ *y* ⟧ ⟹ *P x y* ⟧ ⟹ *distinct-prop P xs*
⟨*proof*⟩

lemma *distinct-prop-True* [*simp*]:
distinct-prop (λ*x y*. *True*) *xs*
⟨*proof*⟩

lemma *distinct-prefix*:
⟦ *distinct xs*; *ys* ≤ *xs* ⟧ ⟹ *distinct ys*
⟨*proof*⟩

lemma *distinct-sets-prop*:
distinct-sets xs = *distinct-prop* (λ*x y*. *x* ∩ *y* = {}) *xs*
⟨*proof*⟩

lemma *distinct-take-strg*:
distinct xs ⟶ *distinct* (*take n xs*)
⟨*proof*⟩

lemma *distinct-prop-prefixE*:
⟦ *distinct-prop P ys*; *prefix xs ys* ⟧ ⟹ *distinct-prop P xs*
⟨*proof*⟩

lemma *distinct-sets-union-sub*:
⟦*x* ∈ *A*; *distinct-sets* [*A*,*B*]⟧ ⟹ *A* ∪ *B* − {*x*} = *A* − {*x*} ∪ *B*
⟨*proof*⟩

lemma *distinct-sets-append*:
distinct-sets (*xs* @ *ys*) ⟹ *distinct-sets xs* ∧ *distinct-sets ys*
⟨*proof*⟩

lemma *distinct-sets-append1*:
distinct-sets (*xs* @ *ys*) ⟹ *distinct-sets xs*
⟨*proof*⟩

lemma *distinct-sets-append2*:
distinct-sets (*xs* @ *ys*) ⟹ *distinct-sets ys*
⟨*proof*⟩

lemma *distinct-sets-append-Cons*:
distinct-sets (*xs* @ *a* # *ys*) ⟹ *distinct-sets* (*xs* @ *ys*)
⟨*proof*⟩

lemma *distinct-sets-append-Cons-disjoint*:
distinct-sets (*xs* @ *a* # *ys*) ⟹ *a* ∩ ⋃ (*set xs*) = {}
⟨*proof*⟩

lemma *distinct-prop-take*:
⟦*distinct-prop P xs*; *i* < *length xs*⟧ ⟹ *distinct-prop P* (*take i xs*)
⟨*proof*⟩

lemma *distinct-sets-take*:
⟦*distinct-sets xs*; *i* < *length xs*⟧ ⟹ *distinct-sets* (*take i xs*)
⟨*proof*⟩

lemma *distinct-prop-take-Suc*:
⟦*distinct-prop P xs*; *i* < *length xs*⟧ ⟹ *distinct-prop P* (*take* (*Suc i*) *xs*)
⟨*proof*⟩

lemma *distinct-sets-take-Suc*:
⟦*distinct-sets xs*; *i* < *length xs*⟧ ⟹ *distinct-sets* (*take* (*Suc i*) *xs*)
⟨*proof*⟩

lemma *distinct-prop-rev*:
distinct-prop P (*rev xs*) = *distinct-prop* (λ*y x*. *P x y*) *xs*
⟨*proof*⟩

lemma *distinct-sets-rev* [*simp*]:
distinct-sets (*rev xs*) = *distinct-sets xs*
⟨*proof*⟩

lemma *distinct-sets-drop*:
⟦*distinct-sets xs*; *i* < *length xs*⟧ ⟹ *distinct-sets* (*drop i xs*)
⟨*proof*⟩

lemma *distinct-sets-drop-Suc*:
⟦*distinct-sets xs*; *i* < *length xs*⟧ ⟹ *distinct-sets* (*drop* (*Suc i*) *xs*)
⟨*proof*⟩

lemma *distinct-sets-take-nth*:
⟦*distinct-sets xs*; *i* < *length xs*; *x* ∈ *set* (*take i xs*)⟧ ⟹ *x* ∩ *xs* ! *i* = {}
⟨*proof*⟩

lemma *distinct-sets-drop-nth*:
⟦*distinct-sets xs*; *i* < *length xs*; *x* ∈ *set* (*drop* (*Suc i*) *xs*)⟧ ⟹ *x* ∩ *xs* ! *i* = {}
⟨*proof*⟩

lemma *distinct-sets-append-distinct*:
⟦*x* ∈ *set xs*; *y* ∈ *set ys*; *distinct-sets* (*xs* @ *ys*)⟧ ⟹ *x* ∩ *y* = {}
⟨*proof*⟩

lemma *distinct-sets-update*:
⟦*a* ⊆ *xs* ! *i*; *distinct-sets xs*; *i* < *length xs*⟧ ⟹ *distinct-sets* (*xs*[*i* := *a*])
⟨*proof*⟩

lemma *distinct-sets-map-update*:
⟦*distinct-sets* (*map f xs*); *i* < *length xs*; *f a* ⊆ *f*(*xs* ! *i*)⟧
⟹ *distinct-sets* (*map f* (*xs*[*i* := *a*]))
⟨*proof*⟩

lemma *Union-list-update*:
⟦*i* < *length xs*; *distinct-sets* (*map f xs*)⟧
⟹ (⋃*x*∈*set* (*xs* [*i* := *a*]). *f x*) = (⋃*x*∈*set xs*. *f x*) − *f* (*xs* ! *i*) ∪ *f a*
⟨*proof*⟩

lemma *fst-enumerate*:
i < *length xs* ⟹ *fst* (*enumerate n xs* ! *i*) = *i* + *n*
⟨*proof*⟩

lemma *snd-enumerate*:
i < *length xs* ⟹ *snd* (*enumerate n xs* ! *i*) = *xs* ! *i*
⟨*proof*⟩

lemma *enumerate-member*:
assumes *i* < *length xs*
shows (*n* + *i*, *xs* ! *i*) ∈ *set* (*enumerate n xs*)
⟨*proof*⟩

lemma *distinct-prop-nth*:
⟦ *distinct-prop P ls*; *n* < *n*′; *n*′ < *length ls* ⟧ ⟹ *P* (*ls* ! *n*) (*ls* ! *n*′)
⟨*proof*⟩

lemma *distinct-prop-iff*:
⟦ ⋀*x y*. *P x y* ⟷ *Q x y* ⟧ ⟹ *distinct-prop P xs* ⟷ *distinct-prop Q xs*
⟨*proof*⟩

lemma *distinct-prop-impl*:
⟦ ⋀*x y*. *x* ∈ *set xs* ⟹ *y* ∈ *set xs* ⟹ *P x y* ⟹ *Q x y*; *distinct-prop P xs* ⟧ ⟹
distinct-prop Q xs
⟨*proof*⟩

lemma *distinct-iff-distinct-prop-ne*: *distinct xs* ⟷ *distinct-prop* (≠) *xs*

⟨*proof*⟩

end

theory *WordSetup*
imports
Word-Lemmas-32-Internal
Word-Lemmas-64-Internal
Distinct-Prop
begin

lemma *distinct-prop-enum*:
⟦ ⋀*x y*. ⟦ *x* ≤ *stop*; *y* ≤ *stop*; *x* ≠ *y* ⟧ ⟹ *P x y* ⟧
⟹ *distinct-prop P* [*0* :: *'a* :: *len word .e. stop*]
⟨*proof*⟩

lemma *distinct-prop-enum-step*:
⟦ ⋀*x y*. ⟦ *x* ≤ *stop div step*; *y* ≤ *stop div step*; *x* ≠ *y* ⟧ ⟹ *P* (*x* ∗ *step*) (*y* ∗ *step*) ⟧
⟹ *distinct-prop P* [*0*, *step .e. stop*]
⟨*proof*⟩

lemmas *word-bits-def* =
Machine-Word-64-Basics.word-bits-def Machine-Word-32-Basics.word-bits-def

lemmas *word-size-def* =
Machine-Word-64-Basics.word-size-def Machine-Word-32-Basics.word-size-def

lemmas *word-bits-size* =
Machine-Word-64.word-bits-size Machine-Word-32.word-bits-size

lemmas *word-bits-len-of* =
Machine-Word-64.word-bits-len-of Machine-Word-32.word-bits-len-of

lemmas *word-bits-conv* =
Machine-Word-64-Basics.word-bits-conv Machine-Word-32-Basics.word-bits-conv

hide-const (**open**) *Machine-Word-32-Basics.word-bits*
hide-const (**open**) *Machine-Word-64-Basics.word-bits*
hide-const (**open**) *Machine-Word-32-Basics.word-size*
hide-const (**open**) *Machine-Word-64-Basics.word-size*
hide-const (**open**) *Machine-Word-32-Basics.word-size-bits*
hide-const (**open**) *Machine-Word-64-Basics.word-size-bits*

end

theory *Addr-Type-ARM*
imports
Target-Architecture
WordSetup
begin

if-architecture-context (*ARM*)
begin
type-synonym *addr-bitsize* = *32*

definition *addr-bitsize* :: *nat* **where** *addr-bitsize* ≡ *32*
definition *addr-align* :: *nat* **where** *addr-align* ≡ *2*

abbreviation (*input*) *array-outer-max-size-exponent* ≡ *19*::*nat*
abbreviation (*input*) *array-outer-max-count-exponent* ≡ *13*::*nat*
abbreviation (*input*) *array-inner-max-size-exponent* ≡ *6*::*nat*

abbreviation *word-bits* ≡ *Machine-Word-32-Basics.word-bits*
abbreviation *word-size* ≡ *Machine-Word-32-Basics.word-size*
abbreviation *word-size-bits* ≡ *Machine-Word-32-Basics.word-size-bits*
type-synonym *char-c* = *8 word*
end

end

theory *Addr-Type-ARM64*
imports
Target-Architecture
WordSetup
begin

if-architecture-context (*ARM64*)
begin
type-synonym *addr-bitsize* = *64*

definition *addr-bitsize* :: *nat* **where** *addr-bitsize* ≡ *64*
definition *addr-align* :: *nat* **where** *addr-align* ≡ *3*

abbreviation (*input*) *array-outer-max-size-exponent* ≡ *26*::*nat*
abbreviation (*input*) *array-outer-max-count-exponent* ≡ *20*::*nat*
abbreviation (*input*) *array-inner-max-size-exponent* ≡ *6*::*nat*

abbreviation *word-bits* ≡ *Machine-Word-64-Basics.word-bits*
abbreviation *word-size* ≡ *Machine-Word-64-Basics.word-size*
abbreviation *word-size-bits* ≡ *Machine-Word-64-Basics.word-size-bits*
type-synonym *char-c* = *8 signed word*
end

end

theory *Addr-Type-ARM-HYP*
imports
Target-Architecture
WordSetup
begin

if-architecture-context (*ARM-HYP*)
begin
type-synonym *addr-bitsize* = *32*

definition *addr-bitsize* :: *nat* **where** *addr-bitsize* ≡ *32*
definition *addr-align* :: *nat* **where** *addr-align* ≡ *2*

abbreviation (*input*) *array-outer-max-size-exponent* ≡ *19*::*nat*
abbreviation (*input*) *array-outer-max-count-exponent* ≡ *13*::*nat*
abbreviation (*input*) *array-inner-max-size-exponent* ≡ *6*::*nat*

abbreviation *word-bits* ≡ *Machine-Word-32-Basics.word-bits*
abbreviation *word-size* ≡ *Machine-Word-32-Basics.word-size*
abbreviation *word-size-bits* ≡ *Machine-Word-32-Basics.word-size-bits*
type-synonym *char-c* = *8 word*
end

end

theory *Addr-Type-RISCV64*
imports
Target-Architecture
WordSetup
begin

if-architecture-context (*RISCV64*)
begin
type-synonym *addr-bitsize* = *64*

definition *addr-bitsize* :: *nat* **where** *addr-bitsize* ≡ *64*
definition *addr-align* :: *nat* **where** *addr-align* ≡ *3*

abbreviation (*input*) *array-outer-max-size-exponent* ≡ *26*::*nat*
abbreviation (*input*) *array-outer-max-count-exponent* ≡ *20*::*nat*
abbreviation (*input*) *array-inner-max-size-exponent* ≡ *6*::*nat*

abbreviation *word-bits* ≡ *Machine-Word-64-Basics.word-bits*
abbreviation *word-size* ≡ *Machine-Word-64-Basics.word-size*
abbreviation *word-size-bits* ≡ *Machine-Word-64-Basics.word-size-bits*
type-synonym *char-c* = *8 word*
end

end

theory *Addr-Type-X64*
  imports
    Target-Architecture
    WordSetup
begin

if-architecture-context (*X64*)
begin

type-synonym *addr-bitsize* = *64*

definition *addr-bitsize* :: *nat* **where** *addr-bitsize* ≡ *64*
definition *addr-align* :: *nat* **where** *addr-align* ≡ *3*

abbreviation (*input*) *array-outer-max-size-exponent* ≡ *26*::*nat*
abbreviation (*input*) *array-outer-max-count-exponent* ≡ *20*::*nat*
abbreviation (*input*) *array-inner-max-size-exponent* ≡ *6*::*nat*

abbreviation *word-bits* ≡ *Machine-Word-64-Basics.word-bits*
abbreviation *word-size* ≡ *Machine-Word-64-Basics.word-size*
abbreviation *word-size-bits* ≡ *Machine-Word-64-Basics.word-size-bits*
type-synonym *char-c* = *8 word*
end

end

theory *Addr-Type*
  imports
    ARM/Addr-Type-ARM
    ARM64/Addr-Type-ARM64

ARM-HYP/Addr-Type-ARM-HYP
RISCV64/Addr-Type-RISCV64
X64/Addr-Type-X64
begin

typ *addr-bitsize*
term *array-outer-max-size-exponent*
type-synonym *addr* = *addr-bitsize word*

declare *addr-align-def*[*simp*]
declare *addr-bitsize-def*[*simp*]

definition *addr-card* :: *nat* **where**
addr-card ≡ *card* (*UNIV*::*addr set*)

lemma *addr-card*:
addr-card = *2*^*addr-bitsize*
⟨*proof*⟩

lemma *len-of-addr-card*:
2 ^ *len-of TYPE*(*addr-bitsize*) = *addr-card*
⟨*proof*⟩

lemma *of-nat-addr-card* [*simp*]:
of-nat addr-card = (*0*::*addr*)
⟨*proof*⟩

end

theory *CTypesBase*
imports
Addr-Type
begin

11.3 Type setup

type-synonym *byte* = *8 word*

type-synonym *memory* = *addr* ⇒ *byte*
type-synonym *'a mem-upd* = *addr* ⇒ *'a* ⇒ *memory* ⇒ *memory*
type-synonym *'a mem-read* = *addr* ⇒ *memory* ⇒ *'a*

class *unit-class* =
assumes *there-is-only-one*: *x* = *y*

instantiation *unit* :: *unit-class*
begin
instance ⟨*proof*⟩
end

11.3.1 Pointers

datatype *'a ptr* = *Ptr addr*

abbreviation
NULL :: *'a ptr* **where**
NULL ≡ *Ptr 0*

⟨*ML*⟩

syntax
-Ptr :: *type* ⇒ *logic* (‹(‹*indent=1 notation=‹mixfix PTR››PTR*/(*1 '*(*-'*)))›)
syntax-consts
-Ptr == *Ptr*
translations
PTR('*a*) => *CONST Ptr* :: (*addr* ⇒ *'a ptr*)
⟨*ML*⟩

primrec
ptr-val :: *'a ptr* ⇒ *addr*
where
ptr-val-def: *ptr-val* (*Ptr a*) = *a*

primrec
ptr-coerce :: *'a ptr* ⇒ *'b ptr* **where**
ptr-coerce (*Ptr a*) = *Ptr a*

syntax
-Ptr-coerce :: *type* ⇒ *type* ⇒ *logic*
(‹(‹*indent=1 notation=‹mixfix PTR-COERCE››PTR'-COERCE*/(‹*indent=1 notation=‹infix coerce›› '*(- → -*'*)))›)
syntax-consts
-Ptr-coerce == *ptr-coerce*
translations
PTR-COERCE('*a* → *'b*) => *CONST ptr-coerce* :: (*'a ptr* ⇒ *'b ptr*)
⟨*ML*⟩

definition

ptr-less :: *'a ptr* ⇒ *'a ptr* ⇒ *bool* (**infixl** ‹<$_p$› *50*) **where**
p <$_p$ *q* ≡ *ptr-val p* < *ptr-val q*

definition
ptr-le :: *'a ptr* ⇒ *'a ptr* ⇒ *bool* (**infixl** ‹≤$_p$› *50*) **where**

$p \leq_p q \equiv$ *ptr-val* $p \leq$ *ptr-val* q

instantiation *ptr* :: (*type*) *ord*
begin

definition
ptr-less-def': $p < q \equiv p <_p q$
definition
ptr-le-def': $p \leq q \equiv p \leq_p q$

instance ⟨*proof*⟩

end

lemma *ptr-val-case*: *ptr-val* p = (*case* p *of Ptr* $v \Rightarrow v$)
⟨*proof*⟩

instantiation *ptr* :: (*type*) *linorder*
begin
instance
⟨*proof*⟩
end

11.3.2 Raw heap

A raw map from addresses to bytes

type-synonym *heap-mem* = *addr* ⇒ *byte*

For heap h, pointer p and nat n, *heap-list* h n p returns the list of bytes in the heap taken from addresses $\{p..+n\}$

primrec
heap-list :: *heap-mem* ⇒ *nat* ⇒ *addr* ⇒ *byte list*
where
heap-list-base: *heap-list* h *0* p = []
| *heap-list-rec*: *heap-list* h (*Suc* n) p = h p # *heap-list* h n (p + *1*)

11.4 Intervals

For word a and nat b, $\{a..+b\}$ is the set of words x, with *unat* $(x - a) < b$.

definition
intvl :: 'a::*len word* × *nat* ⇒ 'a::*len word set* **where**
intvl $x \equiv \{z.\ \exists k.\ z =$ *fst* $x +$ *of-nat* $k \wedge k <$ *snd* $x\}$

abbreviation
intvl-abbr :: 'a::*len word* ⇒ *nat* ⇒ 'a *word set*
(‹(‹*open-block notation*=‹*mixfix intvl*››{-..+-})›)
where $\{a..+b\} \equiv$ *intvl* (a,b)

11.5 *dt-tuple*: a reimplementation of 3 item tuples

datatype
  $('a, 'b, 'c)$ *dt-tuple* = *DTuple* (*dt-fst*: $'a$) (*dt-snd*: $'b$) (*dt-trd*: $'c$)

lemma *dt-prj-simps*[*simp*]:
  dt-fst (*DTuple a b c*) = *a*
  dt-snd (*DTuple a b c*) = *b*
  dt-trd (*DTuple a b c*) = *c*
  ⟨*proof*⟩

lemma *split-DTuple-All*:
  $(\forall x.\ P\ x) = (\forall a\ b\ c.\ P\ (DTuple\ a\ b\ c))$
  ⟨*proof*⟩

lemma *surjective-dt-tuple*:
  p = *DTuple* (*dt-fst p*) (*dt-snd p*) (*dt-trd p*)
  ⟨*proof*⟩

lemma *split-DTuple-all*[*no-atp*]: $(\bigwedge x.\ PROP\ P\ x) \equiv (\bigwedge a\ b\ c.\ PROP\ P\ (DTuple\ a\ b\ c))$
⟨*proof*⟩

type-synonym *normalisor* = *byte list* ⇒ *byte list*

11.6 Properties of pointers

lemma *Ptr-ptr-val* [*simp*]:
  Ptr (*ptr-val p*) = *p*
  ⟨*proof*⟩

lemma *ptr-val-ptr-coerce* [*simp*]:
  ptr-val (*ptr-coerce p*) = *ptr-val p*
  ⟨*proof*⟩

lemma *Ptr-ptr-coerce* [*simp*]:
  Ptr (*ptr-val p*) = *ptr-coerce p*
  ⟨*proof*⟩

lemma *ptr-coerce-id* [*simp*]:
  ptr-coerce p = *p*
  ⟨*proof*⟩

lemma *ptr-coerce-idem* [*simp*]:
  ptr-coerce (*ptr-coerce p*) = *ptr-coerce p*
  ⟨*proof*⟩

lemma *ptr-val-inj* [*simp*]:
  (*ptr-val p* = *ptr-val q*) = (*p* = *q*)

⟨*proof*⟩

lemma *ptr-coerce-NULL* [*simp*]:
(*ptr-coerce p* = *NULL*) = (*p* = *NULL*)
⟨*proof*⟩

lemma *NULL-ptr-val*:
(*p* = *NULL*) = (*ptr-val p* = *0*)
⟨*proof*⟩

lemma *ptr-NULL-conv*: *ptr-coerce NULL* = *NULL*
⟨*proof*⟩

instantiation *ptr* :: (*type*) *finite*
begin
instance
⟨*proof*⟩
end

11.7 Properties of the raw heap

lemma *heap-list-length* [*simp*]:
length (*heap-list h n p*) = *n*
⟨*proof*⟩

lemma *heap-list-split*:
shows *k* ≤ *n* ⟹ *heap-list h n x* = *heap-list h k x* @ *heap-list h* (*n* − *k*) (*x* + *of-nat k*)
⟨*proof*⟩

lemma *heap-list-split2*:
heap-list h (*x* + *y*) *p* = *heap-list h x p* @ *heap-list h y* (*p* + *of-nat x*)
⟨*proof*⟩

11.8 Properties of intervals

lemma *intvlI*:
x < *n* ⟹ *p* + *of-nat x* ∈ {*p*..+*n*}
⟨*proof*⟩

lemma *intvlD*:
q ∈ {*p*..+*n*} ⟹ ∃ *k*. *q* = *p* + *of-nat k* ∧ *k* < *n*
⟨*proof*⟩

lemma *intvl-empty* [*simp*]:
{*p*..+*0*} = {}
⟨*proof*⟩

lemma *intvl-Suc*:
$q \in \{p..+Suc\ 0\} \Longrightarrow p = q$
⟨*proof*⟩

lemma *intvl-self*:
$0 < n \Longrightarrow x \in \{x..+n\}$
⟨*proof*⟩

lemma *intvl-start-inter*:
$[\![\ 0 < m;\ 0 < n\ ]\!] \Longrightarrow \{p..+m\} \cap \{p..+n\} \neq \{\}$
⟨*proof*⟩

lemma *intvl-overflow*:
assumes *2^len-of TYPE('a)* $\leq n$
shows $\{(p::'a::len\ word)..+n\} = UNIV$
⟨*proof*⟩

lemma *intvl-self-offset*:
fixes *p::'a::len word*
assumes *a*: *2^len-of TYPE('a)* $- n < x$ **and** *b*: $x <$ *2^len-of TYPE('a)* **and**
c: $(p::'a::len\ word) \notin \{p +$ *of-nat x..+n*$\}$
shows *False*
⟨*proof*⟩

lemma *intvl-mem-offset*:
$[\![\ q \in \{p..+unat\ x\};\ q \notin \{p..+unat\ y\};\ unat\ y \leq unat\ x\ ]\!] \Longrightarrow$
$q \in \{p + y..+unat\ x - unat\ y\}$
⟨*proof*⟩

lemma *intvl-plus-sub-offset*:
$x \in \{p + y..+q - unat\ y\} \Longrightarrow x \in \{p..+q\}$
⟨*proof*⟩

lemma *intvl-plus-sub-Suc*:
$x \in \{p + 1..+q - Suc\ 0\} \Longrightarrow x \in \{p..+q\}$
⟨*proof*⟩

lemma *intvl-neq-start*:
$[\![\ (q::'a::len\ word) \in \{p..+n\};\ p \neq q\ ]\!] \Longrightarrow q \in \{p + 1..+n - Suc\ 0\}$
⟨*proof*⟩

lemmas *unat-simps'* =
word-arith-nat-defs word-unat.eq-norm len-of-addr-card mod-less

lemma *intvl-offset-nmem*:
$[\![\ q \in \{(p::'a::len\ word)..+unat\ x\};\ y \leq$ *2^len-of TYPE('a)* $- unat\ x\ ]\!] \Longrightarrow$
$q \notin \{p + x..+y\}$
⟨*proof*⟩

lemma *intvl-Suc-nmem'* [*simp*]:
$n < 2\hat{}$*len-of TYPE*('*a*) $\Longrightarrow$ (*p*::'*a*::*len word*) $\notin$ {*p* + *1*..+*n* − *Suc 0*}
⟨*proof*⟩

lemma *intvl-Suc-nmem''*:
$n \leq 2\hat{}$*len-of TYPE*('*a*) $\Longrightarrow$ (*p*::'*a*::*len word*) $\notin$ {*p* + *1*..+*n* − *Suc 0*}
⟨*proof*⟩

lemma *intvl-start-le*:
$x \leq y \Longrightarrow \{p..+x\} \subseteq \{p..+y\}$
⟨*proof*⟩

lemma *intvl-sub-eq*:
assumes $x \leq y$
shows {*p* + *x*..+*unat* (*y* − *x*)} = {*p*..+*unat y*} − {*p*..+*unat x*}
⟨*proof*⟩

lemma *intvl-disj-offset*:
{*x* + *a*..+*c*} ∩ {*x* + *b*..+*d*} = {} = ({*a*..+*c*} ∩ {*b*..+*d*} = {})
⟨*proof*⟩

lemma *intvl-sub-offset*:
unat x+*y* ≤ *z* ⟹ {*k*+*x*..+*y*} ⊆ {*k*..+*z*}
⟨*proof*⟩

lemma *disjnt-intvl-offsetp*[*simp*]:
disjnt {*a* + *x* ..+ *n*} {*a* + *y* ..+ *m*} ⟷ *disjnt* {*x* ..+ *n*} {*y* ..+ *m*}
⟨*proof*⟩

lemma *intvl-eq-of-nat-Ico-add*: {*of-nat n*::'*a*::*len word*..+ *m*} = *of-nat* ‘ {*n* ..< *n* + *m*}
⟨*proof*⟩

lemma *intvl-le*:
assumes *n2* + *off1* ≤ *n1*
shows {*p* + *of-nat off1*..+*n2*} ⊆ {*p*..+*n1*}
⟨*proof*⟩

lemma *intvl-disj-left*:
fixes *a b* :: *addr*
assumes *a-n*: *unat a* + *n* ≤ *unat b* **and** *b-m*: *unat b* + *m* ≤ *addr-card* + *unat a*
shows {*a* ..+ *n*} ∩ {*b* ..+ *m*} = {}
⟨*proof*⟩

end

theory *CTypesDefs*
imports
CTypesBase
begin

11.9 C types setup

type-synonym *field-name* = *string*
type-synonym *qualified-field-name* = *field-name list*

type-synonym *typ-name* = *string*

A *typ-desc* wraps a *typ-struct* with a typ name. A *typ-struct* is either a Scalar, with size, alignment and either a field description (for *typ-info*) or a 'normalisor' (for *typ-uinfo*), or an Aggregate, with a list of *typ-desc*, field name, and field descripton (for the complete sub-structure) or unit (for *typ-uinfo*). The field description for aggregates is an extension of the original work of H. Tuch. It is used to make the construction of a new structure from nested structures / arrays more efficient. Properties like commutation of fields can be expressed and proven for the toplevel fields only, without having to re-examine the nested leafs of the tree.

datatype
('*a*,'*b*) *typ-desc* = *TypDesc nat* ('*a*, '*b*) *typ-struct typ-name*
and ('*a*,'*b*) *typ-struct* = *TypScalar nat nat* '*a* |
TypAggregate (('*a*, '*b*) *typ-desc*, *field-name*, '*b*) *dt-tuple list*

datatype-compat *dt-tuple*
datatype-compat *typ-desc typ-struct*
print-theorems

lemma *typ-desc-induct*:
⟦⋀*nat typ-struct list*. *P2 typ-struct* ⟹ *P1* (*TypDesc nat typ-struct list*); ⋀*nat1 nat2 a*. *P2* (*TypScalar nat1 nat2 a*);
⋀*list*. *P3 list* ⟹ *P2* (*TypAggregate list*); *P3* []; ⋀*dt-tuple list*. ⟦*P4 dt-tuple*; *P3 list*⟧ ⟹ *P3* (*dt-tuple* # *list*);
⋀*typ-desc list b*. *P1 typ-desc* ⟹ *P4* (*DTuple typ-desc list b*)⟧
⟹ *P1 typ-desc*
⟨*proof*⟩

lemma *typ-struct-induct*:
⟦⋀*nat typ-struct list*. *P2 typ-struct* ⟹ *P1* (*TypDesc nat typ-struct list*); ⋀*nat1 nat2 a*. *P2* (*TypScalar nat1 nat2 a*);
⋀*list*. *P3 list* ⟹ *P2* (*TypAggregate list*); *P3* []; ⋀*dt-tuple list*. ⟦*P4 dt-tuple*; *P3 list*⟧ ⟹ *P3* (*dt-tuple* # *list*);

$\bigwedge$*typ-desc list b. P1 typ-desc* $\Longrightarrow$ *P4* (*DTuple typ-desc list b*)$\rrbracket$
$\Longrightarrow$ *P2 typ-struct*
⟨*proof*⟩

lemma *typ-list-induct*:
$\llbracket\bigwedge$*nat typ-struct list. P2 typ-struct* $\Longrightarrow$ *P1* (*TypDesc nat typ-struct list*); $\bigwedge$*nat1 nat2 a. P2* (*TypScalar nat1 nat2 a*);
$\bigwedge$*list. P3 list* $\Longrightarrow$ *P2* (*TypAggregate list*); *P3* []; $\bigwedge$*dt-tuple list.* $\llbracket$*P4 dt-tuple*; *P3 list*$\rrbracket$ $\Longrightarrow$ *P3* (*dt-tuple # list*);
$\bigwedge$*typ-desc list b. P1 typ-desc* $\Longrightarrow$ *P4* (*DTuple typ-desc list b*)$\rrbracket$
$\Longrightarrow$ *P3 list*
⟨*proof*⟩

lemma *typ-dt-tuple-induct*:
$\llbracket\bigwedge$*nat typ-struct list. P2 typ-struct* $\Longrightarrow$ *P1* (*TypDesc nat typ-struct list*); $\bigwedge$*nat1 nat2 a. P2* (*TypScalar nat1 nat2 a*);
$\bigwedge$*list. P3 list* $\Longrightarrow$ *P2* (*TypAggregate list*); *P3* []; $\bigwedge$*dt-tuple list.* $\llbracket$*P4 dt-tuple*; *P3 list*$\rrbracket$ $\Longrightarrow$ *P3* (*dt-tuple # list*);
$\bigwedge$*typ-desc list b. P1 typ-desc* $\Longrightarrow$ *P4* (*DTuple typ-desc list b*)$\rrbracket$
$\Longrightarrow$ *P4 dt-tuple*
⟨*proof*⟩
lemmas *typ-desc-typ-struct-inducts* [*case-names*
TypDesc TypScalar TypAggregate Nil-typ-desc Cons-typ-desc DTuple-typ-desc, *induct type*] =
typ-desc-induct typ-struct-induct typ-list-induct typ-dt-tuple-induct

— Make sure list induct rule is tried first
declare *list.induct* [*induct type*]

type-synonym (*'a*, *'b*) *typ-tuple* = ((*'a*, *'b*) *typ-desc*,*field-name*, *'b*) *dt-tuple*

type-synonym *typ-uinfo* = (*normalisor*, *unit*) *typ-desc*
type-synonym *typ-uinfo-struct* = (*normalisor*, *unit*) *typ-struct*
type-synonym *typ-uinfo-tuple* = (*normalisor*, *unit*) *typ-tuple*

record *'a field-desc* =
field-access :: *'a* ⇒ *byte list* ⇒ *byte list*
field-update :: *byte list* ⇒ *'a* ⇒ *'a*
field-sz :: *nat*

type-synonym (*'a*, *'b*) *typ-info* = (*'a field-desc*, *'b*) *typ-desc*
type-synonym (*'a*, *'b*) *typ-info-struct* = (*'a field-desc*, *'b*) *typ-struct*
type-synonym (*'a*, *'b*) *typ-info-tuple* = (*'a field-desc*, *'b*) *typ-tuple*

type-synonym *'a xtyp-tuple* = (*'a*, *'a*) *typ-tuple*
type-synonym *'a xtyp-info* = (*'a field-desc*, *'a field-desc*) *typ-desc*
type-synonym *'a xtyp-info-struct* = (*'a field-desc*, *'a field-desc*) *typ-struct*
type-synonym *'a xtyp-info-tuple* = *'a field-desc xtyp-tuple*

definition *fu-commutes* :: $('b \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('c \Rightarrow 'a \Rightarrow 'a) \Rightarrow bool$ **where**
fu-commutes f g $\equiv \forall v\ bs\ bs'.\ f\ bs\ (g\ bs'\ v) = g\ bs'\ (f\ bs\ v)$

size-td returns the sum of the sizes of all Scalar fields comprising a *typ-desc* i.e. the overall size of the type

primrec
size-td :: $('a, 'b)$ *typ-desc* $\Rightarrow$ *nat* **and**
size-td-struct :: $('a, 'b)$ *typ-struct* $\Rightarrow$ *nat* **and**
size-td-list :: $('a, 'b)$ *typ-tuple list* $\Rightarrow$ *nat* **and**
size-td-tuple :: $('a, 'b)$ *typ-tuple* $\Rightarrow$ *nat*
where
tz0: *size-td* (*TypDesc algn st nm*) = *size-td-struct st*

| *tz1*: *size-td-struct* (*TypScalar n algn d*) = *n*
| *tz2*: *size-td-struct* (*TypAggregate xs*) = *size-td-list xs*

| *tz3*: *size-td-list* [] = *0*
| *tz4*: *size-td-list* (*x*#*xs*) = *size-td-tuple x* + *size-td-list xs*

| *tz5*: *size-td-tuple* (*DTuple t n d*) = *size-td t*

access-ti overlays the byte-wise representation of an object on a given byte list, given the *typ-info* (i.e. the layout)

primrec
access-ti :: $('a, 'b)$ *typ-info* $\Rightarrow$ ($'a \Rightarrow$ *byte list* $\Rightarrow$ *byte list*) **and**
access-ti-struct :: $('a, 'b)$ *typ-info-struct* $\Rightarrow$
($'a \Rightarrow$ *byte list* $\Rightarrow$ *byte list*) **and**
access-ti-list :: $('a, 'b)$ *typ-info-tuple list* $\Rightarrow$
($'a \Rightarrow$ *byte list* $\Rightarrow$ *byte list*) **and**
access-ti-tuple :: $('a, 'b)$ *typ-info-tuple* $\Rightarrow$ ($'a \Rightarrow$ *byte list* $\Rightarrow$ *byte list*)
where
fa0: *access-ti* (*TypDesc algn st nm*) = *access-ti-struct st*

| *fa1*: *access-ti-struct* (*TypScalar n algn d*) = *field-access d*
| *fa2*: *access-ti-struct* (*TypAggregate xs*) = *access-ti-list xs*

| *fa3*: *access-ti-list* [] = ($\lambda v\ bs.$ [])
| *fa4*: *access-ti-list* (*x*#*xs*) =
($\lambda v\ bs.$ *access-ti-tuple x v* (*take* (*size-td-tuple x*) *bs*) @
access-ti-list xs v (*drop* (*size-td-tuple x*) *bs*))

| *fa5*: *access-ti-tuple* (*DTuple t nm d*) = *access-ti t*

access-ti$_0$ overlays the representation of an object on a list of zero bytes

definition *access-ti*$_0$ :: $('a, 'b)$ *typ-info* $\Rightarrow$ ($'a \Rightarrow$ *byte list*) **where**
access-ti$_0$ *t* $\equiv \lambda v.$ *access-ti t v* (*replicate* (*size-td t*) *0*)

update-ti updates an object, given a list of bytes (the representation of the new value), and the *typ-info*

primrec
update-ti :: ('*a*, '*b*) *typ-info* ⇒ (*byte list* ⇒ '*a* ⇒ '*a*) **and**
update-ti-struct :: ('*a*, '*b*) *typ-info-struct* ⇒ (*byte list* ⇒ '*a* ⇒ '*a*) **and**
update-ti-list :: ('*a*, '*b*) *typ-info-tuple list* ⇒ (*byte list* ⇒ '*a* ⇒ '*a*) **and**
update-ti-tuple :: ('*a*, '*b*) *typ-info-tuple* ⇒ (*byte list* ⇒ '*a* ⇒ '*a*)
where
fu0: *update-ti* (*TypDesc algn st nm*) = *update-ti-struct st*

| *fu1*: *update-ti-struct* (*TypScalar n algn d*) = *field-update d*
| *fu2*: *update-ti-struct* (*TypAggregate xs*) = *update-ti-list xs*

| *fu3*: *update-ti-list* [] = (λ*bs*. *id*)
| *fu4*: *update-ti-list* (*x*#*xs*) = (λ*bs v*.
update-ti-tuple x (*take* (*size-td-tuple x*) *bs*)
(*update-ti-list xs* (*drop* (*size-td-tuple x*) *bs*) *v*))

| *fu5*: *update-ti-tuple* (*DTuple t nm d*) = *update-ti t*

update-ti-t updates an object only if the length of the supplied representation equals the object size

definition *update-ti-t* :: ('*a*, '*b*) *typ-info* ⇒ (*byte list* ⇒ '*a* ⇒ '*a*) **where**
update-ti-t t ≡ λ*bs*. *if length bs* = *size-td t then*
update-ti t bs else id

definition *update-ti-struct-t* :: ('*a*, '*b*) *typ-info-struct* ⇒ (*byte list* ⇒ '*a* ⇒ '*a*)
where
update-ti-struct-t t ≡ λ*bs*. *if length bs* = *size-td-struct t then*
update-ti-struct t bs else id

definition *update-ti-list-t* :: ('*a*, '*b*) *typ-info-tuple list* ⇒ (*byte list* ⇒ '*a* ⇒ '*a*)
where
update-ti-list-t t ≡ λ*bs*. *if length bs* = *size-td-list t then*
update-ti-list t bs else id

definition *update-ti-tuple-t* :: ('*a*, '*b*) *typ-info-tuple* ⇒ (*byte list* ⇒ '*a* ⇒ '*a*) **where**
update-ti-tuple-t t ≡ λ*bs*. *if length bs* = *size-td-tuple t then*
update-ti-tuple t bs else id

lemma *update-ti-t-struct-t* [*simp*]: *update-ti-t* (*TypDesc algn st nm*) = *update-ti-struct-t st*
⟨*proof*⟩

lemma *update-ti-update-ti-t*:
length bs = *size-td s* ⟹ *update-ti s bs v* = *update-ti-t s bs v*
⟨*proof*⟩

field-desc generates the access/update pair for a field, given the field's *type-desc*

definition *field-desc* :: ('*a*, '*b*) *typ-info* ⇒ '*a field-desc* **where**

field-desc t ≡ (| *field-access* = *access-ti t*,
field-update = *update-ti-t t*, *field-sz* = *size-td t* |)

declare *field-desc-def* [*simp add*]

definition *field-desc-struct* :: ('*a*, '*b*) *typ-info-struct* ⇒ '*a field-desc* **where**
field-desc-struct t ≡ (| *field-access* = *access-ti-struct t*,
field-update = *update-ti-struct-t t*, *field-sz* = *size-td-struct t* |)

declare *field-desc-struct-def* [*simp add*]

definition *field-desc-list* :: ('*a*, '*b*) *typ-info-tuple list* ⇒ '*a field-desc*
where
field-desc-list t ≡ (| *field-access* = *access-ti-list t*,
field-update = *update-ti-list-t t*, *field-sz* = *size-td-list t* |)

declare *field-desc-list-def* [*simp add*]

definition *field-desc-tuple* :: ('*a*, '*b*) *typ-info-tuple* ⇒ '*a field-desc*
where
field-desc-tuple t ≡ (| *field-access* = *access-ti-tuple t*,
field-update = *update-ti-tuple-t t*, *field-sz* = *size-td-tuple t* |)

declare *field-desc-tuple-def* [*simp add*]

primrec
map-td :: (*nat* ⇒ *nat* ⇒ '*a* ⇒ '*b*) ⇒ ('*c* ⇒ '*d*) ⇒ ('*a*, '*c*) *typ-desc* ⇒ ('*b*, '*d*) *typ-desc* **and**
map-td-struct :: (*nat* ⇒ *nat* ⇒ '*a* ⇒ '*b*) ⇒ ('*c* ⇒ '*d*) ⇒ ('*a*, '*c*) *typ-struct* ⇒ ('*b*, '*d*) *typ-struct* **and**
map-td-list :: (*nat* ⇒ *nat* ⇒ '*a* ⇒ '*b*) ⇒ ('*c* ⇒ '*d*) ⇒ ('*a*, '*c*) *typ-tuple list* ⇒ ('*b*, '*d*) *typ-tuple list* **and**
map-td-tuple :: (*nat* ⇒ *nat* ⇒ '*a* ⇒ '*b*) ⇒ ('*c* ⇒ '*d*) ⇒ ('*a*, '*c*) *typ-tuple* ⇒ ('*b*, '*d*) *typ-tuple*
where
mat0: *map-td f g* (*TypDesc algn st nm*) = *TypDesc algn* (*map-td-struct f g st*) *nm*

| *mat1*: *map-td-struct f g* (*TypScalar n algn d*) = *TypScalar n algn* (*f n algn d*)
| *mat2*: *map-td-struct f g* (*TypAggregate xs*) = *TypAggregate* (*map-td-list f g xs*)

| *mat3*: *map-td-list f g* [] = []
| *mat4*: *map-td-list f g* (*x*#*xs*) = *map-td-tuple f g x* # *map-td-list f g xs*

| *mat5*: *map-td-tuple f g* (*DTuple t n d*) = *DTuple* (*map-td f g t*) *n* (*g d*)

definition *field-norm* :: *nat* ⇒ *nat* ⇒ ('*a*,'*b*) *field-desc-scheme* ⇒ (*byte list* ⇒ *byte list*)

where
field-norm ≡ λ*n algn d bs*.
if length bs = *n then*
field-access d (*field-update d bs undefined*) (*replicate n 0*) *else*
[]

definition *export-uinfo* :: (′*a*, ′*b*) *typ-info* ⇒ *typ-uinfo* **where**
export-uinfo t ≡ *map-td field-norm* (λ-. ()) *t*

primrec
wf-desc :: (′*a*, ′*b*) *typ-desc* ⇒ *bool* **and**
wf-desc-struct :: (′*a*, ′*b*) *typ-struct* ⇒ *bool* **and**
wf-desc-list :: (′*a*, ′*b*) *typ-tuple list* ⇒ *bool* **and**
wf-desc-tuple :: (′*a*, ′*b*) *typ-tuple* ⇒ *bool*
where
wfd0: *wf-desc* (*TypDesc algn ts n*) = *wf-desc-struct ts*

| *wfd1*: *wf-desc-struct* (*TypScalar n algn d*) = *True*
| *wfd2*: *wf-desc-struct* (*TypAggregate ts*) = *wf-desc-list ts*

| *wfd3*: *wf-desc-list* [] = *True*
| *wfd4*: *wf-desc-list* (*x*#*xs*) = (*wf-desc-tuple x* ∧ ¬ *dt-snd x* ∈ *dt-snd* ‘ *set xs* ∧
wf-desc-list xs)

| *wfd5*: *wf-desc-tuple* (*DTuple x n d*) = *wf-desc x*

primrec
wf-size-desc :: (′*a*, ′*b*) *typ-desc* ⇒ *bool* **and**
wf-size-desc-struct :: (′*a*, ′*b*) *typ-struct* ⇒ *bool* **and**
wf-size-desc-list :: (′*a*, ′*b*) *typ-tuple list* ⇒ *bool* **and**
wf-size-desc-tuple :: (′*a*, ′*b*) *typ-tuple* ⇒ *bool*
where
wfsd0: *wf-size-desc* (*TypDesc algn ts n*) = *wf-size-desc-struct ts*

| *wfsd1*: *wf-size-desc-struct* (*TypScalar n algn d*) = (*0* < *n*)
| *wfsd2*: *wf-size-desc-struct* (*TypAggregate ts*) =
(*ts* ≠ [] ∧ *wf-size-desc-list ts*)

| *wfsd3*: *wf-size-desc-list* [] = *True*
| *wfsd4*: *wf-size-desc-list* (*x*#*xs*) =
(*wf-size-desc-tuple x* ∧ *wf-size-desc-list xs*)

| *wfsd5*: *wf-size-desc-tuple* (*DTuple x n d*) = *wf-size-desc x*

definition
typ-struct :: (′*a*, ′*b*) *typ-desc* ⇒ (′*a*, ′*b*) *typ-struct*

where
typ-struct t = (*case t of TypDesc algn st sz* ⇒ *st*)

lemma *typ-struct* [*simp*]:
typ-struct (*TypDesc algn st sz*) = *st*
⟨*proof*⟩

primrec
typ-name :: ('*a*, '*b*) *typ-desc* ⇒ *typ-name*
where
typ-name (*TypDesc algn st nm*) = *nm*

primrec
norm-tu :: *typ-uinfo* ⇒ *normalisor* **and**
norm-tu-struct :: *typ-uinfo-struct* ⇒ *normalisor* **and**
norm-tu-list :: *typ-uinfo-tuple list* ⇒ *normalisor* **and**
norm-tu-tuple :: *typ-uinfo-tuple* ⇒ *normalisor*
where
tn0: *norm-tu* (*TypDesc algn st nm*) = *norm-tu-struct st*

| *tn1*: *norm-tu-struct* (*TypScalar n aln f*) = *f*
| *tn2*: *norm-tu-struct* (*TypAggregate xs*) = *norm-tu-list xs*

| *tn3*: *norm-tu-list* [] = (λ*bs*. [])
| *tn4*: *norm-tu-list* (*x*#*xs*) = (λ*bs*.
norm-tu-tuple x (*take* (*size-td-tuple x*) *bs*) @
norm-tu-list xs (*drop* (*size-td-tuple x*) *bs*))

| *tn5*: *norm-tu-tuple* (*DTuple t n d*) = *norm-tu t*

class *c-type-name* =
fixes
typ-name-itself :: '*a itself* ⇒ *typ-name*

class *c-type* = *c-type-name* +
fixes
typ-info-t :: '*a itself* ⇒ '*a xtyp-info*
assumes *type-name-itself-typ-name*: *typ-name-itself T* = *typ-name* (*typ-info-t T*)

instance *c-type* ⊆ *type* ⟨*proof*⟩

definition (**in** *c-type*) *typ-uinfo-t* :: '*a itself* ⇒ *typ-uinfo* **where**
typ-uinfo-t t ≡ *export-uinfo* (*typ-info-t TYPE*('*a*))

definition (**in** *c-type*) *to-bytes* :: '*a* ⇒ *byte list* ⇒ *byte list* **where**
to-bytes v ≡ *access-ti* (*typ-info-t TYPE*('*a*)) *v*

definition (**in** *c-type*) *from-bytes* :: *byte list* ⇒ *'a* **where**
from-bytes bs ≡
field-update (*field-desc* (*typ-info-t TYPE*('*a*))) *bs undefined*

type-synonym ('*a*, '*b*) *flr* = (('*a*, '*b*) *typ-desc* × *nat*) *option*

primrec
field-lookup :: ('*a*, '*b*) *typ-desc* ⇒ *qualified-field-name* ⇒ *nat* ⇒ ('*a*, '*b*) *flr* **and**
field-lookup-struct :: ('*a*, '*b*) *typ-struct* ⇒ *qualified-field-name* ⇒ *nat* ⇒
('*a*, '*b*) *flr* **and**
field-lookup-list :: ('*a*, '*b*) *typ-tuple list* ⇒ *qualified-field-name* ⇒ *nat* ⇒
('*a*, '*b*) *flr* **and**
field-lookup-tuple :: ('*a*, '*b*) *typ-tuple* ⇒ *qualified-field-name* ⇒ *nat* ⇒ ('*a*, '*b*) *flr*
where
fl0: *field-lookup* (*TypDesc algn st nm*) *f m* =
(*if f*=[] *then Some* (*TypDesc algn st nm*,*m*) *else field-lookup-struct st f m*)

| *fl1*: *field-lookup-struct* (*TypScalar n algn d*) *f m* = *None*
| *fl2*: *field-lookup-struct* (*TypAggregate xs*) *f m* = *field-lookup-list xs f m*

| *fl3*: *field-lookup-list* [] *f m* = *None*
| *fl4*: *field-lookup-list* (*x*#*xs*) *f m* = (
case field-lookup-tuple x f m of
None ⇒ *field-lookup-list xs f* (*m* + *size-td* (*dt-fst x*)) |
Some y ⇒ *Some y*)

| *fl5*: *field-lookup-tuple* (*DTuple t nm d*) *f m* =
(*if nm*=*hd f* ∧ *f* ≠ [] *then field-lookup t* (*tl f*) *m else None*)

lemma *field-lookup-wf-desc-pres*:
fixes *t*::('*a*, '*b*) *typ-desc*
and *st*::('*a*, '*b*) *typ-struct*
and *ts*::('*a*, '*b*) *typ-tuple list*
and *x*::('*a*, '*b*) *typ-tuple*
shows
wf-desc t ⟹ *field-lookup t f n* = *Some* (*s*, *m*) ⟹ *wf-desc s*
wf-desc-struct st ⟹ *field-lookup-struct st f n* = *Some* (*s*, *m*) ⟹ *wf-desc s*
wf-desc-list ts ⟹ *field-lookup-list ts f n* = *Some* (*s*, *m*) ⟹ *wf-desc s*
wf-desc-tuple x ⟹ *field-lookup-tuple x f n* = *Some* (*s*, *m*) ⟹ *wf-desc s*
⟨*proof*⟩

definition *map-td-flr* :: (*nat* ⇒ *nat* ⇒ *'a* ⇒ *'b*) ⇒ (*'c* ⇒ *'d*) ⇒
(('*a*,'*c*) *typ-desc* × *nat*) *option* ⇒ ('*b*, '*d*) *flr*
where
map-td-flr f g ≡ *case-option None* (λ(*s*,*n*). *Some* (*map-td f g s*,*n*))

definition
import-flr :: (*nat* ⇒ *nat* ⇒ *'b* ⇒ *'a*) ⇒ (*'d* ⇒ *'c*) ⇒ (*'a*, *'c*) *flr* ⇒ ((*'b*,*'d*) *typ-desc* × *nat*) *option* ⇒ *bool*
where
import-flr f g s k ≡ *case-option* (*k*=*None*)
(λ(*s*,*m*). *case-option False* (λ(*t*,*n*). *n*=*m* ∧ *map-td f g t*=*s*) *k*)
s

definition
field-offset-untyped :: (*'a*, *'b*) *typ-desc* ⇒ *qualified-field-name* ⇒ *nat*
where
field-offset-untyped t n ≡ *snd* (*the* (*field-lookup t n 0*))

definition (**in** *c-type*)
field-offset :: *'a itself* ⇒ *qualified-field-name* ⇒ *nat*
where
field-offset t n ≡ *field-offset-untyped* (*typ-uinfo-t TYPE*(*'a*)) *n*

definition (**in** *c-type*)
field-ti :: *'a itself* ⇒ *qualified-field-name* ⇀ *'a xtyp-info*
where
field-ti t n ≡ *case-option None* (*Some* ∘ *fst*)
(*field-lookup* (*typ-info-t TYPE*(*'a*)) *n 0*)

definition (**in** *c-type*)
field-size :: *'a itself* ⇒ *qualified-field-name* ⇒ *nat*
where
field-size t n ≡ *size-td* (*the* (*field-ti t n*))

definition (**in** *c-type*)
field-lvalue :: *'a ptr* ⇒ *qualified-field-name* ⇒ *addr*
(‹(‹*open-block notation*=‹*mixfix field-lvalue*››&*'*(-→-*'*))›)
where
&(*p*→*f*) ≡ *ptr-val* (*p*::*'a ptr*) + *of-nat* (*field-offset TYPE*(*'a*) *f*)

definition (**in** *c-type*)
size-of :: *'a itself* ⇒ *nat* **where**
size-of t ≡ *size-td* (*typ-info-t TYPE*(*'a*))

lemma (**in** *c-type*) *size-of-fold*: *size-td* (*typ-info-t TYPE*(*'a*)) = *size-of TYPE*(*'a*)
⟨*proof*⟩

definition (**in** *c-type*)
norm-bytes :: *'a itself* ⇒ *normalisor* **where**
norm-bytes t ≡ *norm-tu* (*export-uinfo* (*typ-info-t t*))

definition (**in** *c-type*) *to-bytes-p* :: *'a* ⇒ *byte list* **where**
to-bytes-p v ≡ *to-bytes v* (*replicate* (*size-of TYPE*(*'a*)) *0*)

definition (**in** *c-type*) *zero* ::*'a* **where**
zero ≡ *from-bytes* (*replicate* (*size-of TYPE*(*'a*)) *0*)

hide-const (**open**) *zero* — mandatory qualifier: *c-type-class.zero*

syntax
-zero :: *type* ⇒ *logic* (‹(‹*indent=1 notation=‹mixfix ZERO››ZERO*/(*1 '*(*-'*))›)
syntax-consts
-zero == *c-type-class.zero*
translations
ZERO(*'a*) => *CONST c-type-class.zero* :: (*'a*)

⟨*ML*⟩

primrec
align-td-wo-align :: (*'a*, *'b*) *typ-desc* ⇒ *nat* **and**
align-td-wo-align-struct :: (*'a*, *'b*) *typ-struct* ⇒ *nat* **and**
align-td-wo-align-list :: (*'a*, *'b*) *typ-tuple list* ⇒ *nat* **and**
align-td-wo-align-tuple :: (*'a*, *'b*) *typ-tuple* ⇒ *nat*
where
al0: *align-td-wo-align* (*TypDesc algn st nm*) = *align-td-wo-align-struct st*

| *al1*: *align-td-wo-align-struct* (*TypScalar n algn d*) = *algn*
| *al2*: *align-td-wo-align-struct* (*TypAggregate xs*) = *align-td-wo-align-list xs*

| *al3*: *align-td-wo-align-list* [] = *0*
| *al4*: *align-td-wo-align-list* (*x*#*xs*) = *max* (*align-td-wo-align-tuple x*) (*align-td-wo-align-list xs*)

| *al5*: *align-td-wo-align-tuple* (*DTuple t n d*) = *align-td-wo-align t*

primrec
align-td :: (*'a*, *'b*) *typ-desc* ⇒ *nat* **and**
align-td-struct :: (*'a*, *'b*) *typ-struct* ⇒ *nat* **and**
align-td-list :: (*'a*, *'b*) *typ-tuple list* ⇒ *nat* **and**
align-td-tuple :: (*'a*, *'b*) *typ-tuple* ⇒ *nat*
where
al0: *align-td* (*TypDesc algn st nm*) = *algn*

| *al1*: *align-td-struct* (*TypScalar n algn d*) = *algn*
| *al2*: *align-td-struct* (*TypAggregate xs*) = *align-td-list xs*

| *al3*: *align-td-list* [] = *0*
| *al4*: *align-td-list* (*x*#*xs*) = *max* (*align-td-tuple x*) (*align-td-list xs*)

| *al5*: *align-td-tuple* (*DTuple t n d*) = *align-td t*

primrec
  wf-align :: (*'a*, *'b*) *typ-desc* ⇒ *bool* **and**
  wf-align-struct :: (*'a*, *'b*) *typ-struct* ⇒ *bool* **and**
  wf-align-list :: (*'a*, *'b*) *typ-tuple list* ⇒ *bool* **and**
  wf-align-tuple :: (*'a*, *'b*) *typ-tuple* ⇒ *bool*
where
  wfal0: *wf-align* (*TypDesc algn ts n*) = (*align-td-wo-align-struct ts* ≤ *algn* ∧
                                    align-td-struct ts ≤ *algn* ∧
                                    wf-align-struct ts)

| *wfal1*: *wf-align-struct* (*TypScalar n algn d*) = *True*
| *wfal2*: *wf-align-struct* (*TypAggregate ts*) = (*wf-align-list ts*)

| *wfal3*: *wf-align-list* [] = *True*
| *wfal4*: *wf-align-list* (*x*#*xs*) =
          (*wf-align-tuple x* ∧ *wf-align-list xs*)

| *wfal5*: *wf-align-tuple* (*DTuple x n d*) = *wf-align x*

definition (**in** *c-type*) *align-of* :: *'a itself* ⇒ *nat* **where**
  align-of t ≡ *2*^(*align-td* (*typ-info-t TYPE*(*'a*)))

lemma *align-td-wo-align-le-align-td*:
  fixes *t*::(*'a*,*'b*) *typ-info* **and**
   st::(*'a*,*'b*) *typ-info-struct* **and**
   fs::(*'a*,*'b*) *typ-info-tuple list* **and**
   f::(*'a*,*'b*) *typ-info-tuple*
 shows
 wf-align t ⟹ *align-td-wo-align t* ≤ *align-td t*
 wf-align-struct st ⟹ *align-td-wo-align-struct st* ≤ *align-td-struct st*
 wf-align-list fs ⟹ *align-td-wo-align-list fs* ≤ *align-td-list fs*
 wf-align-tuple f ⟹ *align-td-wo-align-tuple f* ≤ *align-td-tuple f*
    ⟨*proof*⟩

lemma (**in** *c-type*) *align-td-wo-align-le-align-of*:
  assumes *wf*: *wf-align* (*typ-info-t TYPE*(*'a*))
  shows *2*^(*align-td-wo-align* (*typ-info-t TYPE*(*'a*))) ≤ *align-of* (*TYPE*(*'a*))
  ⟨*proof*⟩

definition (**in** *c-type*)
  ptr-add :: *'a ptr* ⇒ *int* ⇒ *'a ptr* (**infixl** ‹+$_p$› *65*)
where
  ptr-add (*a* :: *'a ptr*) *w* ≡
    Ptr (*ptr-val a* + *of-int w* ∗ *of-nat* (*size-of* (*TYPE*(*'a*))))

lemma (**in** *c-type*) *ptr-add-def'*:
ptr-add (*Ptr p* :: ('*a*) *ptr*) *n*
= (*Ptr* (*p* + *of-int n* ∗ *of-nat* (*size-of TYPE*('*a*))))
⟨*proof*⟩

definition (**in** *c-type*)
ptr-sub :: '*a ptr* ⇒ '*a ptr* ⇒ *addr-bitsize signed word* (**infixl** ‹−$_p$› *65*)
where
ptr-sub (*a* :: '*a ptr*) *p* ≡
ucast (*ptr-val a* − *ptr-val p*) *div of-nat* (*size-of* (*TYPE*('*a*)))

definition (**in** *c-type*) *ptr-aligned* :: '*a ptr* ⇒ *bool* **where**
ptr-aligned p ≡ *align-of TYPE*('*a*) *dvd unat* (*ptr-val* (*p*::'*a ptr*))

type-synonym '*a ptr-guard* = '*a ptr* ⇒ *bool*

definition (**in** *c-type*) *c-null-guard* :: '*a ptr-guard* **where**
c-null-guard ≡ λ*p*. *0* ∉ {*ptr-val p*..+*size-of TYPE*('*a*)}

definition (**in** *c-type*) *c-guard* :: '*a ptr-guard* **where**
c-guard ≡ λ*p*. *ptr-aligned p* ∧ *c-null-guard p*

primrec
td-set :: ('*a*, '*b*) *typ-desc* ⇒ *nat* ⇒ (('*a*, '*b*) *typ-desc* × *nat*) *set* **and**
td-set-struct :: ('*a*, '*b*) *typ-struct* ⇒ *nat* ⇒ (('*a*, '*b*) *typ-desc* × *nat*) *set* **and**
td-set-list :: ('*a*, '*b*) *typ-tuple list* ⇒ *nat* ⇒ (('*a*,'*b*) *typ-desc* × *nat*) *set* **and**
td-set-tuple :: ('*a*, '*b*) *typ-tuple* ⇒ *nat* ⇒ (('*a*, '*b*) *typ-desc* × *nat*) *set*
where
ts0: *td-set* (*TypDesc algn st nm*) *m* = {(*TypDesc algn st nm*,*m*)} ∪ *td-set-struct st m*

| *ts1*: *td-set-struct* (*TypScalar n algn d*) *m* = {}
| *ts2*: *td-set-struct* (*TypAggregate xs*) *m* = *td-set-list xs m*

| *ts3*: *td-set-list* [] *m* = {}
| *ts4*: *td-set-list* (*x*#*xs*) *m* = *td-set-tuple x m* ∪ *td-set-list xs* (*m* + *size-td* (*dt-fst x*))

| *ts5*: *td-set-tuple* (*DTuple t nm d*) *m* = *td-set t m*

instantiation *typ-desc* :: (*type*, *type*) *ord*
begin

definition
typ-tag-le-def: *s* ≤ (*t*::('*a*, '*b*) *typ-desc*) ≡ (∃ *n*. (*s*,*n*) ∈ *td-set t 0*)
definition
typ-tag-lt-def: *s* < (*t*::('*a*, '*b*) *typ-desc*) ≡ *s* ≤ *t* ∧ *s* ≠ *t*
instance ⟨*proof*⟩

end

definition
fd-cons-double-update :: (*'a*, *'x*) *field-desc-scheme* ⇒ *bool*
where
fd-cons-double-update d ≡
(∀ *v bs bs'*. *length bs* = *length bs'* ⟶ *field-update d bs* (*field-update d bs' v*) = *field-update d bs v*)

definition
fd-cons-update-access :: (*'a*, *'x*) *field-desc-scheme* ⇒ *nat* ⇒ *bool*
where
fd-cons-update-access d n ≡
(∀ *v bs*. *length bs* = *n* ⟶ *field-update d* (*field-access d v bs*) *v* = *v*)

definition
norm-desc :: (*'a*, *'x*) *field-desc-scheme* ⇒ *nat* ⇒ (*byte list* ⇒ *byte list*)
where
norm-desc d n ≡ λ*bs*. *field-access d* (*field-update d bs undefined*) (*replicate n 0*)

definition
fd-cons-length :: (*'a*, *'x*) *field-desc-scheme* ⇒ *nat* ⇒ *bool*
where
fd-cons-length d n ≡ ∀ *v bs*. *length bs* = *n* ⟶ *length* (*field-access d v bs*) = *n*

definition
fd-cons-access-update :: (*'a*, *'x*) *field-desc-scheme* ⇒ *nat* ⇒ *bool*
where
fd-cons-access-update d n ≡ ∀ *bs bs' v v'*. *length bs* = *n* ⟶
length bs' = *n* ⟶
field-access d (*field-update d bs v*) *bs'* = *field-access d* (*field-update d bs v'*) *bs'*

definition
fd-cons-update-normalise :: (*'a*, *'x*) *field-desc-scheme* ⇒ *nat* ⇒ *bool*
where
fd-cons-update-normalise d n ≡
(∀ *v bs*. *length bs*=*n* ⟶ *field-update d* (*norm-desc d n bs*) *v* = *field-update d bs v*)

definition
fd-cons-desc :: (*'a*,*'x*) *field-desc-scheme* ⇒ *nat* ⇒ *bool*
where
fd-cons-desc d n ≡ *fd-cons-double-update d* ∧
fd-cons-update-access d n ∧
fd-cons-access-update d n ∧

fd-cons-length d n

definition
fd-cons :: (*'a*, *'b*) *typ-info* ⇒ *bool*
where
fd-cons t ≡ *fd-cons-desc* (*field-desc t*) (*size-td t*)

definition
fd-cons-struct :: (*'a*, *'b*) *typ-info-struct* ⇒ *bool*
where
fd-cons-struct t ≡ *fd-cons-desc* (*field-desc-struct t*) (*size-td-struct t*)

definition
fd-cons-list :: (*'a*, *'b*) *typ-info-tuple list* ⇒ *bool*
where
fd-cons-list t ≡ *fd-cons-desc* (*field-desc-list t*) (*size-td-list t*)

definition
fd-cons-tuple :: (*'a*, *'b*) *typ-info-tuple* ⇒ *bool*
where
fd-cons-tuple t ≡ *fd-cons-desc* (*field-desc-tuple t*) (*size-td-tuple t*)

definition
fa-fu-ind :: *'a field-desc* ⇒ *'a field-desc* ⇒ *nat* ⇒ *nat* ⇒*bool*
where
fa-fu-ind d d' n n' ≡ ∀ *v bs bs'*. *length bs* = *n* ⟶ *length bs'* = *n'* ⟶
field-access d (*field-update d' bs v*) *bs'* = *field-access d v bs'*

definition
wf-fdp :: ((*'a*, *'b*) *typ-info* × *qualified-field-name*) *set* ⇒ *bool*
where
wf-fdp t ≡ ∀ *x m*. (*x*,*m*) ∈ *t* ⟶ (*fd-cons x* ∧ (∀ *y n*. (*y*,*n*) ∈ *t* ∧ ¬ *m* ≤ *n* ∧ ¬ *n* ≤ *m*
⟶ *fu-commutes* (*field-update* (*field-desc x*)) (*field-update* (*field-desc y*)) ∧
fa-fu-ind (*field-desc x*) (*field-desc y*) (*size-td y*) (*size-td x*)))

lemma *wf-fdp-list*:
wf-fdp (*xs* ∪ *ys*) ⟹ *wf-fdp xs* ∧ *wf-fdp ys*
⟨*proof*⟩

primrec
wf-fd :: (*'a*, *'b*) *typ-info* ⇒ *bool* **and**
wf-fd-struct :: (*'a*, *'b*) *typ-info-struct* ⇒ *bool* **and**
wf-fd-list :: (*'a*, *'b*) *typ-info-tuple list* ⇒ *bool* **and**
wf-fd-tuple :: (*'a*, *'b*) *typ-info-tuple* ⇒ *bool*
where
wffd0: *wf-fd* (*TypDesc algn ts n*) = (*wf-fd-struct ts*)

| *wffd1*: *wf-fd-struct* (*TypScalar n algn d*) = *fd-cons-struct* (*TypScalar n algn d*::(*'a*,*'b*) *typ-info-struct*)
| *wffd2*: *wf-fd-struct* (*TypAggregate ts*) = *wf-fd-list ts*

| *wffd3*: *wf-fd-list* [] = *True*
| *wffd4*: *wf-fd-list* (*x*#*xs*) = (*wf-fd-tuple x* ∧ *wf-fd-list xs* ∧
fu-commutes (*update-ti-tuple-t x*) (*update-ti-list-t xs*) ∧
fa-fu-ind (*field-desc-tuple x*) (*field-desc-list xs*) (*size-td-list xs*) (*size-td-tuple x*)∧
fa-fu-ind (*field-desc-list xs*) (*field-desc-tuple x*) (*size-td-tuple x*) (*size-td-list xs*))

| *wffd5*: *wf-fd-tuple* (*DTuple x n d*) = *wf-fd x*

definition
tf-set :: (*'a*, *'b*) *typ-info* ⇒ ((*'a*, *'b*) *typ-info* × *qualified-field-name*) *set*
where
tf-set td ≡ {(*s*,*f*) | *s f*. ∃ *n*. *field-lookup td f 0* = *Some* (*s*,*n*)}

definition
tf-set-struct :: (*'a*, *'b*) *typ-info-struct* ⇒ ((*'a*, *'b*) *typ-info* × *qualified-field-name*) *set*
where
tf-set-struct td ≡ {(*s*,*f*) | *s f*. ∃ *n*. *field-lookup-struct td f 0* = *Some* (*s*,*n*)}

definition
tf-set-list :: (*'a*, *'b*) *typ-info-tuple list* ⇒ ((*'a*, *'b*) *typ-info* × *qualified-field-name*) *set*
where
tf-set-list td ≡ {(*s*,*f*) | *s f*. ∃ *n*. *field-lookup-list td f 0* = *Some* (*s*,*n*)}

definition
tf-set-tuple :: (*'a*, *'b*) *typ-info-tuple* ⇒ ((*'a*, *'b*) *typ-info* × *qualified-field-name*) *set*
where
tf-set-tuple td ≡ {(*s*,*f*) | *s f*. ∃ *n*. *field-lookup-tuple td f 0* = *Some* (*s*,*n*)}

record *'a leaf-desc* =
lf-fd :: *'a field-desc*
lf-sz :: *nat*
lf-fn :: *qualified-field-name*

primrec
lf-set :: (*'a*, *'b*) *typ-info* ⇒ *qualified-field-name* ⇒ *'a leaf-desc set* **and**
lf-set-struct :: (*'a*, *'b*) *typ-info-struct* ⇒ *qualified-field-name* ⇒ *'a leaf-desc set*
and

lf-set-list :: (*'a*, *'b*) *typ-info-tuple list* ⇒ *qualified-field-name* ⇒ *'a leaf-desc set*
and
lf-set-tuple :: (*'a*, *'b*) *typ-info-tuple* ⇒ *qualified-field-name* ⇒ *'a leaf-desc set*
where
fds0: *lf-set* (*TypDesc algn st nm*) *fn* = *lf-set-struct st fn*

| *fds1*: *lf-set-struct* (*TypScalar n algn d*) *fn* = {(⦇ *lf-fd* = *d*, *lf-sz* = *n*, *lf-fn* = *fn* ⦈)}
| *fds2*: *lf-set-struct* (*TypAggregate xs*) *fn* = *lf-set-list xs fn*

| *fds3*: *lf-set-list* [] *fn* = {}
| *fds4*: *lf-set-list* (*x*#*xs*) *fn* = *lf-set-tuple x fn* ∪ *lf-set-list xs fn*

| *fds5*: *lf-set-tuple* (*DTuple t n d*) *fn* = *lf-set t* (*fn*@[*n*])

definition
wf-lf :: *'a leaf-desc set* ⇒ *bool*
where
wf-lf D ≡ ∀ *x*. *x* ∈ *D* ⟶ (*fd-cons-desc* (*lf-fd x*) (*lf-sz x*) ∧ (∀ *y*. *y* ∈ *D* ⟶ *lf-fn y* ≠ *lf-fn x*
⟶ *fu-commutes* (*field-update* (*lf-fd x*)) (*field-update* (*lf-fd y*)) ∧
fa-fu-ind (*lf-fd x*) (*lf-fd y*) (*lf-sz y*) (*lf-sz x*)))

definition
ti-ind :: *'a leaf-desc set* ⇒ *'a leaf-desc set* ⇒ *bool*
where
ti-ind X Y ≡ ∀ *x y*. *x* ∈ *X* ∧ *y* ∈ *Y* ⟶ (
fu-commutes (*field-update* (*lf-fd x*)) (*field-update* (*lf-fd y*)) ∧
fa-fu-ind (*lf-fd x*) (*lf-fd y*) (*lf-sz y*) (*lf-sz x*) ∧
fa-fu-ind (*lf-fd y*) (*lf-fd x*) (*lf-sz x*) (*lf-sz y*))

definition
t2d :: ((*'a*, *'b*) *typ-info* × *qualified-field-name*) ⇒ *'a leaf-desc*
where
t2d x ≡ ⦇ *lf-fd* = *field-desc* (*fst x*), *lf-sz* = *size-td* (*fst x*), *lf-fn* = *snd x*⦈

definition
fd-consistent :: (*'a*, *'b*) *typ-info* ⇒ *bool*
where
fd-consistent t ≡ ∀ *f s n*. *field-lookup t f 0* = *Some* (*s*,*n*)
⟶ *fd-cons s*

class *wf-type* = *c-type* +
assumes *wf-desc* [*simp*]: *wf-desc* (*typ-info-t TYPE*(*'a*))
assumes *wf-size-desc* [*simp*]: *wf-size-desc* (*typ-info-t TYPE*(*'a*))
assumes *wf-lf* [*simp*]: *wf-lf* (*lf-set* (*typ-info-t TYPE*(*'a*)) [])

definition
super-update-bs :: *byte list* ⇒ *byte list* ⇒ *nat* ⇒ *byte list*
where
super-update-bs v bs n ≡ *take n bs* @ *v* @
drop (*n* + *length v*) *bs*

definition
disj-fn :: *qualified-field-name* ⇒ *qualified-field-name* ⇒ *bool*
where
disj-fn s t ≡ ¬ *s* ≤ *t* ∧ ¬ *t* ≤ *s*

definition
fs-path :: *qualified-field-name list* ⇒ *qualified-field-name set*
where
fs-path xs ≡ {*x*. ∃ *k*. *k* ∈ *set xs* ∧ *x* ≤ *k*} ∪ {*x*. ∃ *k*. *k* ∈ *set xs* ∧ *k* ≤ *x*}

definition
field-names :: ('*a*, '*b*) *typ-desc* ⇒ *qualified-field-name set*
where
field-names t ≡ {*f*. *field-lookup t f 0* ≠ *None*}

definition
align-field :: ('*a*, '*b*) *typ-desc* ⇒ *bool*
where
align-field ti ≡ ∀ *f s n*. *field-lookup ti f 0* = *Some* (*s*,*n*) ⟶
2^(*align-td s*) *dvd n*

class *mem-type-sans-size* = *wf-type* +
assumes *upd*:
length bs = *size-of TYPE*('*a*) ⟶
update-ti-t (*typ-info-t TYPE*('*a*)) *bs v*
= *update-ti-t* (*typ-info-t TYPE*('*a*)) *bs w*
assumes *align-size-of*: *align-of* (*TYPE*('*a*)) *dvd size-of TYPE*('*a*)
assumes *align-field*: *align-field* (*typ-info-t TYPE*('*a*))
assumes *wf-align*: *wf-align* (*typ-info-t TYPE*('*a*))

class *mem-type* = *mem-type-sans-size* +
assumes *max-size*: *size-of* (*TYPE*('*a*)) < *addr-card*

primrec
aggregate :: ('*a*, '*b*) *typ-desc* ⇒ *bool* **and**
aggregate-struct :: ('*a*, '*b*) *typ-struct* ⇒ *bool*
where
aggregate (*TypDesc algn st tn*) = *aggregate-struct st*

| *aggregate-struct* (*TypScalar n algn d*) = *False*

| *aggregate-struct* (*TypAggregate ts*) = *True*

class *simple-mem-type* = *mem-type* +
  assumes *simple-tag*: ¬ *aggregate* (*typ-info-t TYPE*('*a*))

definition
  field-of :: *addr* ⇒ ('*a*, '*b*) *typ-desc* ⇒ ('*a*, '*b*) *typ-desc* ⇒ *bool*
where
  field-of q s t ≡ (*s*,*unat q*) ∈ *td-set t 0*

definition (**in** *c-type*)
  field-of-t :: '*a ptr* ⇒ '*b*::*c-type ptr* ⇒ *bool*
where
  field-of-t p q ≡ *field-of* (*ptr-val p* − *ptr-val q*) (*typ-uinfo-t TYPE*('*a*))
    (*typ-uinfo-t TYPE*('*b*))

definition (**in** *c-type*)
  h-val :: *heap-mem* ⇒ '*a ptr* ⇒ '*a*
where
  h-val h ≡ λ*p*. *from-bytes* (*heap-list h* (*size-of TYPE* ('*a*))
    (*ptr-val* (*p*::'*a ptr*)))

primrec
  heap-update-list :: *addr* ⇒ *byte list* ⇒ *heap-mem* ⇒ *heap-mem*
where
  heap-update-list-base: *heap-update-list p* [] *h* = *h*
| *heap-update-list-rec*:
  heap-update-list p (*x*#*xs*) *h* = *heap-update-list* (*p* + *1*) *xs* (*h*(*p*:= *x*))

type-synonym '*a typ-heap-g* = '*a ptr* ⇒ '*a*

definition (**in** *c-type*)
  lift :: *heap-mem* ⇒ '*a typ-heap-g*
where
  lift h ≡ *h-val h*

definition (**in** *c-type*)
  heap-update :: '*a ptr* ⇒ '*a* ⇒ *heap-mem* ⇒ *heap-mem*
where
  heap-update p v h ≡ *heap-update-list* (*ptr-val p*) (*to-bytes v* (*heap-list h* (*size-of* *TYPE*('*a*)) (*ptr-val p*))) *h*

definition (**in** *c-type*)
  heap-update-padding :: '*a ptr* ⇒ '*a* ⇒ *byte list* ⇒ *heap-mem* ⇒ *heap-mem* **where**
heap-update-padding p v bs h =
  heap-update-list (*ptr-val p*) (*to-bytes v bs*) *h*

lemma (**in** *c-type*) *heap-update-heap-update-padding-conv*:

heap-update p v h = heap-update-padding p v (heap-list h (size-of TYPE('a)) (ptr-val p)) h
⟨*proof*⟩

definition (**in** *c-type*) *heap-upd* **where**
heap-upd p f s = heap-update p (f (h-val s p)) s

definition *heap-upd-list* :: *nat* ⇒ *addr* ⇒ (*byte list* ⇒ *byte list*) ⇒ *heap-mem* ⇒ *heap-mem* **where**
heap-upd-list n p f h = heap-update-list p (f (heap-list h n p)) h

fun
fold-td' :: (*typ-name* ⇒ ('*a* × *field-name*) *list* ⇒ '*a*) × ('*a*, '*b*) *typ-desc* ⇒ '*a*
where
fot0: *fold-td'* (*f*,*TypDesc algn st nm*) = (*case st of*
TypScalar n algn d ⇒ *d* |
TypAggregate ts ⇒ *f nm* (*map* (λ*x*. *case x of DTuple t n d* ⇒ (*fold-td'* (*f*,*t*),*n*)) *ts*))

definition
fold-td :: (*typ-name* ⇒ ('*a* × *field-name*) *list* ⇒ '*a*) ⇒ ('*a*, '*b*) *typ-desc* ⇒ '*a*
where
fold-td ≡ λ*f t*. *fold-td'* (*f*,*t*)

declare *fold-td-def* [*simp*]

definition
fold-td-struct :: *typ-name* ⇒ (*typ-name* ⇒ ('*a* × *field-name*) *list* ⇒ '*a*) ⇒ ('*a*, '*b*) *typ-struct* ⇒ '*a*
where
fold-td-struct tn f st ≡ (*case st of*
TypScalar n algn d ⇒ *d* |
TypAggregate ts ⇒ *f tn* (*map* (λ*x*. *case x of DTuple t n d* ⇒ (*fold-td'* (*f*,*t*),*n*)) *ts*))

declare *fold-td-struct-def* [*simp*]

definition
fold-td-list :: *typ-name* ⇒ (*typ-name* ⇒ ('*a* × *field-name*) *list* ⇒ '*a*) ⇒ ('*a*, '*b*) *typ-tuple list* ⇒ '*a*
where
fold-td-list tn f ts ≡ *f tn* (*map* (λ*x*. *case x of DTuple t n d* ⇒ (*fold-td'* (*f*,*t*),*n*)) *ts*)

declare *fold-td-list-def* [*simp*]

definition
fold-td-tuple :: (*typ-name* ⇒ ('*a* × *field-name*) *list* ⇒ '*a*) ⇒ ('*a*, '*b*) *typ-tuple* ⇒

$'a$
where
fold-td-tuple f x ≡ (*case x of DTuple t n d* ⇒ *fold-td'* (*f*,*t*))

declare *fold-td-tuple-def* [*simp*]

fun
map-td' :: ((*nat* ⇒ *nat* ⇒ *'a* ⇒ *'b*) × (*'c* ⇒ *'d*)) × (*'a*, *'c*) *typ-desc* ⇒ (*'b*, *'d*) *typ-desc*
where
map-td' ((*f*,*g*),*TypDesc algn st nm*) = (*TypDesc algn* (*case st of*
TypScalar n algn d ⇒ *TypScalar n algn* (*f n algn d*) |
TypAggregate ts ⇒ *TypAggregate* (*map* (λ*x*. *case x of DTuple t n d* ⇒ *DTuple* (*map-td'* ((*f*,*g*),*t*)) *n* (*g d*)) *ts*)) *nm*)

definition
tnSum :: *typ-name* ⇒ (*nat* × *field-name*) *list* ⇒ *nat*
where
tnSum ≡ λ*tn ts*. *foldr* ((+) *o fst*) *ts 0*

definition
tnMax :: *typ-name* ⇒ (*nat* × *field-name*) *list* ⇒ *nat*
where
tnMax ≡ λ*tn ts*. *foldr* (λ*x y*. *max* (*fst x*) *y*) *ts 0*

definition
wfd :: *typ-name* ⇒ (*bool* × *field-name*) *list* ⇒ *bool*
where
wfd ≡ λ*tn ts*. *distinct* (*map snd ts*) ∧ *foldr* (∧) (*map fst ts*) *True*

definition
wfsd :: *typ-name* ⇒ (*bool* × *field-name*) *list* ⇒ *bool*
where
wfsd ≡ λ*tn ts*. *ts* ≠ [] ∧ *foldr* (∧) (*map fst ts*) *True*

definition *component-desc-tuple t* ≡ *case t of* (*DTuple t n d*) ⇒ *d*

lemma *component-desc-tuple-simp* [*simp*]: *component-desc-tuple* (*DTuple t n d*) = *d*
⟨*proof*⟩

primrec *split-list*:: *nat list* ⇒ *'a list* ⇒ *'a list list* **where**
split-list [] *bs* = [*bs*]
| *split-list* (*n*#*ns*) *bs* = *take n bs* # *split-list ns* (*drop n bs*)

lemma *concat-split* [*simp*]: *concat* (*split-list ns bs*) = *bs*
⟨*proof*⟩

primrec *split-map'*:: ('*b* ⇒ *nat*) ⇒ ('*b* ⇒ '*a list* ⇒ '*c*) ⇒ '*b list* ⇒ '*a list* ⇒ '*c list* **where**
  split-map' n f [] *bs* = []
| *split-map' n f* (*x*#*xs*) *bs* = *f x* (*take* (*n x*) *bs*) # *split-map' n f xs* (*drop* (*n x*) *bs*)

lemma *length-split-map'* [*simp*]: *length* (*split-map' n f xs bs*) = *length xs*
  ⟨*proof*⟩

lemma *split-map' n f xs bs* = *map* (λ(*f'*,*bs'*). *f' bs'*) (*zip* (*map f xs*) (*split-list* (*map n xs*) *bs*))
  ⟨*proof*⟩

primrec *split-map*:: ('*b* ⇒ '*a list* ⇒ '*c list*) ⇒ '*b list* ⇒ '*a list* ⇒ '*c list list* **where**
  split-map f [] *bs* = []
| *split-map f* (*x*#*xs*) *bs* = *f x bs* # *split-map f xs* (*drop* (*length* (*f x bs*)) *bs*)

lemma *length-split-map* [*simp*]: *length* (*split-map f xs bs*) = *length xs*
  ⟨*proof*⟩

primrec *split-fold*:: ('*b* ⇒ '*a list* ⇒ '*d* ⇒ ('*d* × '*a list*)) ⇒ '*b list* ⇒ '*a list* ⇒ '*d* ⇒ ('*d* × '*a list*) **where**
  split-fold f [] *bs s* = (*s*, *bs*)
| *split-fold f* (*x*#*xs*) *bs s* = (*let* (*s'*, *bs'*) = *f x bs s in split-fold f xs bs' s'*)

definition *apply-field-update*::'*a field-desc* ⇒ *byte list* ⇒ '*a* ⇒ ('*a* × *byte list*) **where**
  apply-field-update d bs s ≡ (*field-update d bs s*, *drop* (*field-sz d*) *bs*)

definition *apply-field-updates* :: '*a field-desc list* ⇒ *byte list* ⇒ '*a* ⇒ ('*a* × *byte list*) **where**
  apply-field-updates ≡ *split-fold apply-field-update*

lemma *apply-field-updates-Nil* [*simp*]: *apply-field-updates* [] *bs s* = (*s*, *bs*)
  ⟨*proof*⟩

lemma *apply-field-updates-Cons* [*simp*]: *apply-field-updates* (*d*#*ds*) *bs s* =
  apply-field-updates ds (*drop* (*field-sz d*) *bs*) (*field-update d bs s*)
  ⟨*proof*⟩

definition *component-access-tuple v t bs* ≡ *case t of* (*DTuple t n d*) ⇒ *field-access d v bs*

lemma *component-access-tuple-simp* [*simp*]: *component-access-tuple v* (*DTuple t n d*) = *field-access d v*
  ⟨*proof*⟩

definition *component-access-struct st v bs* =
(*case st of TypScalar - - d* ⇒ *field-access d v bs*
| *TypAggregate fs* ⇒ *concat* (*split-map* (*component-access-tuple v*) *fs bs*))

lemma *component-access-struct-scalar* [*simp*]:
component-access-struct (*TypScalar sz align d*) = *field-access d*
⟨*proof*⟩

lemma *component-access-struct-aggregate* [*simp*]:
component-access-struct (*TypAggregate fs*) *v bs* = *concat* (*split-map* (*component-access-tuple v*) *fs bs*)
⟨*proof*⟩

lemma *component-access-struct-aggregate-nil* [*simp*]:
component-access-struct (*TypAggregate* []) = (λ*v bs*. [])
⟨*proof*⟩

definition *component-update-tuple t bs* ≡ *case t of* (*DTuple t n d*) ⇒ *field-update d bs*

lemma *component-update-tuple-simp* [*simp*]: *component-update-tuple* (*DTuple t n d*) = *field-update d*
⟨*proof*⟩

definition *component-update-struct st bs v* =
(*case st of TypScalar - - d* ⇒ *field-update d bs v*
| *TypAggregate fs* ⇒ *fst* (*apply-field-updates* (*map dt-trd fs*) *bs v*))

lemma *component-update-struct-scalar* [*simp*]:
component-update-struct (*TypScalar sz align d*) = *field-update d*
⟨*proof*⟩

lemma *component-update-struct-aggregate* [*simp*]:
component-update-struct (*TypAggregate fs*) *bs v* = *fst* (*apply-field-updates* (*map dt-trd fs*) *bs v*)
⟨*proof*⟩

lemma *component-update-struct-aggregate-nil* [*simp*]:
component-update-struct (*TypAggregate* []) = (λ*bs*. *id*)
⟨*proof*⟩

definition *component-sz-struct st* =
(*case st of TypScalar - - d* ⇒ *field-sz d*
| *TypAggregate fs* ⇒ *foldl* (+) *0* (*map* (*field-sz* ∘ *dt-trd*) *fs*))

lemma *component-sz-struct-scalar* [*simp*]:
component-sz-struct (*TypScalar sz align d*) = *field-sz d*
⟨*proof*⟩

lemma *component-sz-struct-aggregate* [*simp*]:
component-sz-struct (*TypAggregate fs*) = *foldl* (+) *0* (*map* (*field-sz o dt-trd*) *fs*)
⟨*proof*⟩

definition *component-desc-struct st* =
(|*field-access* = *component-access-struct st*,
field-update = *component-update-struct st*,
field-sz = *component-sz-struct st*|)

lemma *component-desc-struct-simps* [*simp*]:
field-access (*component-desc-struct st*) = *component-access-struct st*
field-update (*component-desc-struct st*) = *component-update-struct st*
field-sz (*component-desc-struct st*) = *component-sz-struct st*
component-desc-struct (*TypScalar sz align d*) = *d*
⟨*proof*⟩

definition *component-desc t* ≡ *case t of TypDesc algn st n* ⇒ *component-desc-struct st*

lemma *component-desc-simps* [*simp*]: *component-desc* (*TypDesc algn st n*) = *component-desc-struct st*
⟨*proof*⟩

definition (**in** *c-type*) *xto-bytes* :: *'a* ⇒ *byte list* ⇒ *byte list* **where**
xto-bytes ≡ *field-access* (*component-desc* (*typ-info-t TYPE*(*'a*)))

definition (**in** *c-type*) *xfrom-bytes* :: *byte list* ⇒ *'a* **where**
xfrom-bytes bs ≡ *field-update* (*component-desc* (*typ-info-t TYPE*(*'a*))) *bs undefined*

primrec
toplevel-field-descs :: *'a xtyp-info* ⇒ *'a field-desc list* **and**
toplevel-field-descs-struct :: *'a xtyp-info-struct* ⇒ *'a field-desc list* **and**
toplevel-field-descs-list :: *'a xtyp-info-tuple list* ⇒ *'a field-desc list* **and**
toplevel-field-descs-tuple :: *'a xtyp-info-tuple* ⇒ *'a field-desc list*
where
tfd0: *toplevel-field-descs* (*TypDesc algn st nm*) = *toplevel-field-descs-struct st*

| *tfd1*: *toplevel-field-descs-struct* (*TypScalar n algn d*) = [*d*]

| *tfd2*: *toplevel-field-descs-struct* (*TypAggregate xs*) = *toplevel-field-descs-list xs*

| *tfd3*: *toplevel-field-descs-list* [] = []
| *tfd4*: *toplevel-field-descs-list* (*x*#*xs*) = *toplevel-field-descs-tuple x* @ *toplevel-field-descs-list xs*

| *tfd5*: *toplevel-field-descs-tuple* (*DTuple t nm d*) = [*d*]

locale *padding-base* =
  fixes *acc*::'*a* ⇒ *byte list* ⇒ *byte list*
  fixes *upd*:: *byte list* ⇒ '*a* ⇒ '*a*
  fixes *sz*::*nat*
begin
  definition *eq-padding*::*byte list* ⇒ *byte list* ⇒ *bool* **where**
    eq-padding bs bs' ≡ *length bs* = *sz* ∧ *length bs'* = *sz* ∧ (∀ *v*. *acc v bs* = *acc v bs'*)
  definition *eq-upto-padding*::*byte list* ⇒ *byte list* ⇒ *bool* **where**
    eq-upto-padding bs bs' ≡ *length bs* = *sz* ∧ *length bs'* = *sz* ∧ (∀ *v*. *upd bs v* = *upd bs' v*)

definition *is-padding-byte*::*nat* ⇒ *bool* **where**
  is-padding-byte i ≡ *i* < *sz* ∧ (∀ *v bs*. *length bs* = *sz* ⟶
    (*acc v bs* ! *i* = *bs* ! *i*) ∧
    (∀ *b*. *upd bs v* = *upd* (*bs*[*i*:=*b*]) *v*))

definition *is-value-byte*::*nat* ⇒ *bool* **where**
  is-value-byte i ≡ *i* < *sz* ∧ (∀ *v bs*. *length bs* = *sz* ⟶
    (∀ *bs'*. *length bs'* = *sz* ⟶ (*acc* (*upd bs v*) *bs'* ! *i*) = *bs* ! *i*) ∧
    (∀ *b*. *acc v bs* = *acc v* (*bs*[*i*:=*b*])))

lemma *is-padding-byteI*:
  assumes *i* < *sz*
  assumes ⋀*v bs*. *i* < *sz* ⟹ *length bs* = *sz* ⟹ *acc v bs* ! *i* = *bs* ! *i*
  assumes ⋀*v bs b*. *i* < *sz* ⟹ *length bs* = *sz* ⟹ *upd bs v* = *upd* (*bs*[*i*:=*b*]) *v*
  shows *is-padding-byte i*
  ⟨*proof*⟩

lemma *is-value-byteI*:
  assumes *i* < *sz*
  assumes ⋀*v bs bs'*. *i* < *sz* ⟹ *length bs* = *sz* ⟹ *length bs'* = *sz* ⟹
      (*acc* (*upd bs v*) *bs'* ! *i*) = *bs* ! *i*
  assumes ⋀*v bs b*. *i* < *sz* ⟹ *length bs* = *sz* ⟹ *acc v bs* = *acc v* (*bs*[*i*:=*b*])
  shows *is-value-byte i*
  ⟨*proof*⟩

lemma *is-padding-byte-acc-id*:
  is-padding-byte i ⟹ *length bs* = *sz* ⟹ (*acc v bs* ! *i*) = *bs* ! *i*

⟨*proof*⟩

lemma *is-value-byte-acc-upd-cancel*:
  is-value-byte i ⟹ *length bs* = *sz* ⟹ *length bs′* = *sz* ⟹
    (*acc* (*upd bs v*) *bs′* ! *i*) = *bs* ! *i*
⟨*proof*⟩

lemma *is-padding-byte-eq-upto-padding*: *is-padding-byte i* ⟹ *length bs* = *sz* ⟹ *eq-upto-padding bs* (*bs*[*i*:=*b*])
⟨*proof*⟩

lemma *is-value-byte-eq-padding*: *is-value-byte i* ⟹ *length bs* = *sz* ⟹ *eq-padding bs* (*bs*[*i*:=*b*])
⟨*proof*⟩

lemma *is-padding-byte-acc-neq*:
  is-padding-byte i ⟹ *b* ≠ *bs*!*i* ⟹ *length bs* = *sz* ⟹ *acc v bs* ≠ *acc v* (*bs*[*i*:=*b*])
⟨*proof*⟩

lemma *is-padding-byte-acc-eq*:
  is-padding-byte i ⟹ *length bs* = *sz* ⟹
      acc v bs ! *i* = *bs* ! *i*
⟨*proof*⟩

lemma *is-padding-byte-upd-eq*:
  is-padding-byte i ⟹ *length bs* = *sz* ⟹ *upd bs v* = *upd* (*bs*[*i*:=*b*]) *v*
⟨*proof*⟩

lemma *is-padding-byte-not-eq-padding*: *is-padding-byte i* ⟹ *b* ≠ *bs*!*i* ⟹ *length bs* = *sz* ⟹ ¬ *eq-padding bs* (*bs*[*i*:=*b*])
⟨*proof*⟩

lemma *is-value-byte-upd-neq*:
  is-value-byte i ⟹ *b* ≠ *bs*!*i* ⟹ *length bs* = *sz* ⟹ *upd bs v* ≠ *upd* (*bs*[*i*:=*b*]) *v*
⟨*proof*⟩

lemma *is-value-byte-update-depends*:
  assumes *is-value*: *is-value-byte i*
  assumes *lbs*: *length bs* = *sz*
  shows ∃ *b*. *upd* (*bs*[*i* := *b*]) *v* ≠ *upd bs v*
⟨*proof*⟩

lemma *is-value-byte-acc-eq*:
  is-value-byte i ⟹ *length bs* = *sz* ⟹ *acc v bs* = *acc v* (*bs*[*i*:=*b*])
⟨*proof*⟩

lemma *is-value-byte-not-eq-upto-padding*:
  is-value-byte i ⟹ *b* ≠ *bs*!*i* ⟹ *length bs* = *sz* ⟹ ¬ *eq-upto-padding bs* (*bs*[*i*:=*b*])

⟨*proof*⟩

lemma *eq-paddingI*[*intro?*]:
assumes *length bs* = *sz*
assumes *length bs'* = *sz*
assumes ⋀*v*. *acc v bs* = *acc v bs'*
shows *eq-padding bs bs'*
⟨*proof*⟩

lemma *eq-upto-paddingI*[*intro?*]:
assumes *length bs* = *sz*
assumes *length bs'* = *sz*
assumes ⋀*v*. *upd bs v* = *upd bs' v*
shows *eq-upto-padding bs bs'*
⟨*proof*⟩

lemma *eq-padding-length1*: *eq-padding bs bs'* ⟹ *length bs* = *sz*
⟨*proof*⟩

lemma *eq-padding-length2*: *eq-padding bs bs'* ⟹ *length bs'* = *sz*
⟨*proof*⟩

lemma *eq-upto-padding-length1*: *eq-upto-padding bs bs'* ⟹ *length bs* = *sz*
⟨*proof*⟩

lemma *eq-upto-padding-length2*: *eq-upto-padding bs bs'* ⟹ *length bs'* = *sz*
⟨*proof*⟩

lemma *eq-padding-length-eq*: *eq-padding bs bs'* ⟹ *length bs* = *length bs'*
⟨*proof*⟩

lemma *eq-upto-padding-length-eq*: *eq-upto-padding bs bs'* ⟹ *length bs* = *length bs'*
⟨*proof*⟩

lemma *eq-padding-acc*: *eq-padding bs bs'* ⟹ *acc v bs* = *acc v bs'*
⟨*proof*⟩

lemma *eq-upto-padding-upd*: *eq-upto-padding bs bs'* ⟹ *upd bs v* = *upd bs' v*
⟨*proof*⟩

lemma *eq-padding-refl*[*simp*]: *length bs* = *sz* ⟹ *eq-padding bs bs*
⟨*proof*⟩

lemma *eq-upto-padding-refl*[*simp*]: *length bs* = *sz* ⟹ *eq-upto-padding bs bs*
⟨*proof*⟩

lemma *eq-padding-sym*: *eq-padding bs bs'* $\longleftrightarrow$ *eq-padding bs' bs*
⟨*proof*⟩

lemma *eq-padding-symp*: *symp eq-padding*
⟨*proof*⟩

lemma *eq-upto-padding-sym*: *eq-upto-padding bs bs'* $\longleftrightarrow$ *eq-upto-padding bs' bs*
⟨*proof*⟩

lemma *eq-upto-padding-symp*: *symp eq-upto-padding*
⟨*proof*⟩

lemma *eq-padding-trans*: *eq-padding bs1 bs2* $\Longrightarrow$ *eq-padding bs2 bs3* $\Longrightarrow$ *eq-padding bs1 bs3*
⟨*proof*⟩

lemma *eq-padding-transp*: *transp eq-padding*
⟨*proof*⟩

lemma *eq-upto-padding-trans*: *eq-upto-padding bs1 bs2* $\Longrightarrow$ *eq-upto-padding bs2 bs3* $\Longrightarrow$ *eq-upto-padding bs1 bs3*
⟨*proof*⟩

lemma *eq-upto-padding-transp*: *transp eq-upto-padding*
⟨*proof*⟩

lemma *eq-padding-to-padding-byte-eq*: *eq-padding bs bs'* $\Longrightarrow$
length bs = *sz* $\wedge$ *length bs'* = *sz* $\wedge$ $(\forall i.$ *is-padding-byte i* $\longrightarrow$ *bs* ! *i* = *bs'* ! *i*)
⟨*proof*⟩

lemma *eq-upto-padding-to-value-byte-eq*: *eq-upto-padding bs bs'* $\Longrightarrow$
length bs = *sz* $\wedge$ *length bs'* = *sz* $\wedge$ $(\forall i.$ *is-value-byte i* $\longrightarrow$ *bs* ! *i* = *bs'* ! *i*)
⟨*proof*⟩

end

locale *complement-padding* = *padding-base* +
assumes *complement*: *i* < *sz* $\Longrightarrow$
is-padding-byte i $\neq$ *is-value-byte i*
begin

lemma *eq-padding-eq-upto-padding-complement*:
assumes *lbs*: *length bs* = *sz*
assumes *i-bound*: *i* < *length bs*
assumes *neq*: *b* $\neq$ *bs*!*i*
shows *eq-padding bs* (*bs*[*i* := *b*]) $\neq$ *eq-upto-padding bs* (*bs*[*i* := *b*])
⟨*proof*⟩

lemma *eq-padding-padding-byte-id*:
assumes *eq-padding*: *eq-padding bs bs′*
assumes *is-padding*: *is-padding-byte i*
shows *bs!i = bs′!i*
⟨*proof*⟩

lemma *eq-upto-padding-value-byte-id*:
assumes *eq-padding*: *eq-upto-padding bs bs′*
assumes *is-value*: *is-value-byte i*
shows *bs!i = bs′!i*
⟨*proof*⟩

lemma *eq-upto-padding-neq-is-padding-byte*:
assumes *eq-upto-padding*: *eq-upto-padding bs bs′*
assumes *i-bound*: *i < length bs*
assumes *neq*: *bs!i ≠ bs′!i*
shows *is-padding-byte i*
⟨*proof*⟩

lemma *eq-padding-neq-is-value-byte*:
assumes *eq-upto-padding*: *eq-padding bs bs′*
assumes *i-bound*: *i < length bs*
assumes *neq*: *bs!i ≠ bs′!i*
shows *is-value-byte i*
⟨*proof*⟩

lemma *padding-eq-complement*:
assumes *eq-padding*: *eq-padding bs bs′*
assumes *eq-upto-padding*: *eq-upto-padding bs bs′*
shows *bs = bs′*
⟨*proof*⟩

end

locale *padding-lense = complement-padding +*
assumes *double-update*: *upd bs (upd bs′ s) = upd bs s*
assumes *access-update*: *acc (upd bs s) bs′ = acc (upd bs s′) bs′*
assumes *update-access*: *upd (acc s bs) s = s*
assumes *access-size*: *acc s (take sz bs) = acc s bs*
assumes *access-result-size*: *length (acc s bs) = sz*
assumes *update-size*: *upd (take sz bs) = upd bs*
begin

lemma *update-size-ext*: *upd (take sz bs) s = upd bs s*

⟨*proof*⟩

lemma *update-access-append*: *upd* ((*acc s bs*)@*bs'*) *s* = *s* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *field-access-eq-padding1*: *length bs* = *sz* ⟹ *eq-padding* (*acc v bs*) *bs*
⟨*proof*⟩

lemma *field-access-eq-padding2*: *length bs* = *sz* ⟹ *eq-padding* (*acc v2 bs*) (*acc v2 bs*)
⟨*proof*⟩

lemma *field-access-eq-upto-padding*: *length bs* = *sz* ⟹ *length bs* = *sz* ⟹
eq-upto-padding (*acc v bs*) (*acc v bs'*)
⟨*proof*⟩

lemma *padding-byte-to-eq-padding-eq*:
eq-padding bs bs'
if
length bs = *sz*
length bs' = *sz*
(∀ *i*. *is-padding-byte i* ⟶ *bs* ! *i* = *bs'* ! *i*)
⟨*proof*⟩

lemma *eq-padding-is-padding-byte-conv*:
eq-padding bs bs' ⟷ *length bs* = *sz* ∧ *length bs'* = *sz* ∧ (∀ *i*. *is-padding-byte i* ⟶ *bs* ! *i* = *bs'* ! *i*)
⟨*proof*⟩

lemma *value-byte-to-eq-upto-padding-eq*:
assumes *lbs*: *length bs* = *sz*
assumes *lbs'*: *length bs'* = *sz*
assumes *is-value*: ∀ *i*. *is-value-byte i* ⟶ *bs* ! *i* = *bs'* ! *i*
shows *eq-upto-padding bs bs'*
⟨*proof*⟩

lemma *eq-upto-padding-is-value-byte-conv*:
eq-upto-padding bs bs' ⟷ *length bs* = *sz* ∧ *length bs'* = *sz* ∧ (∀ *i*. *is-value-byte i* ⟶ *bs* ! *i* = *bs'* ! *i*)
⟨*proof*⟩

end

locale *wf-field-desc* = *padding-lense* (*field-access d*) (*field-update d*) (*field-sz d*) **for** *d*::'*a field-desc*

locale *field-desc-independent* =

fixes
acc::*'a* ⇒ *byte list* ⇒ *byte list* **and**
upd::*byte list* ⇒ *'a* ⇒ *'a* **and**
D:: *'a field-desc set*
assumes *fu-commutes*: $d \in D \Longrightarrow$ *fu-commutes upd* (*field-update d*)
assumes *acc-upd-old*: $d \in D \Longrightarrow$ *acc* (*field-update d bs v*) *bs'* = *acc v bs'*
assumes *acc-upd-new*: $d \in D \Longrightarrow$ *field-access d* (*upd bs' v*) *bs* = *field-access d v bs*

primrec
field-descs :: *'a xtyp-info* ⇒ *'a field-desc list* **and**
field-descs-struct :: *'a xtyp-info-struct* ⇒ *'a field-desc list* **and**
field-descs-list :: *'a xtyp-info-tuple list* ⇒ *'a field-desc list* **and**
field-descs-tuple :: *'a xtyp-info-tuple* ⇒ *'a field-desc list*
where
fd0: *field-descs* (*TypDesc algn st nm*) = *field-descs-struct st*

| *fd1*: *field-descs-struct* (*TypScalar n algn d*) = [*d*]
| *fd2*: *field-descs-struct* (*TypAggregate xs*) = *field-descs-list xs*

| *fd3*: *field-descs-list* [] = []
| *fd4*: *field-descs-list* (*x*#*xs*) = *field-descs-tuple x* @ *field-descs-list xs*

| *fd5*: *field-descs-tuple* (*DTuple t nm d*) = [*d*] @ *field-descs t*

primrec
wf-component-descs :: *'a xtyp-info* ⇒ *bool* **and**
wf-component-descs-struct :: *'a xtyp-info-struct* ⇒ *bool* **and**
wf-component-descs-list :: *'a xtyp-info-tuple list* ⇒ *bool* **and**
wf-component-descs-tuple :: *'a xtyp-info-tuple* ⇒ *bool*
where
wfc0: *wf-component-descs* (*TypDesc algn st nm*) = *wf-component-descs-struct st*

| *wfc1*: *wf-component-descs-struct* (*TypScalar n algn d*) = (*n* = *field-sz d*)
| *wfc2*: *wf-component-descs-struct* (*TypAggregate xs*) = (*wf-component-descs-list xs*)

| *wfc3*: *wf-component-descs-list* [] = *True*
| *wfc4*: *wf-component-descs-list* (*x*#*xs*) = (*wf-component-descs-tuple x* ∧ *wf-component-descs-list xs*)

| *wfc5*: *wf-component-descs-tuple* (*DTuple t nm d*) =
(*d* = *component-desc t* ∧ *wf-component-descs t*)

primrec *field-descs-independent*:: *'a field-desc list* ⇒ *bool*
where
field-descs-independent [] = *True* |
field-descs-independent (*d*#*ds*) =
(*field-desc-independent* (*field-access d*) (*field-update d*) (*set ds*) ∧

field-descs-independent ds)

primrec
component-descs-independent :: *'a xtyp-info* ⇒ *bool* **and**
component-descs-independent-struct :: *'a xtyp-info-struct* ⇒ *bool* **and**
component-descs-independent-list :: *'a xtyp-info-tuple list* ⇒ *bool* **and**
component-descs-independent-tuple :: *'a xtyp-info-tuple* ⇒ *bool*
where
cdi0: *component-descs-independent* (*TypDesc algn st nm*) = *component-descs-independent-struct st*

| *cdi1*: *component-descs-independent-struct* (*TypScalar n algn d*) = *True*
| *cdi2*: *component-descs-independent-struct* (*TypAggregate xs*) = (*component-descs-independent-list xs*)

| *cdi3*: *component-descs-independent-list* [] = *True*
| *cdi4*: *component-descs-independent-list* (*f*#*fs*) = (*field-descs-independent* (*toplevel-field-descs-list* (*f*#*fs*)) ∧
component-descs-independent-tuple f ∧ *component-descs-independent-list fs*)

| *cdi5*: *component-descs-independent-tuple* (*DTuple ft fn d*) = (*component-descs-independent ft*)

locale *wf-field-descs* =
fixes *D*::*'a field-desc set*
assumes *wf-desc*[*intro*]: *d* ∈ *D* ⟹ *wf-field-desc d*

lemma *wf-field-descs-union* [*simp*]: *wf-field-descs* (*D* ∪ *E*) = (*wf-field-descs D* ∧ *wf-field-descs E*)
⟨*proof*⟩

lemma *wf-field-descs-empty*[*simp*]: *wf-field-descs* {}
⟨*proof*⟩

lemma *wf-field-descs-insert* [*simp*]: *wf-field-descs* (*insert d D*) = (*wf-field-desc d* ∧ *wf-field-descs D*)
⟨*proof*⟩

definition *padding-desc*:: *nat* ⇒ *'a field-desc* **where**
padding-desc n = (⦇*field-access* = λ*v bs*. *if n* ≤ *length bs then take n bs else replicate n 0*, *field-update* = λ*bs*. *id*, *field-sz* = *n*⦈)

definition *is-padding-desc d* = (∃ *n*. *d* = *padding-desc n*)

definition *padding-tag n* = *TypDesc 0* (*TypScalar n 0* (*padding-desc n*)) *''!pad-typ''*

definition *is-padding-tag t* = (∃ *n*. *t* = *padding-tag n*)

definition *padding-field-name f* = (∃ *xs*. *f* = *CHR* ''!'' # *xs*)

lemma *padding-field-name-empty*[*simp*]: *padding-field-name* [] = *False*
⟨*proof*⟩

lemma *padding-field-name-cons*[*simp*]: *padding-field-name* (*f*#*fs*) = (*f* = *CHR* ''!'')
⟨*proof*⟩

definition *qualified-padding-field-name fs* = (∃ *f fs'*. *fs*= *f*#*fs'* ∧ *padding-field-name* (*last fs*))

lemma *qualified-pading-field-name-empty*[*simp*]: *qualified-padding-field-name* [] = *False*
⟨*proof*⟩

lemma *qualified-pading-field-name-single*[*simp*]: *qualified-padding-field-name* [*f*] = *padding-field-name f*
⟨*proof*⟩

lemma *qualified-pading-field-name-cons*[*simp*]: *qualified-padding-field-name* (*f*#*fs*) = *padding-field-name* (*last* (*f*#*fs*))
⟨*proof*⟩

primrec
wf-padding :: ('*a*, '*b*) *typ-info* ⇒ *bool* **and**
wf-padding-struct :: ('*a*, '*b*) *typ-info-struct* ⇒ *bool* **and**
wf-padding-list :: ('*a*, '*b*) *typ-info-tuple list* ⇒ *bool* **and**
wf-padding-tuple :: ('*a*, '*b*) *typ-info-tuple* ⇒ *bool*
where
wf-padding (*TypDesc algn st nm*) = *wf-padding-struct st*

| *wf-padding-struct* (*TypScalar m algn d*) = *True*
| *wf-padding-struct* (*TypAggregate xs*) = *wf-padding-list xs*

| *wf-padding-list* [] = *True*
| *wf-padding-list* (*x*#*xs*) = (*wf-padding-tuple x* ∧ *wf-padding-list xs*)

| *wf-padding-tuple* (*DTuple s f d*) = ((*padding-field-name f* ⟶ *is-padding-tag s*) ∧ *wf-padding s*)

class *xmem-type* = *mem-type* +
assumes *wf-component-descs*: *wf-component-descs* (*typ-info-t TYPE*('*a*))
assumes *component-descs-independent*: *component-descs-independent* (*typ-info-t TYPE*('*a*))

assumes *wf-field-descs*: *wf-field-descs* (*set* (*field-descs* (*typ-info-t TYPE*('*a*))))
assumes *wf-padding*: *wf-padding* (*typ-info-t TYPE*('*a*))

locale *fields-contained* =
fixes
acc::'*a* ⇒ *byte list* ⇒ *byte list* **and**
upd::*byte list* ⇒ '*a* ⇒ '*a* **and**
D:: '*a field-desc set*
assumes *access-contained*: *d* ∈ *D* ⟹ *acc s* = *acc s'* ⟹ *field-access d s* = *field-access d s'*
assumes *update-contained*: *d* ∈ *D* ⟹ ∃ *bs'*. ∀ *s'*. *acc s'* = *acc s* ⟶ *field-update d bs s'* = *upd bs' s'*

primrec
contained-field-descs :: '*a xtyp-info* ⇒ *bool* **and**
contained-field-descs-struct :: '*a xtyp-info-struct* ⇒ *bool* **and**
contained-field-descs-list :: '*a xtyp-info-tuple list* ⇒ *bool* **and**
contained-field-descs-tuple :: '*a xtyp-info-tuple* ⇒ *bool*
where
wfd0: *contained-field-descs* (*TypDesc algn st nm*) = *contained-field-descs-struct st*

| *wfd1*: *contained-field-descs-struct* (*TypScalar n algn d*) = *True*
| *wfd2*: *contained-field-descs-struct* (*TypAggregate xs*) = *contained-field-descs-list xs*

| *wfd3*: *contained-field-descs-list* [] = *True*
| *wfd4*: *contained-field-descs-list* (*x*#*xs*) = (*contained-field-descs-tuple x* ∧ *contained-field-descs-list xs*)

| *wfd5*: *contained-field-descs-tuple* (*DTuple t nm d*) =
(*contained-field-descs t* ∧
fields-contained (*field-access d*) (*field-update d*) (*set* (*toplevel-field-descs t*)))

class *xmem-contained-type* = *xmem-type* +
assumes *contained-field-descs*: *contained-field-descs* (*typ-info-t* (*TYPE*('*a*)))

In order to construct a type of class *mem-type* we usually construct extended type-info for class *xmem-contained-type*. The extendend type-info is better behaved with respect to composition of sub-structures / arrays.

end

theory *CTypes*
imports
CTypesDefs HOL−Eisbach.Eisbach-Tools

begin

11.10 *super-update-bs*

lemma *length-super-update-bs* [*simp*]:
n + length v ≤ length bs ⟹ length (super-update-bs v bs n) = length bs
⟨*proof*⟩

lemma *drop-super-update-bs*:
⟦ k ≤ n; n ≤ length bs ⟧ ⟹ drop k (super-update-bs v bs n) = super-update-bs v (drop k bs) (n − k)
⟨*proof*⟩

lemma *drop-super-update-bs2*:
⟦ n ≤ length bs; n + length v ≤ k ⟧ ⟹
drop k (super-update-bs v bs n) = drop k bs
⟨*proof*⟩

lemma *take-super-update-bs*:
⟦ k ≤ n; n ≤ length bs ⟧ ⟹ take k (super-update-bs v bs n) = take k bs
⟨*proof*⟩

lemma *take-super-update-bs2*:
⟦ n ≤ length bs; n + length v ≤ k ⟧ ⟹
take k (super-update-bs v bs n) = super-update-bs v (take k bs) n
⟨*proof*⟩

lemma *take-drop-super-update-bs*:
length v = n ⟹ m ≤ length bs ⟹ take n (drop m (super-update-bs v bs m)) = v
⟨*proof*⟩

lemma *super-update-bs-take-drop*:
n + m ≤ length bs ⟹ super-update-bs (take m (drop n bs)) bs n = bs
⟨*proof*⟩

lemma *super-update-bs-same-length*: *length bs = length xbs ⟹ super-update-bs bs xbs 0 = bs*
⟨*proof*⟩

lemma *super-update-bs-append-drop-first*:
length xbs = m ⟹ n + length bs ≤ m ⟹ drop m (super-update-bs bs (xbs @ ybs) n) = ybs
⟨*proof*⟩

lemma *super-update-bs-append-take-first*:
length xbs = m ⟹ n + length bs ≤ m ⟹ take m (super-update-bs bs (xbs @ ybs) n) = (super-update-bs bs xbs n)
⟨*proof*⟩

lemma *super-update-bs-append-drop-second*:
length xbs = *m* ⟹ *m* ≤ *n* ⟹
drop m (*super-update-bs bs* (*xbs* @ *ybs*) *n*) = (*super-update-bs bs ybs* (*n* − *m*))
⟨*proof*⟩

lemma *super-update-bs-append-take-second*:
length xbs = *m* ⟹ *m* ≤ *n* ⟹
take m (*super-update-bs bs* (*xbs* @ *ybs*) *n*) = *xbs*
⟨*proof*⟩

lemma *super-update-bs-length*: *length bs* + *n* ≤ *length xbs* ==> *length* (*super-update-bs bs xbs n*) = *length xbs*
⟨*proof*⟩

lemma *super-update-bs-append2*: *length xbs* ≤ *n* ⟹
super-update-bs bs (*xbs* @ *ybs*) *n* = *xbs* @ *super-update-bs bs ybs* (*n* − *length xbs*)
⟨*proof*⟩

lemma *super-update-bs-append1*: *n* + *length bs* ≤ *length xbs* ⟹
super-update-bs bs (*xbs* @ *ybs*) *n* = *super-update-bs bs xbs n* @ *ybs*
⟨*proof*⟩

11.11 Rest

lemma *fu-commutes*:
fu-commutes f g ⟹ *f bs* (*g bs′ v*) = *g bs′* (*f bs v*)
⟨*proof*⟩

lemma *size-td-list-append* [*simp*]:
size-td-list (*xs*@*ys*) = *size-td-list xs* + *size-td-list ys*
⟨*proof*⟩

lemma *access-ti-append*:
⋀*bs*. *length bs* = *size-td-list* (*xs*@*ys*) ⟹
access-ti-list (*xs*@*ys*) *t bs* =
access-ti-list xs t (*take* (*size-td-list xs*) *bs*) @
access-ti-list ys t (*drop* (*size-td-list xs*) *bs*)
⟨*proof*⟩

lemma *update-ti-append* [*simp*]:
⋀*bs*. *update-ti-list* (*xs*@*ys*) *bs v* =
update-ti-list xs (*take* (*size-td-list xs*) *bs*)
(*update-ti-list ys* (*drop* (*size-td-list xs*) *bs*) *v*)
⟨*proof*⟩

lemma *update-ti-struct-t-typscalar* [*simp*]:
update-ti-struct-t (*TypScalar n algn d*) =
(λ*bs*. *if length bs* = *n then field-update d bs else id*)
⟨*proof*⟩

lemma *update-ti-list-t-empty* [*simp*]:
update-ti-list-t [] = (λ*x*. *id*)
⟨*proof*⟩

lemma *update-ti-list-t-cons* [*simp*]:
update-ti-list-t (*x*#*xs*) = (λ*bs v*.
if length bs = *size-td-tuple x* + *size-td-list xs then*
update-ti-tuple-t x (*take* (*size-td-tuple x*) *bs*)
(*update-ti-list-t xs* (*drop* (*size-td-tuple x*) *bs*) *v*) *else*
v)
⟨*proof*⟩

lemma *update-ti-append-s* [*simp*]:
⋀*bs*. *update-ti-list-t* (*xs*@*ys*) *bs v* = (
if length bs = *size-td-list xs* + *size-td-list ys then*
update-ti-list-t xs (*take* (*size-td-list xs*) *bs*)
(*update-ti-list-t ys* (*drop* (*size-td-list xs*) *bs*) *v*) *else*
v)
⟨*proof*⟩

lemma *update-ti-tuple-t-dtuple* [*simp*]:
update-ti-tuple-t (*DTuple t f d*) = *update-ti-t t*
⟨*proof*⟩

---

lemma *field-desc-empty* [*simp*]:
field-desc (*TypDesc algn* (*TypAggregate* []) *nm*) =
(| *field-access* = λ*x bs*. [],
field-update = λ*x*. *id*, *field-sz* = *0* |)
⟨*proof*⟩

lemma *export-uinfo-typdesc-simp* [*simp*]:
export-uinfo (*TypDesc algn st nm*) = *map-td field-norm* (λ-. ()) (*TypDesc algn st nm*)
⟨*proof*⟩

lemma *map-td-list-append* [*simp*]:
map-td-list f g (*xs*@*ys*) = *map-td-list f g xs* @ *map-td-list f g ys*
⟨*proof*⟩

lemma *map-td-id*:
map-td (λ*n algn*. *id*) *id t* = (*t*::('*a*, '*b*) *typ-desc*)
map-td-struct (λ*n algn*. *id*) *id st* = (*st*::('*a*, '*b*) *typ-struct*)

map-td-list (λ*n algn. id*) *id ts* = (*ts*::('*a*, '*b*) *typ-tuple list*)
map-td-tuple (λ*n algn. id*) *id x* = (*x*::('*a*, '*b*) *typ-tuple*)
⟨*proof*⟩

lemma *dt-snd-map-td-list*:
dt-snd ‘ *set* (*map-td-list f g ts*) = *dt-snd* ‘ *set ts*
⟨*proof*⟩

lemma *wf-desc-map*:
shows *wf-desc* (*map-td f g t*) = *wf-desc t* **and**
wf-desc-struct (*map-td-struct f g st*) = *wf-desc-struct st* **and**
wf-desc-list (*map-td-list f g ts*) = *wf-desc-list ts* **and**
wf-desc-tuple (*map-td-tuple f g x*) = *wf-desc-tuple x*
⟨*proof*⟩

lemma *wf-desc-list-append* [*simp*]:
wf-desc-list (*xs*@*ys*) =
(*wf-desc-list xs* ∧ *wf-desc-list ys* ∧ *dt-snd* ‘ *set xs* ∩ *dt-snd* ‘ *set ys* = {})
⟨*proof*⟩

lemma *wf-size-desc-list-append* [*simp*]:
wf-size-desc-list (*xs*@*ys*) = (*wf-size-desc-list xs* ∧ *wf-size-desc-list ys*)
⟨*proof*⟩

lemma *norm-tu-list-append* [*simp*]:
norm-tu-list (*xs*@*ys*) *bs* =
norm-tu-list xs (*take* (*size-td-list xs*) *bs*) @ *norm-tu-list ys* (*drop* (*size-td-list xs*) *bs*)
⟨*proof*⟩

lemma *wf-size-desc-gt*:
shows *wf-size-desc* (*t*::('*a*, '*b*) *typ-desc*) ⟹ 0 < *size-td t* **and**
wf-size-desc-struct st ⟹ 0 < *size-td-struct* (*st*::('*a*,'*b*) *typ-struct*) **and**
⟦ *ts* ≠ []; *wf-size-desc-list ts* ⟧ ⟹ 0 < *size-td-list* (*ts*::('*a*,'*b*) *typ-tuple list*)
and
wf-size-desc-tuple x ⟹ 0 < *size-td-tuple* (*x*::('*a*,'*b*) *typ-tuple*)
⟨*proof*⟩

lemma *field-lookup-empty* [*simp*]:
field-lookup t [] *n* = *Some* (*t*,*n*)
⟨*proof*⟩

lemma *field-lookup-tuple-empty* [*simp*]:
field-lookup-tuple x [] *n* = *None*
⟨*proof*⟩

lemma *field-lookup-list-empty* [*simp*]:
field-lookup-list ts [] *n* = *None*

⟨*proof*⟩

lemma *field-lookup-struct-empty* [*simp*]:
field-lookup-struct st [] *n* = *None*
⟨*proof*⟩

lemma *field-lookup-list-append*:
field-lookup-list (*xs*@*ys*) *f n* = (*case field-lookup-list xs f n of*
None ⇒ *field-lookup-list ys f* (*n* + *size-td-list xs*)
| *Some y* ⇒ *Some y*)
⟨*proof*⟩

lemma *field-lookup-list-None*:
f ∉ *dt-snd* ‘ *set ts* ⟹ *field-lookup-list ts* (*f*#*fs*) *m* = *None*
⟨*proof*⟩

lemma *field-lookup-list-Some*:
f ∈ *dt-snd* ‘ *set ts* ⟹ *field-lookup-list ts* [*f*] *m* ≠ *None*
⟨*proof*⟩

lemma *field-lookup-offset-le*:
shows ⋀*s m n f*. *field-lookup t f m* = *Some* ((*s*::(′*a*,′*b*) *typ-desc*),*n*) ⟹ *m* ≤ *n*
and
⋀*s m n f*. *field-lookup-struct st f m* = *Some* ((*s*::(′*a*,′*b*) *typ-desc*),*n*) ⟹ *m* ≤ *n* **and**
⋀*s m n f*. *field-lookup-list ts f m* = *Some* ((*s*::(′*a*, ′*b*) *typ-desc*),*n*) ⟹ *m* ≤ *n* **and**
⋀*s m n f*. *field-lookup-tuple x f m* = *Some* ((*s*::(′*a*, ′*b*) *typ-desc*),*n*) ⟹ *m* ≤ *n*
⟨*proof*⟩

lemma *field-lookup-offset′*:
shows ⋀*f m m′ n t′*. (*field-lookup t f m* = *Some* ((*t′*::(′*a*,′*b*) *typ-desc*),*m* + *n*)) =
(*field-lookup t f m′* = *Some* (*t′*,*m′* + *n*)) **and**
⋀*f m m′ n t′*. (*field-lookup-struct st f m* = *Some* ((*t′*::(′*a*,′*b*) *typ-desc*),*m* + *n*)) =
(*field-lookup-struct st f m′* = *Some* (*t′*,*m′* + *n*)) **and**
⋀*f m m′ n t′*. (*field-lookup-list ts f m* = *Some* ((*t′*::(′*a*,′*b*) *typ-desc*),*m* + *n*)) =
(*field-lookup-list ts f m′* = *Some* (*t′*,*m′* + *n*)) **and**
⋀*f m m′ n t′*. (*field-lookup-tuple x f m* = *Some* ((*t′*::(′*a*,′*b*) *typ-desc*),*m* + *n*)) =
(*field-lookup-tuple x f m′* = *Some* (*t′*,*m′* + *n*))
⟨*proof*⟩

lemma *field-lookup-offset*:
(*field-lookup t f m* = *Some* (*t′*,*m* + *n*)) = (*field-lookup t f 0* = *Some* (*t′*,*n*))
⟨*proof*⟩

lemma *field-lookup-offset2*:
field-lookup t f m = Some (t',n) ⟹ field-lookup t f 0 = Some (t',n − m)
⟨*proof*⟩

lemma *field-lookup-list-offset*:
(field-lookup-list ts f m = Some (t',m + n)) = (field-lookup-list ts f 0 = Some (t',n))
⟨*proof*⟩

lemma *field-lookup-list-offset2*:
field-lookup-list ts f m = Some (t',n) ⟹ field-lookup-list ts f 0 = Some (t',n − m)
⟨*proof*⟩

lemma *field-lookup-list-offset3*:
field-lookup-list ts f 0 = Some (t',n) ⟹ field-lookup-list ts f m = Some (t',m + n)
⟨*proof*⟩

lemma *field-lookup-list-offsetD*:
⟦ field-lookup-list ts f 0 = Some (s,k); field-lookup-list ts f m = Some (t,n) ⟧ ⟹ s=t ∧ n=m+k
⟨*proof*⟩

lemma *field-lookup-offset-None*:
(field-lookup t f m = None) = (field-lookup t f 0 = None)
⟨*proof*⟩

lemma *field-lookup-list-offset-None*:
(field-lookup-list ts f m = None) = (field-lookup-list ts f 0 = None)
⟨*proof*⟩

lemma *map-td-size* [*simp*]:
shows *size-td (map-td f g t) = size-td t* **and**
size-td-struct (map-td-struct f g st) = size-td-struct st **and**
size-td-list (map-td-list f g ts) = size-td-list ts **and**
size-td-tuple (map-td-tuple f g x) = size-td-tuple x
⟨*proof*⟩

lemma *td-set-map-td1*:
(s, n) ∈ td-set t k ⟹ (map-td f g s, n) ∈ td-set (map-td f g t) k **and**
(s, n) ∈ td-set-struct st k ⟹ (map-td f g s, n) ∈ td-set-struct (map-td-struct f g st) k **and**
(s, n) ∈ td-set-list ts k ⟹ (map-td f g s, n) ∈ td-set-list (map-td-list f g ts) k **and**
(s, n) ∈ td-set-tuple td k ⟹ (map-td f g s, n) ∈ td-set-tuple (map-td-tuple f g td) k
⟨*proof*⟩

lemma *size-td-tuple-dt-fst*:
size-td-tuple p = size-td (dt-fst p)
⟨*proof*⟩

lemma *td-set-map-td2*:
(s', n) ∈ td-set (map-td f g t) k ⟹ ∃ s. (s, n) ∈ td-set t k ∧ s' = map-td f g s **and**
(s', n) ∈ td-set-struct (map-td-struct f g st) k ⟹ ∃ s. (s, n) ∈ td-set-struct st k ∧ s' = map-td f g s **and**
(s', n) ∈ td-set-list (map-td-list f g ts) k ⟹ ∃ s. (s, n) ∈ td-set-list ts k ∧ s' = map-td f g s **and**
(s', n) ∈ td-set-tuple (map-td-tuple f g td) k ⟹ ∃ s. (s, n) ∈ td-set-tuple td k ∧ s' = map-td f g s
⟨*proof*⟩

lemma *td-set-offset1*:
(s, n) ∈ td-set t k ⟹ (s, n + l) ∈ td-set t (k + l) **and**
(s, n) ∈ td-set-struct st k ⟹ (s, n + l) ∈ td-set-struct st (k + l) **and**
(s, n) ∈ td-set-list xs k ⟹ (s, n + l) ∈ td-set-list xs (k + l) **and**
(s, n) ∈ td-set-tuple td k ⟹ (s, n + l) ∈ td-set-tuple td (k + l)
⟨*proof*⟩

lemma *td-set-offset2*:
(s, n + l) ∈ td-set t (k + l) ⟹ (s, n) ∈ td-set t k **and**
(s, n + l) ∈ td-set-struct st (k + l) ⟹ (s, n) ∈ td-set-struct st k **and**
(s, n + l) ∈ td-set-list xs (k + l) ⟹ (s, n) ∈ td-set-list xs k **and**
(s, n + l) ∈ td-set-tuple td (k + l) ⟹ (s, n) ∈ td-set-tuple td k
⟨*proof*⟩

lemma *td-set-offset-conv*: *(s, n) ∈ td-set t k ⟷ (s, n + l) ∈ td-set t (k + l)*
⟨*proof*⟩

lemma *td-set-offset-0-conv*: *(s, n + k) ∈ td-set t k ⟷ (s, n) ∈ td-set t 0*
⟨*proof*⟩

lemma *td-set-offset-le'*:
∀ s m n. ((s::('a,'b) typ-desc),n) ∈ td-set t m ⟶ m ≤ n
∀ s m n. ((s::('a,'b) typ-desc),n) ∈ td-set-struct st m ⟶ m ≤ n
∀ s m n. ((s::('a,'b) typ-desc),n) ∈ td-set-list ts m ⟶ m ≤ n
∀ s m n. ((s::('a,'b) typ-desc),n) ∈ td-set-tuple x m ⟶ m ≤ n
⟨*proof*⟩

lemma *td-set-offset-0-conv'*: *(s, n) ∈ td-set t k ⟷ (∃ n'. (s, n') ∈ td-set t 0 ∧ n = n' + k)*
⟨*proof*⟩

lemma *td-set-list-set-td-set1*: *(s, n) ∈ td-set-list ts k ⟹*
(∃ t n'. t ∈ set ts ∧ (s, n') ∈ td-set (dt-fst t) 0)
⟨*proof*⟩

lemma *export-uinfo-size* [*simp*]:
size-td (*export-uinfo t*) = *size-td* (*t*::('*a*,'*b*) *typ-info*)
⟨*proof*⟩

lemma (**in** *c-type*) *typ-uinfo-size* [*simp*]:
size-td (*typ-uinfo-t TYPE*('*a*)) = *size-td* (*typ-info-t TYPE*('*a*))
⟨*proof*⟩

lemma *wf-size-desc-map*:
shows *wf-size-desc* (*map-td f g t*) = *wf-size-desc t* **and**
wf-size-desc-struct (*map-td-struct f g st*) = *wf-size-desc-struct st* **and**
wf-size-desc-list (*map-td-list f g ts*) = *wf-size-desc-list ts* **and**
wf-size-desc-tuple (*map-td-tuple f g x*) = *wf-size-desc-tuple x*
⟨*proof*⟩

lemma *map-td-flr-Some* [*simp*]:
map-td-flr f g (*Some* (*t*,*n*)) = *Some* (*map-td f g t*,*n*)
⟨*proof*⟩

lemma *map-td-flr-None* [*simp*]:
map-td-flr f g None = *None*
⟨*proof*⟩

lemma *field-lookup-map*:
shows ⋀*f m s*. *field-lookup t f m* = *s* ⟹
field-lookup (*map-td fupd g t*) *f m* = *map-td-flr fupd g s* **and**
⋀*f m s*. *field-lookup-struct st f m* = *s* ⟹
field-lookup-struct (*map-td-struct fupd g st*) *f m* = *map-td-flr fupd g s*
and
⋀*f m s*. *field-lookup-list ts f m* = *s* ⟹
field-lookup-list (*map-td-list fupd g ts*) *f m* = *map-td-flr fupd g s* **and**
⋀*f m s*. *field-lookup-tuple x f m* = *s* ⟹
field-lookup-tuple (*map-td-tuple fupd g x*) *f m* = *map-td-flr fupd g s*
⟨*proof*⟩

lemma *field-lookup-export-uinfo-Some*:
field-lookup (*t*::('*a*,'*b*) *typ-info*) *f m* = *Some* (*s*,*n*) ⟹
field-lookup (*export-uinfo t*) *f m* = *Some* (*export-uinfo s*,*n*)
⟨*proof*⟩

lemma *field-lookup-struct-export-Some*:
field-lookup-struct (*st*::('*a*,'*b*) *typ-struct*) *f m* = *Some* (*s*,*n*) ⟹
field-lookup-struct (*map-td-struct fupd g st*) *f m* = *Some* (*map-td fupd g s*,*n*)
⟨*proof*⟩

lemma *field-lookup-struct-export-None*:
field-lookup-struct (*st*::('*a*, '*b*) *typ-struct*) *f m* = *None* ⟹
field-lookup-struct (*map-td-struct fupd g st*) *f m* = *None*

⟨*proof*⟩

lemma *field-lookup-list-export-Some*:
field-lookup-list (*ts*::('*a*, '*b*) *typ-tuple list*) *f m* = *Some* (*s*,*n*) ⟹
field-lookup-list (*map-td-list fupd g ts*) *f m* = *Some* (*map-td fupd g s*,*n*)
⟨*proof*⟩

lemma *field-lookup-list-export-None*:
field-lookup-list (*ts*::('*a*, '*b*) *typ-tuple list*) *f m* = *None* ⟹
field-lookup-list (*map-td-list fupd g ts*) *f m* = *None*
⟨*proof*⟩

lemma *field-lookup-tuple-export-Some*:
field-lookup-tuple (*x*::('*a*, '*b*) *typ-tuple*) *f m* = *Some* (*s*,*n*) ⟹
field-lookup-tuple (*map-td-tuple fupd g x*) *f m* = *Some* (*map-td fupd g s*,*n*)
⟨*proof*⟩

lemma *field-lookup-tuple-export-None*:
field-lookup-tuple (*x*::('*a*, '*b*) *typ-tuple*) *f m* = *None* ⟹
field-lookup-tuple (*map-td-tuple fupd g x*) *f m* = *None*
⟨*proof*⟩

lemma *import-flr-Some* [*simp*]:
import-flr f g (*Some* (*map-td f g t*,*n*)) (*Some* (*t*,*n*))
⟨*proof*⟩

lemma *import-flr-None* [*simp*]:
import-flr f g None None
⟨*proof*⟩

lemma *field-lookup-export-uinfo-rev''*:
⋀*f m s*. *field-lookup* (*map-td fupd g t*) *f m* = *s* ⟹
import-flr fupd g s ((*field-lookup t f m*)::('*a*,'*b*) *flr*)
⋀*f m s*. *field-lookup-struct* (*map-td-struct fupd g st*) *f m* = *s* ⟹
import-flr fupd g s ((*field-lookup-struct st f m*)::('*a*,'*b*) *flr*)
⋀*f m s*. *field-lookup-list* (*map-td-list fupd g ts*) *f m* = *s* ⟹
import-flr fupd g s ((*field-lookup-list ts f m*)::('*a*,'*b*) *flr*)
⋀*f m s*. *field-lookup-tuple* (*map-td-tuple fupd g x*) *f m* = *s* ⟹
import-flr fupd g s ((*field-lookup-tuple x f m*)::('*a*,'*b*) *flr*)
⟨*proof*⟩

lemma *field-lookup-export-uinfo-rev'*:
(∀*f m s*. *field-lookup* (*map-td fupd g t*) *f m* = *s* ⟶
import-flr fupd g s ((*field-lookup t f m*)::('*a*,'*b*) *flr*)) ∧
(∀*f m s*. *field-lookup-struct* (*map-td-struct fupd g st*) *f m* = *s* ⟶
import-flr fupd g s ((*field-lookup-struct st f m*)::('*a*,'*b*) *flr*)) ∧
(∀*f m s*. *field-lookup-list* (*map-td-list fupd g ts*) *f m* = *s* ⟶

import-flr fupd g s ((*field-lookup-list ts f m*)::('*a*,'*b*) *flr*)) ∧
(∀ *f m s*. *field-lookup-tuple* (*map-td-tuple fupd g x*) *f m* = *s* ⟶
import-flr fupd g s ((*field-lookup-tuple x f m*)::('*a*,'*b*) *flr*))
⟨*proof*⟩

lemma *field-lookup-export-uinfo-Some-rev*:
field-lookup (*export-uinfo* (*t*::('*a*,'*b*) *typ-info*)) *f m* = *Some* (*s*,*n*) ⟹
∃ *k*. *field-lookup t f m* = *Some* (*k*,*n*) ∧ *export-uinfo k* = *s*
⟨*proof*⟩

lemma (**in** *c-type*) *field-lookup-uinfo-Some-rev*:
field-lookup (*typ-uinfo-t* (*TYPE*('*a*))) *f m* = *Some* (*s*,*n*) ⟹
∃ *k*. *field-lookup* (*typ-info-t* (*TYPE*('*a*))) *f m* = *Some* (*k*,*n*) ∧ *export-uinfo k* = *s*
⟨*proof*⟩

lemma (**in** *c-type*) *field-lookup-offset-untyped-eq* [*simp*]:
field-lookup (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*,*n*) ⟹
field-offset-untyped (*typ-uinfo-t TYPE*('*a*)) *f* = *n*
⟨*proof*⟩

lemma (**in** *c-type*) *field-lookup-offset-eq* [*simp*]:
field-lookup (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*,*n*) ⟹
field-offset TYPE('*a*) *f* = *n*
⟨*proof*⟩

lemma *field-offset-self* [*simp*]:
field-offset t [] = *0*
⟨*proof*⟩

lemma (**in** *c-type*) *field-ti-self* [*simp*]:
field-ti TYPE('*a*) [] = *Some* (*typ-info-t TYPE*('*a*))
⟨*proof*⟩

lemma (**in** *c-type*) *field-size-self* [*simp*]:
field-size TYPE('*a*) [] = *size-td* (*typ-info-t TYPE*('*a*))
⟨*proof*⟩

lemma *field-lookup-prefix-None''*:
(∀ *f g m*. *field-lookup* (*t*::('*a*,'*b*) *typ-desc*) *f m* = *None* ⟶ *field-lookup t* (*f*@*g*) *m* = *None*)
(∀ *f g m*. *field-lookup-struct* (*st*::('*a*,'*b*) *typ-struct*) *f m* = *None* ⟶ *f* ≠ [] ⟶
field-lookup-struct st (*f*@*g*) *m* = *None*)
(∀ *f g m*. *field-lookup-list* (*ts*::('*a*,'*b*) *typ-tuple list*) *f m* = *None* ⟶ *f* ≠ [] ⟶
field-lookup-list ts (*f*@*g*) *m* = *None*)
(∀ *f g m*. *field-lookup-tuple* (*x*::('*a*,'*b*) *typ-tuple*) *f m* = *None* ⟶ *f* ≠ [] ⟶
field-lookup-tuple x (*f*@*g*) *m* = *None*)

⟨*proof*⟩

lemma *field-lookup-prefix-None′*:
(∀*f g m*. *field-lookup* (*t*::('*a*,'*b*) *typ-desc*) *f m* = *None* ⟶ *field-lookup t* (*f*@*g*) *m* = *None*) ∧
(∀*f g m*. *field-lookup-struct* (*st*::('*a*,'*b*) *typ-struct*) *f m* = *None* ⟶ *f* ≠ [] ⟶ *field-lookup-struct st* (*f*@*g*) *m* = *None*) ∧
(∀*f g m*. *field-lookup-list* (*ts*::('*a*,'*b*) *typ-tuple list*) *f m* = *None* ⟶ *f* ≠ [] ⟶ *field-lookup-list ts* (*f*@*g*) *m* = *None*) ∧
(∀*f g m*. *field-lookup-tuple* (*x*::('*a*,'*b*) *typ-tuple*) *f m* = *None* ⟶ *f* ≠ [] ⟶ *field-lookup-tuple x* (*f*@*g*) *m* = *None*)
⟨*proof*⟩

lemma *field-lookup-prefix-Some″*:
(∀*f g t′ m n*. *field-lookup t f m* = *Some* ((*t′*::('*a*,'*b*) *typ-desc*),*n*) ⟶ *wf-desc t* ⟶
field-lookup t (*f*@*g*) *m* = *field-lookup t′ g n*)
(∀*f g t′ m n*. *field-lookup-struct st f m* = *Some* ((*t′*::('*a*,'*b*) *typ-desc*),*n*) ⟶ *wf-desc-struct st* ⟶
field-lookup-struct st (*f*@*g*) *m* = *field-lookup t′ g n*)
(∀*f g t′ m n*. *field-lookup-list ts f m* = *Some* ((*t′*::('*a*,'*b*) *typ-desc*),*n*) ⟶ *wf-desc-list ts* ⟶
field-lookup-list ts (*f*@*g*) *m* = *field-lookup t′ g n*)
(∀*f g t′ m n*. *field-lookup-tuple x f m* = *Some* ((*t′*::('*a*,'*b*) *typ-desc*),*n*) ⟶ *wf-desc-tuple x* ⟶
field-lookup-tuple x (*f*@*g*) *m* = *field-lookup t′ g n*)
⟨*proof*⟩

lemma *field-lookup-prefix-Some′*:
(∀*f g t′ m n*. *field-lookup t f m* = *Some* ((*t′*::('*a*, '*b*) *typ-desc*),*n*) ⟶ *wf-desc t* ⟶
field-lookup t (*f*@*g*) *m* = *field-lookup t′ g n*) ∧
(∀*f g t′ m n*. *field-lookup-struct st f m* = *Some* ((*t′*::('*a*,'*b*) *typ-desc*),*n*) ⟶ *wf-desc-struct st* ⟶
field-lookup-struct st (*f*@*g*) *m* = *field-lookup t′ g n*) ∧
(∀*f g t′ m n*. *field-lookup-list ts f m* = *Some* ((*t′*::('*a*,'*b*) *typ-desc*),*n*) ⟶ *wf-desc-list ts* ⟶
field-lookup-list ts (*f*@*g*) *m* = *field-lookup t′ g n*) ∧
(∀*f g t′ m n*. *field-lookup-tuple x f m* = *Some* ((*t′*::('*a*,'*b*) *typ-desc*),*n*) ⟶ *wf-desc-tuple x* ⟶
field-lookup-tuple x (*f*@*g*) *m* = *field-lookup t′ g n*)
⟨*proof*⟩

lemma *field-lookup-append-eq*:
wf-desc t ⟹
field-lookup t (*f* @ *g*) *n* =
Option.bind (*field-lookup t f n*) (λ(*t′*, *m*). *field-lookup t′ g m*)
⟨*proof*⟩

lemma *field-lookup-offset-shift*:
NO-MATCH 0 m $\Longrightarrow$ *field-lookup t f 0 = Some (t', n)* $\Longrightarrow$ *field-lookup t f m = Some (t', m + n)*
⟨*proof*⟩

lemma *field-lookup-offset-shift'*:
field-lookup t f m = Some (s, k) $\Longrightarrow$ *field-lookup t f n = Some (s, k + n − m)*
⟨*proof*⟩

lemma *field-lookup-append*:
assumes *t*: *wf-desc t*
and *f*: *field-lookup t f n = Some (u, m)* **and** *g*: *field-lookup u g l = Some (v, k)*
shows *field-lookup t (f @ g) n = Some (v, k + m − l)*
⟨*proof*⟩

lemma *field-lvalue-empty-simp* [*simp*]:
Ptr &(p→[]) = p
⟨*proof*⟩

lemma *map-td-align* [*simp*]:
align-td-wo-align (map-td f g t) = align-td-wo-align (t::('a,'b) typ-desc)
align-td-wo-align-struct (map-td-struct f g st) = align-td-wo-align-struct (st::('a,'b) typ-struct)
align-td-wo-align-list (map-td-list f g ts) = align-td-wo-align-list (ts::('a,'b) typ-tuple list)
align-td-wo-align-tuple (map-td-tuple f g x) = align-td-wo-align-tuple (x::('a,'b) typ-tuple)
⟨*proof*⟩

lemma *map-td-align'* [*simp*]:
align-td (map-td f g t) = align-td (t::('a,'b) typ-desc)
align-td-struct (map-td-struct f g st) = align-td-struct (st::('a,'b) typ-struct)
align-td-list (map-td-list f g ts) = align-td-list (ts::('a,'b) typ-tuple list)
align-td-tuple (map-td-tuple f g x) = align-td-tuple (x::('a,'b) typ-tuple)
⟨*proof*⟩

lemma *typ-uinfo-align* [*simp*]:
align-td-wo-align (export-uinfo t) = align-td-wo-align (t::('a,'b) typ-info)
⟨*proof*⟩

lemma *ptr-aligned-Ptr-0* [*simp*]:
ptr-aligned NULL
⟨*proof*⟩

lemma *td-set-self* [*simp*]:
(t,m) ∈ td-set t m
⟨*proof*⟩

lemma *td-set-wf-size-desc* [*rule-format*]:
(∀ *s m n*. *wf-size-desc t* ⟶ ((*s*::('*a*,'*b*) *typ-desc*),*m*) ∈ *td-set t n* ⟶ *wf-size-desc s*)
(∀ *s m n*. *wf-size-desc-struct st* ⟶ ((*s*::('*a*,'*b*) *typ-desc*),*m*) ∈ *td-set-struct st n* ⟶ *wf-size-desc s*)
(∀ *s m n*. *wf-size-desc-list ts* ⟶ ((*s*::('*a*,'*b*) *typ-desc*),*m*) ∈ *td-set-list ts n* ⟶ *wf-size-desc s*)
(∀ *s m n*. *wf-size-desc-tuple x* ⟶ ((*s*::('*a*,'*b*) *typ-desc*),*m*) ∈ *td-set-tuple x n* ⟶ *wf-size-desc s*)
⟨*proof*⟩

lemma *td-set-size-lte'*:
(∀ *s k m*. ((*s*::('*a*,'*b*) *typ-desc*),*k*) ∈ *td-set t m* ⟶ *size s* = *size t* ∧ *s*=*t* ∧ *k*=*m* ∨ *size s* < *size t*)
(∀ *s k m*. ((*s*::('*a*,'*b*) *typ-desc*),*k*) ∈ *td-set-struct st m* ⟶ *size s* < *size st*)
(∀ *s k m*. ((*s*::('*a*,'*b*) *typ-desc*),*k*) ∈ *td-set-list xs m* ⟶ *size s* < *size-list* (*size-dt-tuple size* (λ-. *0*) (λ-. *0*)) *xs*)
(∀ *s k m*. ((*s*::('*a*,'*b*) *typ-desc*),*k*) ∈ *td-set-tuple x m* ⟶ *size s* < *size-dt-tuple size* (λ-. *0*) (λ-. *0*) *x*)
⟨*proof*⟩

lemma *td-set-size-lte*:
(*s*,*k*) ∈ *td-set t m* ⟹ *size s* = *size t* ∧ *s*=*t* ∧ *k*=*m* ∨
size s < *size t*
⟨*proof*⟩

lemma *td-set-struct-size-lte*:
(*s*,*k*) ∈ *td-set-struct st m* ⟹ *size s* < *size st*
⟨*proof*⟩

lemma *td-set-list-size-lte*:
(*s*,*k*) ∈ *td-set-list ts m* ⟹ *size s* < *size-list* (*size-dt-tuple size* (λ-. *0*) (λ-. *0*)) *ts*
⟨*proof*⟩

lemma *td-aggregate-not-in-td-set-list* [*simp*]:
¬ (*TypDesc algn* (*TypAggregate xs*) *tn*,*k*) ∈ *td-set-list xs m*
⟨*proof*⟩

lemma *sub-size-td*:
(*s*::('*a*,'*b*) *typ-desc*) ≤ *t* ⟹ *size s* ≤ *size t*
⟨*proof*⟩

lemma *sub-tag-antisym*:
⟦ (*s*::('*a*,'*b*) *typ-desc*) ≤ *t*; *t* ≤ *s* ⟧ ⟹ *s*=*t*
⟨*proof*⟩

lemma *sub-tag-refl*:
(*s*::('*a*,'*b*) *typ-desc*) ≤ *s*
⟨*proof*⟩

lemma *sub-tag-sub'*:
$\forall s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in td\text{-}set\ t\ m \longrightarrow td\text{-}set\ s\ n \subseteq td\text{-}set\ t\ m$
$\forall s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in td\text{-}set\text{-}struct\ ts\ m \longrightarrow td\text{-}set\ s\ n \subseteq td\text{-}set\text{-}struct\ ts\ m$
$\forall s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in td\text{-}set\text{-}list\ xs\ m \longrightarrow td\text{-}set\ s\ n \subseteq td\text{-}set\text{-}list\ xs\ m$
$\forall s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in td\text{-}set\text{-}tuple\ x\ m \longrightarrow td\text{-}set\ s\ n \subseteq td\text{-}set\text{-}tuple\ x\ m$
⟨*proof*⟩

lemma *sub-tag-sub*:
$(s,n) \in td\text{-}set\ t\ m \Longrightarrow td\text{-}set\ s\ n \subseteq td\text{-}set\ t\ m$
⟨*proof*⟩

lemma *td-set-fst*:
$\forall m\ n.\ fst\ `\ td\text{-}set\ (s::('a,\ 'b)\ typ\text{-}desc)\ m = fst\ `\ td\text{-}set\ s\ n$
$\forall m\ n.\ fst\ `\ td\text{-}set\text{-}struct\ (st::('a,'b)\ typ\text{-}struct)\ m = fst\ `\ td\text{-}set\text{-}struct\ st\ n$
$\forall m\ n.\ fst\ `\ td\text{-}set\text{-}list\ (xs::('a,\ 'b)\ typ\text{-}tuple\ list)\ m = fst\ `\ td\text{-}set\text{-}list\ xs\ n$
$\forall m\ n.\ fst\ `\ td\text{-}set\text{-}tuple\ (x::('a,\ 'b)\ typ\text{-}tuple)\ m = fst\ `\ td\text{-}set\text{-}tuple\ x\ n$
⟨*proof*⟩

lemma *sub-tag-trans*:
$[\![\ (s::('a,'b)\ typ\text{-}desc) \le t;\ t \le u\ ]\!] \Longrightarrow s \le u$
⟨*proof*⟩

instantiation *typ-desc* :: (*type*, *type*) *order*
begin
instance
⟨*proof*⟩
end

lemma *td-set-offset-size''*:
$\forall s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in td\text{-}set\ t\ m \longrightarrow size\text{-}td\ s + (n - m) \le size\text{-}td\ t$
$\forall s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in td\text{-}set\text{-}struct\ st\ m \longrightarrow size\text{-}td\ s + (n - m) \le size\text{-}td\text{-}struct\ st$
$\forall s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in td\text{-}set\text{-}list\ ts\ m \longrightarrow size\text{-}td\ s + (n - m) \le size\text{-}td\text{-}list\ ts$
$\forall s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in td\text{-}set\text{-}tuple\ x\ m \longrightarrow size\text{-}td\ s + (n - m) \le size\text{-}td\text{-}tuple\ x$
⟨*proof*⟩

lemma *td-set-offset-size'*:
$(\forall s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in td\text{-}set\ t\ m \longrightarrow size\text{-}td\ s + (n - m) \le size\text{-}td\ t) \wedge$
$(\forall s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in td\text{-}set\text{-}struct\ st\ m \longrightarrow size\text{-}td\ s + (n - m) \le size\text{-}td\text{-}struct\ st) \wedge$

$(\forall\, s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in td\text{-}set\text{-}list\ ts\ m \longrightarrow size\text{-}td\ s + (n - m) \leq size\text{-}td\text{-}list\ ts) \wedge$
$(\forall\, s\ m\ n.\ ((s::('a,'b)\ typ\text{-}desc),n) \in td\text{-}set\text{-}tuple\ x\ m \longrightarrow size\text{-}td\ s + (n - m) \leq size\text{-}td\text{-}tuple\ x)$
⟨*proof*⟩

lemma *td-set-offset-size*:
$(s,n) \in td\text{-}set\ t\ 0 \Longrightarrow size\text{-}td\ s + n \leq size\text{-}td\ t$
⟨*proof*⟩

lemma *td-set-struct-offset-size*:
$(s,n) \in td\text{-}set\text{-}struct\ st\ m \Longrightarrow size\text{-}td\ s + (n - m) \leq size\text{-}td\text{-}struct\ st$
⟨*proof*⟩

lemma *td-set-list-offset-size*:
$(s,n) \in td\text{-}set\text{-}list\ ts\ 0 \Longrightarrow size\text{-}td\ s + n \leq size\text{-}td\text{-}list\ ts$
⟨*proof*⟩

lemma *td-set-offset-size-m*:
$(s,n) \in td\text{-}set\ t\ m \Longrightarrow size\text{-}td\ s + (n - m) \leq size\text{-}td\ t$
⟨*proof*⟩

lemma *td-set-list-offset-size-m*:
$(s,n) \in td\text{-}set\text{-}list\ t\ m \Longrightarrow size\text{-}td\ s + (n - m) \leq size\text{-}td\text{-}list\ t$
⟨*proof*⟩

lemma *td-set-tuple-offset-size-m*:
$(s,n) \in td\text{-}set\text{-}tuple\ t\ m \Longrightarrow size\text{-}td\ s + (n - m) \leq size\text{-}td\text{-}tuple\ t$
⟨*proof*⟩

lemma *td-set-list-offset-le*:
$(s,n) \in td\text{-}set\text{-}list\ ts\ m \Longrightarrow m \leq n$
⟨*proof*⟩

lemma *td-set-tuple-offset-le*:
$(s,n) \in td\text{-}set\text{-}tuple\ ts\ m \Longrightarrow m \leq n$
⟨*proof*⟩

lemma *field-of-self* [*simp*]:
field-of 0 t t
⟨*proof*⟩

lemma *td-set-export-uinfo'*:
$\forall f\ m\ n\ s.\ ((s::('a,'b)\ typ\text{-}info),n) \in td\text{-}set\ t\ m \longrightarrow$
$(export\text{-}uinfo\ s,n) \in td\text{-}set\ (export\text{-}uinfo\ t)\ m$
$\forall f\ m\ n\ s.\ ((s::('a,'b)\ typ\text{-}info),n) \in td\text{-}set\text{-}struct\ st\ m \longrightarrow$
$(export\text{-}uinfo\ s,n) \in td\text{-}set\text{-}struct\ (map\text{-}td\text{-}struct\ field\text{-}norm\ (\lambda\text{-}.\ ())\ st)\ m$

$\forall f\ m\ n\ s.\ ((s::('a,'b)\ typ\text{-}info),n) \in td\text{-}set\text{-}list\ ts\ m \longrightarrow$
$(export\text{-}uinfo\ s,n) \in td\text{-}set\text{-}list\ (map\text{-}td\text{-}list\ field\text{-}norm\ (\lambda\text{-}.\ ())\ ts)\ m$
$\forall f\ m\ n\ s.\ ((s::('a,'b)\ typ\text{-}info),n) \in td\text{-}set\text{-}tuple\ x\ m \longrightarrow$
$(export\text{-}uinfo\ s,n) \in td\text{-}set\text{-}tuple\ (map\text{-}td\text{-}tuple\ field\text{-}norm\ (\lambda\text{-}.\ ())\ x)\ m$
⟨*proof*⟩

lemma *td-set-export-uinfo*:
$(\forall f\ m\ n\ s.\ ((s::('a,'b)\ typ\text{-}info),n) \in td\text{-}set\ t\ m \longrightarrow$
$(export\text{-}uinfo\ s,n) \in td\text{-}set\ (export\text{-}uinfo\ t)\ m) \wedge$
$(\forall f\ m\ n\ s.\ ((s::('a,'b)\ typ\text{-}info),n) \in td\text{-}set\text{-}struct\ st\ m \longrightarrow$
$(export\text{-}uinfo\ s,n) \in td\text{-}set\text{-}struct\ (map\text{-}td\text{-}struct\ field\text{-}norm\ (\lambda\text{-}.\ ())\ st)\ m) \wedge$
$(\forall f\ m\ n\ s.\ ((s::('a,'b)\ typ\text{-}info),n) \in td\text{-}set\text{-}list\ ts\ m \longrightarrow$
$(export\text{-}uinfo\ s,n) \in td\text{-}set\text{-}list\ (map\text{-}td\text{-}list\ field\text{-}norm\ (\lambda\text{-}.\ ())\ ts)\ m) \wedge$
$(\forall f\ m\ n\ s.\ ((s::('a,'b)\ typ\text{-}info),n) \in td\text{-}set\text{-}tuple\ x\ m \longrightarrow$
$(export\text{-}uinfo\ s,n) \in td\text{-}set\text{-}tuple\ (map\text{-}td\text{-}tuple\ field\text{-}norm\ (\lambda\text{-}.\ ())\ x)\ m)$
⟨*proof*⟩

lemma *td-set-export-uinfoD*:
$(s,n) \in td\text{-}set\ t\ m \Longrightarrow (export\text{-}uinfo\ s,n) \in td\text{-}set\ (export\text{-}uinfo\ t)\ m$
⟨*proof*⟩

lemma *td-set-field-lookup''*:
$\forall s\ m\ n.\ wf\text{-}desc\ t \longrightarrow (((s::('a,'b)\ typ\text{-}desc),m + n) \in td\text{-}set\ t\ m \longrightarrow$
$(\exists f.\ field\text{-}lookup\ t\ f\ m = Some\ (s,m+n)))$
$\forall s\ m\ n.\ wf\text{-}desc\text{-}struct\ st \longrightarrow (((s::('a,'b)\ typ\text{-}desc),m + n) \in td\text{-}set\text{-}struct\ st\ m$
$\longrightarrow$
$(\exists f.\ field\text{-}lookup\text{-}struct\ st\ f\ m = Some\ (s,m+n)))$
$\forall s\ m\ n.\ wf\text{-}desc\text{-}list\ ts \longrightarrow (((s::('a,'b)\ typ\text{-}desc),m + n) \in td\text{-}set\text{-}list\ ts\ m \longrightarrow$
$(\exists f.\ field\text{-}lookup\text{-}list\ ts\ f\ m = Some\ (s,m+n)))$
$\forall s\ m\ n.\ wf\text{-}desc\text{-}tuple\ x \longrightarrow (((s::('a,'b)\ typ\text{-}desc),m + n) \in td\text{-}set\text{-}tuple\ x\ m \longrightarrow$
$(\exists f.\ field\text{-}lookup\text{-}tuple\ x\ f\ m = Some\ (s,m+n)))$
⟨*proof*⟩

lemma *td-set-field-lookup'*:
$(\forall s\ m\ n.\ wf\text{-}desc\ t \longrightarrow (((s::('a,'b)\ typ\text{-}desc),m + n) \in td\text{-}set\ t\ m \longrightarrow$
$(\exists f.\ field\text{-}lookup\ t\ f\ m = Some\ (s,m+n)))) \wedge$
$(\forall s\ m\ n.\ wf\text{-}desc\text{-}struct\ st \longrightarrow (((s::('a,'b)\ typ\text{-}desc),m + n) \in td\text{-}set\text{-}struct\ st$
$m \longrightarrow$
$(\exists f.\ field\text{-}lookup\text{-}struct\ st\ f\ m = Some\ (s,m+n)))) \wedge$
$(\forall s\ m\ n.\ wf\text{-}desc\text{-}list\ ts \longrightarrow (((s::('a,'b)\ typ\text{-}desc),m + n) \in td\text{-}set\text{-}list\ ts\ m \longrightarrow$
$(\exists f.\ field\text{-}lookup\text{-}list\ ts\ f\ m = Some\ (s,m+n)))) \wedge$
$(\forall s\ m\ n.\ wf\text{-}desc\text{-}tuple\ x \longrightarrow (((s::('a,'b)\ typ\text{-}desc),m + n) \in td\text{-}set\text{-}tuple\ x\ m$
$\longrightarrow$
$(\exists f.\ field\text{-}lookup\text{-}tuple\ x\ f\ m = Some\ (s,m+n))))$
⟨*proof*⟩

lemma *td-set-field-lookup-rev''*:

$\forall s\ m\ n.\ (\exists f.\ \textit{field-lookup}\ t\ f\ m = \textit{Some}\ (s,m+n)) \longrightarrow$
$((s::('a,'b)\ \textit{typ-desc}),m + n) \in \textit{td-set}\ t\ m$
$\forall s\ m\ n.\ (\exists f.\ \textit{field-lookup-struct}\ st\ f\ m = \textit{Some}\ (s,m+n)) \longrightarrow$
$((s::('a,'b)\ \textit{typ-desc}),m + n) \in \textit{td-set-struct}\ st\ m$
$\forall s\ m\ n.\ (\exists f.\ \textit{field-lookup-list}\ ts\ f\ m = \textit{Some}\ (s,m+n)) \longrightarrow$
$((s::('a,'b)\ \textit{typ-desc}),m + n) \in \textit{td-set-list}\ ts\ m$
$\forall s\ m\ n.\ (\exists f.\ \textit{field-lookup-tuple}\ x\ f\ m = \textit{Some}\ (s,m+n)) \longrightarrow$
$((s::('a,'b)\ \textit{typ-desc}),m + n) \in \textit{td-set-tuple}\ x\ m$
⟨*proof*⟩

lemma *td-set-field-lookup-rev′*:
$(\forall s\ m\ n.\ (\exists f.\ \textit{field-lookup}\ t\ f\ m = \textit{Some}\ (s,m+n)) \longrightarrow$
$((s::('a,'b)\ \textit{typ-desc}),m + n) \in \textit{td-set}\ t\ m) \wedge$
$(\forall s\ m\ n.\ (\exists f.\ \textit{field-lookup-struct}\ st\ f\ m = \textit{Some}\ (s,m+n)) \longrightarrow$
$((s::('a,'b)\ \textit{typ-desc}),m + n) \in \textit{td-set-struct}\ st\ m) \wedge$
$(\forall s\ m\ n.\ (\exists f.\ \textit{field-lookup-list}\ ts\ f\ m = \textit{Some}\ (s,m+n)) \longrightarrow$
$((s::('a,'b)\ \textit{typ-desc}),m + n) \in \textit{td-set-list}\ ts\ m) \wedge$
$(\forall s\ m\ n.\ (\exists f.\ \textit{field-lookup-tuple}\ x\ f\ m = \textit{Some}\ (s,m+n)) \longrightarrow$
$((s::('a,'b)\ \textit{typ-desc}),m + n) \in \textit{td-set-tuple}\ x\ m)$
⟨*proof*⟩

lemma *td-set-field-lookup*:
$\textit{wf-desc}\ t \Longrightarrow k \in \textit{td-set}\ t\ 0 = (\exists f.\ \textit{field-lookup}\ t\ f\ 0 = \textit{Some}\ k)$
⟨*proof*⟩

lemma *td-set-field-lookupD*:
$\textit{field-lookup}\ t\ f\ m = \textit{Some}\ k \Longrightarrow k \in \textit{td-set}\ t\ m$
⟨*proof*⟩

lemma *td-set-struct-field-lookup-structD*:
$\textit{field-lookup-struct}\ st\ f\ m = \textit{Some}\ k \Longrightarrow k \in \textit{td-set-struct}\ st\ m$
⟨*proof*⟩

lemma *field-lookup-struct-td-simp* [*simp*]:
$\textit{field-lookup-struct}\ ts\ f\ m \neq \textit{Some}\ (\textit{TypDesc}\ \textit{algn}\ ts\ nm,\ m)$
⟨*proof*⟩

lemma *td-set-list-field-lookup-listD*:
$\textit{field-lookup-list}\ xs\ f\ m = \textit{Some}\ k \Longrightarrow k \in \textit{td-set-list}\ xs\ m$
⟨*proof*⟩

lemma *td-set-tuple-field-lookup-tupleD*:
$\textit{field-lookup-tuple}\ x\ f\ m = \textit{Some}\ k \Longrightarrow k \in \textit{td-set-tuple}\ x\ m$
⟨*proof*⟩

lemma *field-lookup-offset-size″*:
$\textit{field-lookup}\ t\ f\ n = \textit{Some}\ (u,\ m) \Longrightarrow \textit{size-td}\ u + m \leq \textit{size-td}\ t + n$
⟨*proof*⟩

lemma *field-lookup-offset-size'*:
assumes *n*: *field-lookup t f 0 = Some (t',n)* **shows** *size-td t' + n ≤ size-td t*
⟨*proof*⟩

lemma *intvl-subset-of-field-lookup*:
field-lookup t f 0 = Some (u, n) ⟹
{a + of-nat n ..+ size-td u} ⊆ {a ..+ size-td t}
⟨*proof*⟩

lemma *field-lookup-wf-size-desc-gt*:
⟦ field-lookup t f n = Some (a,b); wf-size-desc t ⟧ ⟹ 0 < size-td a
⟨*proof*⟩

lemma *field-lookup-inject''*:
∀f g m s. wf-size-desc t ⟶ field-lookup (t::('a,'b) typ-desc) f m = Some s ∧ field-lookup t g m = Some s ⟶ f=g
∀f g m s. wf-size-desc-struct st ⟶ field-lookup-struct (st::('a,'b) typ-struct) f m = Some s ∧ field-lookup-struct st g m = Some s ⟶ f=g
∀f g m s. wf-size-desc-list ts ⟶ field-lookup-list (ts::('a,'b) typ-tuple list) f m = Some s ∧ field-lookup-list ts g m = Some s ⟶ f=g
∀f g m s. wf-size-desc-tuple x ⟶ field-lookup-tuple (x::('a,'b) typ-tuple) f m = Some s ∧ field-lookup-tuple x g m = Some s ⟶ f=g
⟨*proof*⟩

lemma *field-lookup-inject'*:
(∀f g m s. wf-size-desc t ⟶ field-lookup (t::('a,'b) typ-desc) f m = Some s ∧ field-lookup t g m = Some s ⟶ f=g) ∧
(∀f g m s. wf-size-desc-struct st ⟶ field-lookup-struct (st::('a,'b) typ-struct) f m = Some s ∧ field-lookup-struct st g m = Some s ⟶ f=g) ∧
(∀f g m s. wf-size-desc-list ts ⟶ field-lookup-list (ts::('a,'b) typ-tuple list) f m = Some s ∧ field-lookup-list ts g m = Some s ⟶ f=g) ∧
(∀f g m s. wf-size-desc-tuple x ⟶ field-lookup-tuple (x::('a,'b) typ-tuple) f m = Some s ∧ field-lookup-tuple x g m = Some s ⟶ f=g)
⟨*proof*⟩

lemma *field-lookup-inject*:
⟦ field-lookup (t::('a,'b) typ-desc) f m = Some s;
field-lookup t g m = Some s; wf-size-desc t ⟧ ⟹ f=g
⟨*proof*⟩

lemma *fd-cons-update-normalise*:
⟦ fd-cons-update-access d n; fd-cons-access-update d n;
fd-cons-double-update d; fd-cons-length d n ⟧ ⟹
fd-cons-update-normalise d n
⟨*proof*⟩

lemma *update-ti-t-update-ti-struct-t* [*simp*]:

update-ti-t (*TypDesc algn st tn*) = *update-ti-struct-t st*
⟨*proof*⟩

lemma *fd-cons-fd-cons-struct* [*simp*]:
fd-cons (*TypDesc algn st tn*) = *fd-cons-struct st*
⟨*proof*⟩

lemma *update-ti-struct-t-update-ti-list-t* [*simp*]:
update-ti-struct-t (*TypAggregate ts*) = *update-ti-list-t ts*
⟨*proof*⟩

lemma *fd-cons-struct-fd-cons-list* [*simp*]:
fd-cons-struct (*TypAggregate ts*) = *fd-cons-list ts*
⟨*proof*⟩

lemma *fd-cons-list-empty* [*simp*]:
fd-cons-list []
⟨*proof*⟩

lemma *fd-cons-double-update-list-append*:
⟦ *fd-cons-double-update* (*field-desc-list xs*);
fd-cons-double-update (*field-desc-list ys*);
fu-commutes (*field-update* (*field-desc-list xs*)) (*field-update* (*field-desc-list ys*))
⟧ ⟹
fd-cons-double-update (*field-desc-list* (*xs*@*ys*))
⟨*proof*⟩

lemma *fd-cons-update-access-list-append*:
⟦ *fd-cons-update-access* (*field-desc-list xs*) (*size-td-list xs*);
fd-cons-update-access (*field-desc-list ys*) (*size-td-list ys*);
fd-cons-length (*field-desc-list xs*) (*size-td-list xs*);
fd-cons-length (*field-desc-list ys*) (*size-td-list ys*) ⟧ ⟹
fd-cons-update-access (*field-desc-list* (*xs*@*ys*)) (*size-td-list* (*xs*@*ys*))
⟨*proof*⟩

lemma *min-ll*:
min (*x* + *y*) *x* = (*x*::*nat*)
⟨*proof*⟩

lemma *fd-cons-access-update-list-append*:
⟦ *fd-cons-access-update* (*field-desc-list xs*) (*size-td-list xs*);
fd-cons-access-update (*field-desc-list ys*) (*size-td-list ys*);
fu-commutes (*field-update* (*field-desc-list xs*)) (*field-update* (*field-desc-list ys*))
⟧ ⟹
fd-cons-access-update (*field-desc-list* (*xs*@*ys*)) (*size-td-list* (*xs*@*ys*))
⟨*proof*⟩

lemma *fd-cons-length-list-append*:

⟦ *fd-cons-length* (*field-desc-list xs*) (*size-td-list xs*);
fd-cons-length (*field-desc-list ys*) (*size-td-list ys*) ⟧ ⟹
fd-cons-length (*field-desc-list* (*xs*@*ys*)) (*size-td-list* (*xs*@*ys*))
⟨*proof*⟩

lemma *wf-fdp-insert*:
wf-fdp (*insert x xs*) ⟹ *wf-fdp* {*x*} ∧ *wf-fdp xs*
⟨*proof*⟩

lemma *wf-fdp-fd-cons*:
⟦ *wf-fdp X*; (*t*,*m*) ∈ *X* ⟧ ⟹ *fd-cons t*
⟨*proof*⟩

lemma *wf-fdp-fu-commutes*:
⟦ *wf-fdp X*; (*s*,*m*) ∈ *X*; (*t*,*n*) ∈ *X*; ¬ *m* ≤ *n*; ¬ *n* ≤ *m* ⟧ ⟹
fu-commutes (*field-update* (*field-desc s*)) (*field-update* (*field-desc t*))
⟨*proof*⟩

lemma *wf-fdp-fa-fu-ind*:
⟦ *wf-fdp X*; (*s*,*m*) ∈ *X*; (*t*,*n*) ∈ *X*; ¬ *m* ≤ *n*; ¬ *n* ≤ *m* ⟧ ⟹
fa-fu-ind (*field-desc s*) (*field-desc t*) (*size-td t*) (*size-td s*)
⟨*proof*⟩

lemma *wf-fdp-mono*:
⟦ *wf-fdp Y*; *X* ⊆ *Y* ⟧ ⟹ *wf-fdp X*
⟨*proof*⟩

lemma *tf0* [*simp*]:
tf-set (*TypDesc algn st nm*) = {(*TypDesc algn st nm*,[])} ∪ *tf-set-struct st*
⟨*proof*⟩

lemma *tf1* [*simp*]: *tf-set-struct* (*TypScalar m algn d*) = {}
⟨*proof*⟩

lemma *tf2* [*simp*]: *tf-set-struct* (*TypAggregate xs*) = *tf-set-list xs*
⟨*proof*⟩

lemma *tf3* [*simp*]: *tf-set-list* [] = {}
⟨*proof*⟩

lemma *tf4*: *tf-set-list* (*x*#*xs*) = *tf-set-tuple x* ∪ {*t*. *t* ∈ *tf-set-list xs* ∧ *snd t* ∉ *snd* ‘ *tf-set-tuple x*}
⟨*proof*⟩

lemma *tf4D*:
t ∈ *tf-set-list* (*x*#*xs*) ⟹ *t* ∈ (*tf-set-tuple x* ∪ *tf-set-list xs*)
⟨*proof*⟩

lemma *tf4'* [*simp*]: *wf-desc-list* (*x*#*xs*) ⟹

tf-set-list (*x*#*xs*) = *tf-set-tuple x* ∪ *tf-set-list xs*
⟨*proof*⟩

lemma *tf5* [*simp*]: *tf-set-tuple* (*DTuple t m d*) = {(*a*,*m*#*b*) | *a b*. (*a*,*b*) ∈ *tf-set t*}
⟨*proof*⟩

lemma *tf-set-self* [*simp*]:
(*t*,[]) ∈ *tf-set t*
⟨*proof*⟩

lemma *tf-set-list-mem*:
wf-desc-list ts ⟹ *DTuple t n d* ∈ *set ts* ⟹ (*t*,[*n*]) ∈ *tf-set-list ts*
⟨*proof*⟩

lemma *tf-set-list-append*:
wf-desc-list (*xs*@*ys*) ⟹ *tf-set-list* (*xs*@*ys*) = *tf-set-list xs* ∪ *tf-set-list ys*
⟨*proof*⟩

lemma *lf-set-list-append* [*simp*]:
lf-set-list (*xs*@*ys*) *fn* = *lf-set-list xs fn* ∪ *lf-set-list ys fn*
⟨*proof*⟩

lemma *ti-ind-sym*:
ti-ind X Y ⟹ *ti-ind Y X*
⟨*proof*⟩

lemma *ti-ind-sym2*:
ti-ind X Y = *ti-ind Y X*
⟨*proof*⟩

lemma *ti-ind-list* [*simp*]:
ti-ind (*X* ∪ *Y*) *Z* = (*ti-ind X Z* ∧ *ti-ind Y Z*)
⟨*proof*⟩

lemma *ti-empty* [*simp*]:
ti-ind {} *X*
⟨*proof*⟩

lemma *wf-lf-list*:
lf-fn ‘ *X* ∩ *lf-fn* ‘ *Y* = {} ⟹
wf-lf (*X* ∪ *Y*) = (*wf-lf X* ∧ *wf-lf Y* ∧ *ti-ind X Y*)
⟨*proof*⟩

lemma *wf-lf-listD*:
wf-lf (*X* ∪ *Y*) ⟹ *wf-lf X* ∧ *wf-lf Y*
⟨*proof*⟩

lemma *ti-ind-fn*:

fixes t::$('a,'b)$ *typ-info* **and**
st::$('a,'b)$ *typ-info-struct* **and**
ts::$('a,'b)$ *typ-info-tuple list* **and**
x::$('a,'b)$ *typ-info-tuple*
shows
$\forall fn.$ *ti-ind* (*lf-set t fn*) Y = *ti-ind* (*lf-set t* []) Y
$\forall fn.$ *ti-ind* (*lf-set-struct st fn*) Y = *ti-ind* (*lf-set-struct st* []) Y
$\forall fn.$ *ti-ind* (*lf-set-list ts fn*) Y = *ti-ind* (*lf-set-list ts* []) Y
$\forall fn.$ *ti-ind* (*lf-set-tuple x fn*) Y = *ti-ind* (*lf-set-tuple x* []) Y
⟨*proof*⟩

lemma *ti-ind-ld-td'*:
fixes t::$('a,'b)$ *typ-info* **and**
st::$('a,'b)$ *typ-info-struct* **and**
ts::$('a,'b)$ *typ-info-tuple list* **and**
x::$('a,'b)$ *typ-info-tuple*
shows
ti-ind (*lf-set t* []) Y ⟶ *ti-ind* {*t2d* (*t*,[])} Y
ti-ind (*lf-set-struct st* []) Y ⟶ *ti-ind* {⦇ *lf-fd* = *field-desc-struct st*, *lf-sz* = *size-td-struct st*, *lf-fn* = [] ⦈} Y
ti-ind (*lf-set-list ts* []) Y ⟶ *ti-ind* {⦇ *lf-fd* = *field-desc-list ts*, *lf-sz* = *size-td-list ts*, *lf-fn* = [] ⦈} Y
ti-ind (*lf-set-tuple x* []) Y ⟶ *ti-ind* {⦇ *lf-fd* = *field-desc-tuple x*, *lf-sz* = *size-td-tuple x*, *lf-fn* = [] ⦈} Y
⟨*proof*⟩

lemma *ti-ind-ld-td-struct*:
ti-ind (*lf-set-struct st fn*) Y ⟹
ti-ind {⦇ *lf-fd* = *field-desc-struct st*, *lf-sz* = *size-td-struct st*, *lf-fn* = [] ⦈} Y
⟨*proof*⟩

lemma *ti-ind-ld-td-list*:
ti-ind (*lf-set-list ts fn*) Y ⟹
ti-ind {⦇ *lf-fd* = *field-desc-list ts*, *lf-sz* = *size-td-list ts*, *lf-fn* = [] ⦈} Y
⟨*proof*⟩

lemma *ti-ind-ld-td-tuple*:
ti-ind (*lf-set-tuple x fn*) Y ⟹
ti-ind {⦇ *lf-fd* = *field-desc-tuple x*, *lf-sz* = *size-td-tuple x*, *lf-fn* = [] ⦈} Y
⟨*proof*⟩

lemma *ti-ind-ld'*:
fixes t::$('a,'b)$ *typ-info* **and**
st::$('a,'b)$ *typ-info-struct* **and**
ts::$('a,'b)$ *typ-info-tuple list* **and**
x::$('a,'b)$ *typ-info-tuple*
shows
ti-ind (*lf-set t* []) Y ⟶ *ti-ind* (*t2d* ‘ (*tf-set t*)) Y
ti-ind (*lf-set-struct st* []) Y ⟶ *ti-ind* (*t2d* ‘ (*tf-set-struct st*)) Y

ti-ind (*lf-set-list ts* []) *Y* ⟶ *ti-ind* (*t2d* ‘ (*tf-set-list ts*)) *Y*
ti-ind (*lf-set-tuple x* []) *Y* ⟶ *ti-ind* (*t2d* ‘ (*tf-set-tuple x*)) *Y*
⟨*proof*⟩

lemma *ti-ind-ld*:
ti-ind (*lf-set t fn*) *Y* ⟹ *ti-ind* (*t2d* ‘ (*tf-set t*)) *Y*
⟨*proof*⟩

lemma *ti-ind-ld-struct*:
ti-ind (*lf-set-struct t fn*) *Y* ⟹ *ti-ind* (*t2d* ‘ (*tf-set-struct t*)) *Y*
⟨*proof*⟩

lemma *ti-ind-ld-list*:
ti-ind (*lf-set-list t fn*) *Y* ⟹ *ti-ind* (*t2d* ‘ (*tf-set-list t*)) *Y*
⟨*proof*⟩

lemma *ti-ind-ld-tuple*:
ti-ind (*lf-set-tuple t fn*) *Y* ⟹ *ti-ind* (*t2d* ‘ (*tf-set-tuple t*)) *Y*
⟨*proof*⟩

lemma *lf-set-fn'*:
fixes *t*::('*a*,'*b*) *typ-info* **and**
st::('*a*,'*b*) *typ-info-struct* **and**
ts::('*a*,'*b*) *typ-info-tuple list* **and**
x::('*a*,'*b*) *typ-info-tuple*
shows
∀ *s fn*. *s* ∈ *lf-set t fn* ⟶ *fn* ≤ *lf-fn s*
∀ *s fn*. *s* ∈ *lf-set-struct st fn* ⟶ *fn* ≤ *lf-fn s*
∀ *s fn*. *s* ∈ *lf-set-list ts fn* ⟶ *fn* ≤ *lf-fn s*
∀ *s fn*. *s* ∈ *lf-set-tuple x fn* ⟶ *fn* ≤ *lf-fn s*
⟨*proof*⟩

lemma *lf-set-fn*:
s ∈ *lf-set* (*t*::('*a*,'*b*) *typ-info*) *fn* ⟹ *fn* ≤ *lf-fn s*
⟨*proof*⟩

lemma *ln-fn-disj*:
dt-snd x ∉ *dt-snd* ‘ *set xs* ⟹ *lf-fn* ‘ *lf-set-tuple x fn* ∩ *lf-fn* ‘ *lf-set-list xs fn* = {}
⟨*proof*⟩

lemma *wf-lf-fn*:
fixes *t*::('*a*,'*b*) *typ-info* **and**
st::('*a*,'*b*) *typ-info-struct* **and**
ts::('*a*,'*b*) *typ-info-tuple list* **and**
x::('*a*,'*b*) *typ-info-tuple*
shows
∀ *fn*. *wf-desc t* ⟶ *wf-lf* (*lf-set t fn*) = *wf-lf* (*lf-set t* [])
∀ *fn*. *wf-desc-struct st* ⟶ *wf-lf* (*lf-set-struct st fn*) = *wf-lf* (*lf-set-struct st* [])
∀ *fn*. *wf-desc-list ts* ⟶ *wf-lf* (*lf-set-list ts fn*) = *wf-lf* (*lf-set-list ts* [])

$\forall fn.\ wf\text{-}desc\text{-}tuple\ x \longrightarrow wf\text{-}lf\ (lf\text{-}set\text{-}tuple\ x\ fn) = wf\text{-}lf\ (lf\text{-}set\text{-}tuple\ x\ [])$
⟨*proof*⟩

lemma *wf-lf-fd-cons'*:
fixes t::$('a,'b)$ *typ-info* **and**
st::$('a,'b)$ *typ-info-struct* **and**
ts::$('a,'b)$ *typ-info-tuple list* **and**
x::$('a,'b)$ *typ-info-tuple*
shows
$\forall m.\ wf\text{-}lf\ (lf\text{-}set\ t\ []) \longrightarrow wf\text{-}desc\ t \longrightarrow fd\text{-}cons\ t$
$\forall m.\ wf\text{-}lf\ (lf\text{-}set\text{-}struct\ st\ []) \longrightarrow wf\text{-}desc\text{-}struct\ st \longrightarrow fd\text{-}cons\text{-}struct\ st$
$\forall m.\ wf\text{-}lf\ (lf\text{-}set\text{-}list\ ts\ []) \longrightarrow wf\text{-}desc\text{-}list\ ts \longrightarrow fd\text{-}cons\text{-}list\ ts$
$\forall m.\ wf\text{-}lf\ (lf\text{-}set\text{-}tuple\ x\ []) \longrightarrow wf\text{-}desc\text{-}tuple\ x \longrightarrow fd\text{-}cons\text{-}tuple\ x$
⟨*proof*⟩

lemma *wf-lf-fd-cons*:
$[\![\ wf\text{-}lf\ (lf\text{-}set\ t\ fn);\ wf\text{-}desc\ t\ ]\!] \Longrightarrow fd\text{-}cons\ t$
⟨*proof*⟩

lemma *wf-lf-fd-cons-struct*:
$[\![\ wf\text{-}lf\ (lf\text{-}set\text{-}struct\ t\ fn);\ wf\text{-}desc\text{-}struct\ t\ ]\!] \Longrightarrow fd\text{-}cons\text{-}struct\ t$
⟨*proof*⟩

lemma *wf-lf-fd-cons-list*:
$[\![\ wf\text{-}lf\ (lf\text{-}set\text{-}list\ t\ fn);\ wf\text{-}desc\text{-}list\ t\ ]\!] \Longrightarrow fd\text{-}cons\text{-}list\ t$
⟨*proof*⟩

lemma *wf-lf-fd-cons-tuple*:
$[\![\ wf\text{-}lf\ (lf\text{-}set\text{-}tuple\ t\ fn);\ wf\text{-}desc\text{-}tuple\ t\ ]\!] \Longrightarrow fd\text{-}cons\text{-}tuple\ t$
⟨*proof*⟩

lemma *wf-lf-fdp'*:
$\forall m.\ wf\text{-}lf\ (lf\text{-}set\ (t::('a,'b)\ typ\text{-}info)\ []) \longrightarrow wf\text{-}desc\ t \longrightarrow wf\text{-}fdp\ (tf\text{-}set\ t)$
$\forall m.\ wf\text{-}lf\ (lf\text{-}set\text{-}struct\ (st::('a,'b)\ typ\text{-}info\text{-}struct)\ []) \longrightarrow wf\text{-}desc\text{-}struct\ st \longrightarrow wf\text{-}fdp\ (tf\text{-}set\text{-}struct\ st)$
$\forall m.\ wf\text{-}lf\ (lf\text{-}set\text{-}list\ (ts::('a,'b)\ typ\text{-}info\text{-}tuple\ list)\ []) \longrightarrow wf\text{-}desc\text{-}list\ ts \longrightarrow wf\text{-}fdp\ (tf\text{-}set\text{-}list\ ts)$
$\forall m.\ wf\text{-}lf\ (lf\text{-}set\text{-}tuple\ (x::('a,'b)\ typ\text{-}info\text{-}tuple)\ []) \longrightarrow wf\text{-}desc\text{-}tuple\ x \longrightarrow wf\text{-}fdp\ (tf\text{-}set\text{-}tuple\ x)$
⟨*proof*⟩

lemma *wf-lf-fdp*:
$[\![\ wf\text{-}lf\ (lf\text{-}set\ t\ []);\ wf\text{-}desc\ t\ ]\!] \Longrightarrow wf\text{-}fdp\ (tf\text{-}set\ t)$
⟨*proof*⟩

lemma *wf-fd-field-lookup* [*rule-format*]:
$\forall f\ m\ n\ s.\ wf\text{-}fd\ (t::('a,'b)\ typ\text{-}info) \longrightarrow field\text{-}lookup\ t\ f\ m = Some\ (s,n) \longrightarrow wf\text{-}fd\ s$

$\forall f\ m\ n\ s.\ wf\text{-}fd\text{-}struct\ (st::('a,'b)\ typ\text{-}info\text{-}struct) \longrightarrow field\text{-}lookup\text{-}struct\ st\ f\ m = Some\ (s,n) \longrightarrow wf\text{-}fd\ s$
$\forall f\ m\ n\ s.\ wf\text{-}fd\text{-}list\ (ts::('a,'b)\ typ\text{-}info\text{-}tuple\ list) \longrightarrow field\text{-}lookup\text{-}list\ ts\ f\ m = Some\ (s,n) \longrightarrow wf\text{-}fd\ s$
$\forall f\ m\ n\ s.\ wf\text{-}fd\text{-}tuple\ (x::('a,'b)\ typ\text{-}info\text{-}tuple) \longrightarrow field\text{-}lookup\text{-}tuple\ x\ f\ m = Some\ (s,n) \longrightarrow wf\text{-}fd\ s$
⟨*proof*⟩

lemma *wf-fd-field-lookupD*:
$[\![\ field\text{-}lookup\ t\ f\ m = Some\ (s,n);\ wf\text{-}fd\ t\ ]\!] \Longrightarrow wf\text{-}fd\ s$
⟨*proof*⟩

lemma *wf-fd-tf-set*:
$[\![\ wf\text{-}fd\ t;\ ((s::('a,'b)\ typ\text{-}info),m) \in tf\text{-}set\ t\ ]\!] \Longrightarrow wf\text{-}fd\ s$
⟨*proof*⟩

lemma *tf-set-field-lookupD*:
$field\text{-}lookup\ t\ f\ m = Some\ (s,n) \Longrightarrow (s,f) \in tf\text{-}set\ t$
⟨*proof*⟩

lemma *fu-commutes-ts*:
$(\bigwedge\ t.\ t \in dt\text{-}fst\ `\ set\ ts \Longrightarrow fu\text{-}commutes\ d\ (update\text{-}ti\text{-}t\ t)) \Longrightarrow$
$fu\text{-}commutes\ d\ (update\text{-}ti\text{-}list\text{-}t\ ts)$
⟨*proof*⟩

lemma *fa-fu-ind-ts*:
$(\bigwedge t.\ t \in dt\text{-}fst\ `\ set\ ts \Longrightarrow fa\text{-}fu\text{-}ind\ d\ (field\text{-}desc\ t)\ (size\text{-}td\ t)\ n) \Longrightarrow$
$fa\text{-}fu\text{-}ind\ d\ (\!|\ field\text{-}access = access\text{-}ti\text{-}list\ ts,$
$field\text{-}update = update\text{-}ti\text{-}list\text{-}t\ ts,\ field\text{-}sz = size\text{-}td\text{-}list\ ts\ |\!)$
$(size\text{-}td\text{-}list\ ts)\ n$
⟨*proof*⟩

lemma *fa-fu-ind-ts2*:
$(\bigwedge t.\ t \in dt\text{-}fst\ `\ set\ ts \Longrightarrow fa\text{-}fu\text{-}ind\ (field\text{-}desc\ t)\ d\ n\ (size\text{-}td\ t)) \Longrightarrow$
$fa\text{-}fu\text{-}ind\ (\!|\ field\text{-}access = access\text{-}ti\text{-}list\ ts,$
$field\text{-}update = update\text{-}ti\text{-}list\text{-}t\ ts,\ field\text{-}sz = size\text{-}td\text{-}list\ ts\ |\!)\ d$
$n\ (size\text{-}td\text{-}list\ ts)$
⟨*proof*⟩

lemma *wf-fdp-fd* [*rule-format*]:
$\forall m.\ wf\text{-}fdp\ (tf\text{-}set\ t) \longrightarrow wf\text{-}desc\ t \longrightarrow wf\text{-}fd\ (t::('a,'b)\ typ\text{-}info)$
$\forall m.\ (case\ st\ of\ TypScalar\ sz\ algn\ d \Rightarrow fd\text{-}cons\text{-}struct\ ((TypScalar\ sz\ algn\ d)::('a,'b)\ typ\text{-}info\text{-}struct)$
$\mid\ - \Rightarrow wf\text{-}fdp\ (tf\text{-}set\text{-}struct\ st)) \longrightarrow wf\text{-}desc\text{-}struct\ st \longrightarrow wf\text{-}fd\text{-}struct\ (st::('a,'b)\ typ\text{-}info\text{-}struct)$
$\forall m.\ wf\text{-}fdp\ (tf\text{-}set\text{-}list\ ts) \longrightarrow wf\text{-}desc\text{-}list\ ts \longrightarrow wf\text{-}fd\text{-}list\ (ts::('a,'b)\ typ\text{-}info\text{-}tuple\ list)$
$\forall m.\ wf\text{-}fdp\ (tf\text{-}set\text{-}tuple\ x) \longrightarrow wf\text{-}desc\text{-}tuple\ x \longrightarrow wf\text{-}fd\text{-}tuple\ (x::('a,'b)\ typ\text{-}info\text{-}tuple)$
⟨*proof*⟩

lemma *wf-fdp-fdD*:
⟦ *wf-fdp* (*tf-set t*); *wf-desc t* ⟧ ⟹ *wf-fd* (*t*::('*a*,'*b*) *typ-info*)
⟨*proof*⟩

lemma *wf-fdp-fd-listD*:
⟦ *wf-fdp* (*tf-set-list t*); *wf-desc-list t* ⟧ ⟹ *wf-fd-list t*
⟨*proof*⟩

lemma *fd-consistentD*:
⟦ *field-lookup t f 0* = *Some* (*s*,*n*); *fd-consistent t* ⟧
⟹ *fd-cons s*
⟨*proof*⟩

lemma *wf-fd-cons-access-update'* [*rule-format*]:
wf-fd (*t*::('*a*,'*b*) *typ-info*) ⟶ *fd-cons-access-update* (*field-desc t*) (*size-td t*)
wf-fd-struct (*st*::('*a*,'*b*) *typ-info-struct*) ⟶ *fd-cons-access-update* (*field-desc-struct st*) (*size-td-struct st*)
wf-fd-list (*ts*::('*a*,'*b*) *typ-info-tuple list*) ⟶ *fd-cons-access-update* (*field-desc-list ts*) (*size-td-list ts*)
wf-fd-tuple (*x*::('*a*,'*b*) *typ-info-tuple*) ⟶ *fd-cons-access-update* (*field-desc-tuple x*) (*size-td-tuple x*)
⟨*proof*⟩

lemma *wf-fd-cons-access-updateD*:
wf-fd t ⟹ *fd-cons-access-update* (*field-desc t*) (*size-td t*)
⟨*proof*⟩

lemma *wf-fd-cons-access-update-structD*:
wf-fd-struct t ⟹ *fd-cons-access-update* (*field-desc-struct t*) (*size-td-struct t*)
⟨*proof*⟩

lemma *wf-fd-cons-access-update-listD*:
wf-fd-list t ⟹ *fd-cons-access-update* (*field-desc-list t*) (*size-td-list t*)
⟨*proof*⟩

lemma *wf-fd-cons-access-update-tupleD*:
wf-fd-tuple t ⟹ *fd-cons-access-update* (*field-desc-tuple t*) (*size-td-tuple t*)
⟨*proof*⟩

lemma *wf-fd-norm-tu*:
∀ *bs*. *wf-fd t* ⟶ *length bs* = *size-td t* ⟶ *norm-tu* (*export-uinfo* (*t*::('*a*,'*b*) *typ-info*)) *bs* = (*access-ti t* (*update-ti-t t bs undefined*) (*replicate* (*size-td t*) *0*))
∀ *bs*. *wf-fd-struct st* ⟶ *length bs* = *size-td-struct st* ⟶ *norm-tu-struct* (*map-td-struct field-norm* (λ-. ()) (*st*::('*a*,'*b*) *typ-info-struct*)) *bs* = (*access-ti-struct st* (*update-ti-struct-t st bs undefined*) (*replicate* (*size-td-struct st*) *0*))
∀ *bs*. *wf-fd-list ts* ⟶ *length bs* = *size-td-list ts* ⟶ *norm-tu-list* (*map-td-list field-norm* (λ-. ()) (*ts*::('*a*,'*b*) *typ-info-tuple list*)) *bs* = (*access-ti-list ts* (*update-ti-list-t ts bs undefined*) (*replicate* (*size-td-list ts*) *0*))

$\forall$ *bs*. *wf-fd-tuple x* $\longrightarrow$ *length bs* = *size-td-tuple x* $\longrightarrow$ *norm-tu-tuple* (*map-td-tuple field-norm* (λ-. ()) (*x*::('*a*,'*b*) *typ-info-tuple*)) *bs* = (*access-ti-tuple x* (*update-ti-tuple-t x bs undefined*) (*replicate* (*size-td-tuple x*) *0*))
⟨*proof*⟩

lemma *wf-fd-norm-tuD*:
⟦ *wf-fd t*; *length bs* = *size-td t* ⟧ $\Longrightarrow$ *norm-tu* (*export-uinfo t*) *bs* =
(*access-ti*$_0$ *t* (*update-ti-t t bs undefined*))
⟨*proof*⟩

lemma *wf-fd-norm-tu-structD*:
⟦ *wf-fd-struct t*; *length bs* = *size-td-struct t* ⟧ $\Longrightarrow$ *norm-tu-struct* (*map-td-struct field-norm* (λ-. ()) *t*) *bs* =
(*access-ti-struct t* (*update-ti-struct-t t bs undefined*) (*replicate* (*size-td-struct t*) *0*))
⟨*proof*⟩

lemma *wf-fd-norm-tu-listD*:
⟦ *wf-fd-list t*; *length bs* = *size-td-list t* ⟧ $\Longrightarrow$ *norm-tu-list* (*map-td-list field-norm* (λ-. ()) *t*) *bs* =
(*access-ti-list t* (*update-ti-list-t t bs undefined*) (*replicate* (*size-td-list t*) *0*))
⟨*proof*⟩

lemma *wf-fd-norm-tu-tupleD*:
⟦ *wf-fd-tuple t*; *length bs* = *size-td-tuple t* ⟧ $\Longrightarrow$ *norm-tu-tuple* (*map-td-tuple field-norm* (λ-. ()) *t*) *bs* =
(*access-ti-tuple t* (*update-ti-tuple-t t bs undefined*) (*replicate* (*size-td-tuple t*) *0*))
⟨*proof*⟩

lemma *wf-fd-cons* [*rule-format*]:
wf-fd t $\longrightarrow$ *fd-cons* (*t*::('*a*,'*b*) *typ-info*)
wf-fd-struct st $\longrightarrow$ *fd-cons-struct* (*st*::('*a*,'*b*) *typ-info-struct*)
wf-fd-list ts $\longrightarrow$ *fd-cons-list* (*ts*::('*a*,'*b*) *typ-info-tuple list*)
wf-fd-tuple x $\longrightarrow$ *fd-cons-tuple* (*x*::('*a*,'*b*) *typ-info-tuple*)
⟨*proof*⟩

lemma *wf-fd-consD*:
wf-fd t $\Longrightarrow$ *fd-cons t*
⟨*proof*⟩

lemma *wf-fd-cons-structD*:
wf-fd-struct t $\Longrightarrow$ *fd-cons-struct t*
⟨*proof*⟩

lemma *wf-fd-cons-listD*:
wf-fd-list t $\Longrightarrow$ *fd-cons-list t*
⟨*proof*⟩

lemma *wf-fd-cons-tupleD*:
wf-fd-tuple t $\Longrightarrow$ *fd-cons-tuple t*
⟨*proof*⟩

lemma *fd-cons-list-append*:
⟦ *wf-fd-list xs*; *wf-fd-list ys*; *fu-commutes*
(*field-update* (*field-desc-list xs*)) (*field-update* (*field-desc-list ys*)) ⟧ $\Longrightarrow$
fd-cons-list (*xs*@*ys*)
⟨*proof*⟩

lemma (**in** *wf-type*) *wf-fd* [*simp*]:
wf-fd (*typ-info-t TYPE*('*a*))
⟨*proof*⟩

lemma (**in** *wf-type*) *fd-cons* [*simp*]:
fd-consistent (*typ-info-t TYPE*('*a*))
⟨*proof*⟩

lemma (**in** *wf-type*) *field-lvalue-append* [*simp*]:
⟦ *field-ti TYPE*('*a*) *f* = *Some t*;
export-uinfo t = *typ-uinfo-t TYPE*('*b*::*c-type*);
field-ti TYPE('*b*) *g* = *Some k* ⟧ $\Longrightarrow$
&(((*Ptr* &((*p*::'*a ptr*)→*f*))::'*b ptr*)→*g*) = &(*p*→*f*@*g*)
⟨*proof*⟩

lemma (**in** *wf-type*) *field-lvalue-cons-unfold'*:
— rhs contains additional type variable '*b*, hence simplifier won't apply this rule
⟦ *field-ti TYPE*('*a*) [*f*] = *Some t*;
export-uinfo t = *typ-uinfo-t TYPE*('*b*::*c-type*);
field-ti TYPE('*b*) *g* = *Some k* ⟧ $\Longrightarrow$
&(*p*→*f*#*g*) = &(((*Ptr* &((*p*::'*a ptr*)→[*f*]))::'*b ptr*)→*g*)
⟨*proof*⟩

lemma (**in** *mem-type*) *field-lvalue-cons-unfold*:
⟦*field-ti TYPE*('*b*) *g* = *Some k*;
export-uinfo t = *export-uinfo* (*typ-info-t TYPE*('*b*::*c-type*));
field-ti TYPE('*a*) [*f*] = *Some t*⟧ $\Longrightarrow$
&(*p*→*f*#*g*) ≡ &(((*Ptr* &((*p*::'*a ptr*)→[*f*]))::'*b ptr*)→*g*)
⟨*proof*⟩

lemma *field-access-update-take-drop* [*rule-format*]:
∀ *f s m n bs bs' v*. *field-lookup t f m* = *Some* (*s*,*m*+*n*) ⟶
length bs = *size-td t* ⟶ *length bs'* = *size-td s* ⟶ *wf-fd t* ⟶
field-access (*field-desc s*) (*field-update* (*field-desc t*) *bs v*) *bs'*
= *field-access* (*field-desc s*) (*field-update* (*field-desc s*)

(*take* (*size-td* (*s*::('*a*,'*b*) *typ-info*)) (*drop n bs*)) *undefined*) *bs'*
∀ *f s m n bs bs' v*. *field-lookup-struct st f m* = *Some* (*s*,*m*+*n*) ⟶
length bs = *size-td-struct st* ⟶ *length bs'* = *size-td s* ⟶ *wf-fd-struct st* ⟶
field-access (*field-desc s*) (*field-update* (*field-desc-struct st*) *bs v*) *bs'*
= *field-access* (*field-desc s*) (*field-update* (*field-desc s*)
(*take* (*size-td* (*s*::('*a*,'*b*) *typ-info*)) (*drop n bs*)) *undefined*) *bs'*
∀ *f s m n bs bs' v*. *field-lookup-list ts f m* = *Some* (*s*,*m*+*n*) ⟶
length bs = *size-td-list ts* ⟶ *length bs'* = *size-td s* ⟶ *wf-fd-list ts* ⟶
field-access (*field-desc s*) (*field-update* (*field-desc-list ts*) *bs v*) *bs'*
= *field-access* (*field-desc s*) (*field-update* (*field-desc s*)
(*take* (*size-td* (*s*::('*a*,'*b*) *typ-info*)) (*drop n bs*)) *undefined*) *bs'*
∀ *f s m n bs bs' v*. *field-lookup-tuple x f m* = *Some* (*s*,*m*+*n*) ⟶
length bs = *size-td-tuple x* ⟶ *length bs'* = *size-td s* ⟶ *wf-fd-tuple x* ⟶
field-access (*field-desc s*) (*field-update* (*field-desc-tuple x*) *bs v*) *bs'*
= *field-access* (*field-desc s*) (*field-update* (*field-desc s*)
(*take* (*size-td* (*s*::('*a*,'*b*) *typ-info*)) (*drop n bs*)) *undefined*) *bs'*
⟨*proof*⟩

lemma *field-access-update-take-dropD*:
⟦ *field-lookup t f m* = *Some* (*s*,*m*+*n*); *length bs* = *size-td t*;
length bs' = *size-td s*; *wf-fd t* ⟧ ⟹
field-access (*field-desc s*) (*field-update* (*field-desc t*) *bs v*) *bs'*
= *field-access* (*field-desc s*) (*field-update* (*field-desc s*)
(*take* (*size-td* (*s*::('*a*,'*b*) *typ-info*)) (*drop n bs*)) *undefined*) *bs'*
⟨*proof*⟩

lemma (**in** *wf-type*) *fi-fa-consistentD*:
⟦ *field-lookup* (*typ-info-t TYPE*('*a*)) *f* 0 = *Some* (*d*,*n*);
length bs = *size-of TYPE*('*a*) ⟧ ⟹
field-access (*field-desc d*) (*from-bytes bs*) (*replicate* (*size-td d*) 0) =
norm-tu (*export-uinfo d*) (*take* (*size-td d*) (*drop n bs*))
⟨*proof*⟩

lemma *fi-fu-consistent* [*rule-format*]:
∀ *f m n s bs v w*. *field-lookup t f m* = *Some* (*s*,*n* + *m*) ⟶ *wf-fd t* ⟶
length bs = *size-td t* ⟶ *length v* = *size-td* (*s*::('*a*,'*b*) *typ-info*) ⟶
field-update (*field-desc t*) (*super-update-bs v bs n*) *w* =
field-update (*field-desc s*) *v* (*field-update* (*field-desc t*) *bs w*)
∀ *f m n s bs v w*. *field-lookup-struct st f m* = *Some* (*s*,*n* + *m*) ⟶ *wf-fd-struct st*
⟶
length bs = *size-td-struct st* ⟶ *length v* = *size-td* (*s*::('*a*,'*b*) *typ-info*) ⟶
field-update (*field-desc-struct st*) (*super-update-bs v bs n*) *w* =
field-update (*field-desc s*) *v* (*field-update* (*field-desc-struct st*) *bs w*)
∀ *f m n s bs v w*. *field-lookup-list ts f m* = *Some* (*s*,*n* + *m*) ⟶ *wf-fd-list ts* ⟶
length bs = *size-td-list ts* ⟶ *length v* = *size-td* (*s*::('*a*,'*b*) *typ-info*) ⟶
field-update (*field-desc-list ts*) (*super-update-bs v bs n*) *w* =
field-update (*field-desc s*) *v* (*field-update* (*field-desc-list ts*) *bs w*)
∀ *f m n s bs v w*. *field-lookup-tuple x f m* = *Some* (*s*,*n* + *m*) ⟶ *wf-fd-tuple x* ⟶

length bs = size-td-tuple x ⟶ length v = size-td (s::('a,'b) typ-info) ⟶
field-update (field-desc-tuple x) (super-update-bs v bs n) w =
field-update (field-desc s) v (field-update (field-desc-tuple x) bs w)
⟨*proof*⟩

lemma *fi-fu-consistentD*:
⟦ *field-lookup t f 0 = Some (s,n)*; *wf-fd t*; *length bs = size-td t*;
length v = size-td s ⟧ ⟹
field-update (field-desc t) (super-update-bs v bs n) w =
field-update (field-desc s) v (field-update (field-desc t) bs w)
⟨*proof*⟩

lemma (**in** *wf-type*) *norm*:
assumes *lbs*: *length bs = size-of TYPE('a)*
shows *from-bytes (norm-bytes TYPE('a) bs) = ((from-bytes bs)::'a)*
⟨*proof*⟩

lemma (**in** *wf-type*) *len*:
length bs = size-of TYPE('a) ⟹
length (to-bytes (x::'a) bs) = size-of TYPE('a)
⟨*proof*⟩

lemma (**in** *wf-type*) *sz-nzero*:
0 < size-of (TYPE('a))
⟨*proof*⟩

lemma *not-disj-fn-empty1* [*simp*]:
¬ *disj-fn [] s*
⟨*proof*⟩

lemma *disj-fn-comm*: *disj-fn a b ⟷ disj-fn b a*
⟨*proof*⟩

lemma *disj-fn-append-right*: *disj-fn x y ⟹ disj-fn x (y @ z)*
⟨*proof*⟩

lemma *disj-fn-cons-consI*[*simp*]: *(x = y ⟶ disj-fn a b) ⟹ disj-fn (x # a) (y # b)*
⟨*proof*⟩

lemma *fd-path-cons* [*simp*]:
f ∉ fs-path (x#xs) = (disj-fn f x ∧ f ∉ fs-path xs)
⟨*proof*⟩

lemma *fu-commutes-lookup-disjD*:
⟦ *field-lookup t f m = Some (d,n)*; *field-lookup t f' m' = Some (d',n')*;
disj-fn f f'; *wf-fdp (tf-set t)* ⟧ ⟹

fu-commutes (*field-update* (*field-desc* (*d*::('*a*,'*b*) *typ-info*)))
(*field-update* (*field-desc d'*))
⟨*proof*⟩

lemma *field-lookup-fa-fu-lhs*:
∀ *f m n s d k*. *field-lookup t f m* = *Some* (*s*,*n*) ⟶ *fa-fu-ind* (*field-desc t*) *d k* (*size-td t*)
⟶ *wf-fd t* ⟶ *fa-fu-ind* (*field-desc* (*s*::('*a*,'*b*) *typ-info*)) *d k* (*size-td s*)
∀ *f m n s d k*. *field-lookup-struct st f m* = *Some* (*s*,*n*) ⟶ *fa-fu-ind* (*field-desc-struct st*) *d k* (*size-td-struct st*)
⟶ *wf-fd-struct st* ⟶ *fa-fu-ind* (*field-desc* (*s*::('*a*,'*b*) *typ-info*)) *d k* (*size-td s*)
∀ *f m n s d k*. *field-lookup-list ts f m* = *Some* (*s*,*n*) ⟶ *fa-fu-ind* (*field-desc-list ts*) *d k* (*size-td-list ts*)
⟶ *wf-fd-list ts* ⟶ *fa-fu-ind* (*field-desc* (*s*::('*a*,'*b*) *typ-info*)) *d k* (*size-td s*)
∀ *f m n s d k*. *field-lookup-tuple x f m* = *Some* (*s*,*n*) ⟶ *fa-fu-ind* (*field-desc-tuple x*) *d k* (*size-td-tuple x*)
⟶ *wf-fd-tuple x* ⟶ *fa-fu-ind* (*field-desc* (*s*::('*a*,'*b*) *typ-info*)) *d k* (*size-td s*)
⟨*proof*⟩

lemma *field-lookup-fa-fu-lhs-listD*:
⟦ *field-lookup-list ts f m* = *Some* (*s*,*n*); *fa-fu-ind* (*field-desc-list ts*) *d k* (*size-td-list ts*);
wf-fd-list ts ⟧ ⟹ *fa-fu-ind* (*field-desc* (*s*::('*a*,'*b*) *typ-info*)) *d k* (*size-td s*)
⟨*proof*⟩

lemma *field-lookup-fa-fu-lhs-tupleD*:
⟦ *field-lookup-tuple x f m* = *Some* (*s*,*n*); *fa-fu-ind* (*field-desc-tuple x*) *d k* (*size-td-tuple x*);
wf-fd-tuple x ⟧ ⟹ *fa-fu-ind* (*field-desc* (*s*::('*a*,'*b*) *typ-info*)) *d k* (*size-td s*)
⟨*proof*⟩

lemma *field-lookup-fa-fu-rhs*:
∀ *f m n s d k* . *field-lookup t f m* = *Some* (*s*,*n*) ⟶ *fa-fu-ind d* (*field-desc t*) (*size-td t*) *k*
⟶ *wf-fd t* ⟶ *fa-fu-ind d* (*field-desc* (*s*::('*a*,'*b*) *typ-info*)) (*size-td s*) *k*
∀ *f m n s d k*. *field-lookup-struct st f m* = *Some* (*s*,*n*) ⟶ *fa-fu-ind d* (*field-desc-struct st*) (*size-td-struct st*) *k*
⟶ *wf-fd-struct st* ⟶ *fa-fu-ind d* (*field-desc* (*s*::('*a*,'*b*) *typ-info*)) (*size-td s*) *k*
∀ *f m n s d k*. *field-lookup-list ts f m* = *Some* (*s*,*n*) ⟶ *fa-fu-ind d* (*field-desc-list ts*) (*size-td-list ts*) *k*
⟶ *wf-fd-list ts* ⟶ *fa-fu-ind d* (*field-desc* (*s*::('*a*,'*b*) *typ-info*)) (*size-td s*) *k*
∀ *f m n s d k*. *field-lookup-tuple x f m* = *Some* (*s*,*n*) ⟶ *fa-fu-ind d* (*field-desc-tuple x*) (*size-td-tuple x*) *k*
⟶ *wf-fd-tuple x* ⟶ *fa-fu-ind d* (*field-desc* (*s*::('*a*,'*b*) *typ-info*)) (*size-td s*) *k*
⟨*proof*⟩

lemma *field-lookup-fa-fu-rhs-listD*:
⟦ *field-lookup-list ts f m* = *Some* (*s*,*n*);

fa-fu-ind d (field-desc-list ts) (size-td-list ts) k; wf-fd-list ts ⟧ ⟹
fa-fu-ind d (field-desc (s::('a,'b) typ-info)) (size-td s) k
⟨*proof*⟩

lemma *field-lookup-fa-fu-rhs-tupleD*:
⟦ *field-lookup-tuple x f m = Some (s,n)*;
fa-fu-ind d (field-desc-tuple x) (size-td-tuple x) k; wf-fd-tuple x ⟧ ⟹
fa-fu-ind d (field-desc (s::('a,'b) typ-info)) (size-td s) k
⟨*proof*⟩

lemma *fa-fu-lookup-ind-list-tuple*:
shows
⟦ *field-lookup-tuple x f m = Some (d',n); wf-fd-tuple x*;
field-lookup-list ts f' m' = Some (d,n'); wf-fd-list ts;
fa-fu-ind (field-desc-list ts) (field-desc-tuple x) (size-td-tuple x) (size-td-list ts)
⟧
⟹ *fa-fu-ind (field-desc d) (field-desc d') (size-td d') (size-td d)*
⟨*proof*⟩

lemma *fa-fu-lookup-ind-tuple-list*:
⟦ *field-lookup-tuple x f m = Some (d,n); wf-fd-tuple x*;
field-lookup-list ts f' m' = Some (d',n'); wf-fd-list ts;
fa-fu-ind (field-desc-tuple x) (field-desc-list ts) (size-td-list ts) (size-td-tuple x)
⟧
⟹ *fa-fu-ind (field-desc d) (field-desc d') (size-td d') (size-td d)*
⟨*proof*⟩

lemma *fa-fu-lookup-disj*:
∀ *f m d n f' m' d' n'. field-lookup t f m = Some (d,n)* ⟶
field-lookup t f' m' = Some (d',n') ⟶ *disj-fn f f'* ⟶
wf-fd t ⟶ *fa-fu-ind (field-desc (d::('a,'b) typ-info)) (field-desc d') (size-td d')*
(size-td d)
∀ *f m d n f' m' d' n'. field-lookup-struct st f m = Some (d,n)* ⟶
field-lookup-struct st f' m' = Some (d',n') ⟶ *disj-fn f f'* ⟶
wf-fd-struct st ⟶ *fa-fu-ind (field-desc (d::('a,'b) typ-info)) (field-desc d')*
(size-td d') (size-td d)
∀ *f m d n f' m' d' n'. field-lookup-list ts f m = Some (d,n)* ⟶
field-lookup-list ts f' m' = Some (d',n') ⟶ *disj-fn f f'* ⟶
wf-fd-list ts ⟶ *fa-fu-ind (field-desc (d::('a,'b) typ-info)) (field-desc d') (size-td*
d') (size-td d)
∀ *f m d n f' m' d' n'. field-lookup-tuple x f m = Some (d,n)* ⟶
field-lookup-tuple x f' m' = Some (d',n') ⟶ *disj-fn f f'* ⟶
wf-fd-tuple x ⟶ *fa-fu-ind (field-desc (d::('a,'b) typ-info)) (field-desc d')*
(size-td d') (size-td d)
⟨*proof*⟩

lemma *fa-fu-lookup-disjD*:
⟦ *field-lookup t f m = Some (d,n); field-lookup t f' m' = Some (d',n')*;

disj-fn f f′; *wf-fd t* ⟧ ⟹
fa-fu-ind (*field-desc* (*d*::('*a*,'*b*) *typ-info*)) (*field-desc d′*) (*size-td d′*) (*size-td d*)
⟨*proof*⟩

lemma *field-access-update-disj*:
⟦ *field-lookup t f m* = *Some* (*d*,*n*); *field-lookup t f′ m′* = *Some* (*d′*,*n′*);
disj-fn f f′; *length bs* = *size-td d′*; *length bs′* = *size-td d*; *wf-fd t* ⟧ ⟹
access-ti d (*update-ti-t d′ bs v*) *bs′* = *access-ti d v bs′*
⟨*proof*⟩

lemma *td-set-list-intvl-sub*:
(*d*,*n*) ∈ *td-set-list t m* ⟹ {*of-nat n*..+*size-td d*} ⊆ {*of-nat m*..+*size-td-list t*}
⟨*proof*⟩

lemma *td-set-tuple-intvl-sub*:
(*d*,*n*) ∈ *td-set-tuple t m* ⟹ {*of-nat n*..+*size-td d*} ⊆ {*of-nat m*..+*size-td-tuple t*}
⟨*proof*⟩

lemma *intvl-inter-le*:
assumes *inter*: *a* + *of-nat k* = *c* + *of-nat ka* **and** *lt-d*: *ka* < *d* **and** *lt-ka*: *k* ≤ *ka*
shows *a* ∈ {*c*..+*d*}
⟨*proof*⟩

lemma *intvl-inter*:
assumes *nondisj*: {*a*..+*b*} ∩ {*c*..+*d*} ≠ {}
shows *a* ∈ {*c*..+*d*} ∨ *c* ∈ {*a*..+*b*}
⟨*proof*⟩

lemma *init-intvl-disj*:
k + *z* < *addr-card* ⟹ {(*p*::*addr*)+*of-nat k*..+*z*} ∩ {*p*..+*k*} = {}
⟨*proof*⟩

lemma *final-intvl-disj*:
⟦ *k* + *z* ≤ *n*; *n* < *addr-card* ⟧ ⟹
{(*p*::*addr*)+*of-nat k*..+*z*} ∩ {*p*+(*of-nat k* + *of-nat z*)..+*n* − (*k*+*z*)} = {}
⟨*proof*⟩

lemma *fa-fu-lookup-disj-inter*:
∀ *f m d n f′ d′ n′*. *field-lookup t f m* = *Some* (*d*,*n*) ⟶
field-lookup t f′ m = *Some* (*d′*,*n′*) ⟶ *disj-fn f f′* ⟶
wf-fd t ⟶ *size-td t* < *addr-card* ⟶
{(*of-nat n*)::*addr*..+*size-td* (*d*::('*a*,'*b*) *typ-info*)} ∩ {*of-nat n′*..+*size-td d′*} = {}
∀ *f m d n f′ m d′ n′*. *field-lookup-struct st f m* = *Some* (*d*,*n*) ⟶
field-lookup-struct st f′ m = *Some* (*d′*,*n′*) ⟶ *disj-fn f f′* ⟶
wf-fd-struct st ⟶ *size-td-struct st* < *addr-card* ⟶
{(*of-nat n*)::*addr*..+*size-td* (*d*::('*a*,'*b*) *typ-info*)} ∩ {*of-nat n′*..+*size-td d′*} =

{}
  ∀*f m d n f′ m d′ n′*. *field-lookup-list ts f m* = *Some* (*d*,*n*) ⟶
      field-lookup-list ts f′ m = *Some* (*d′*,*n′*) ⟶ *disj-fn f f′* ⟶
      wf-fd-list ts ⟶ *size-td-list ts* < *addr-card* ⟶
      {(*of-nat n*)::*addr*..+*size-td* (*d*::('*a*,'*b*) *typ-info*)} ∩ {*of-nat n′*..+*size-td d′*} =
{}
  ∀*f m d n f′ m d′ n′*. *field-lookup-tuple x f m* = *Some* (*d*,*n*) ⟶
      field-lookup-tuple x f′ m = *Some* (*d′*,*n′*) ⟶ *disj-fn f f′* ⟶
      wf-fd-tuple x ⟶ *size-td-tuple x* < *addr-card* ⟶
      {(*of-nat n*)::*addr*..+*size-td* (*d*::('*a*,'*b*) *typ-info*)} ∩ {*of-nat n′*..+*size-td d′*} =
{}
⟨*proof*⟩

lemma *fa-fu-lookup-disj-interD*:
  ⟦ *field-lookup t f m* = *Some* (*d*,*n*); *field-lookup t f′ m* = *Some* (*d′*,*n′*);
      disj-fn f f′; *wf-fd t*; *size-td t* < *addr-card* ⟧ ⟹
      {(*of-nat n*)::*addr*..+*size-td* (*d*::('*a*,'*b*) *typ-info*)} ∩ {*of-nat n′*..+*size-td d′*} =
{}
  ⟨*proof*⟩

lemma *fa-fu-lookup-disj-inter-listD*:
  ⟦ *field-lookup-list ts f m* = *Some* (*d*,*n*);
      field-lookup-list ts f′ m = *Some* (*d′*,*n′*); *disj-fn f f′*;
      wf-fd-list ts; *size-td-list ts* < *addr-card* ⟧ ⟹
      {(*of-nat n*)::*addr*..+*size-td* (*d*::('*a*,'*b*) *typ-info*)} ∩
          {*of-nat n′*..+*size-td d′*} = {}
  ⟨*proof*⟩

lemma (**in** *mem-type-sans-size*) *upd-rf*:
  length bs = *size-of TYPE*('*a*) ⟹
      update-ti-t (*typ-info-t TYPE*('*a*)) *bs v*
          = *update-ti-t* (*typ-info-t TYPE*('*a*)) *bs w*
  ⟨*proof*⟩

lemma (**in** *mem-type-sans-size*) *inv*:
  length bs = *size-of TYPE*('*a*) ⟹
      from-bytes (*to-bytes* (*x*::'*a*) *bs*) = *x*
  ⟨*proof*⟩

lemma (**in** *mem-type*) *align*:
  align-of (*TYPE*('*a*)) *dvd addr-card*
⟨*proof*⟩

lemma (**in** *mem-type*) *to-bytes-inj*:
  to-bytes (*v*::'*a*) = *to-bytes* (*v′*::'*a*) ⟹ *v*=*v′*
  ⟨*proof*⟩

lemmas *unat-simps* = *unat-simps′ max-size*

lemmas (**in** *mem-type*) *mem-type-simps* [*simp*] = *inv len sz-nzero max-size align*
lemmas *mem-type-simps* [*simp*] = *inv len sz-nzero max-size align*

lemma (**in** *mem-type*) *ptr-aligned-plus*:
assumes *aligned*: *ptr-aligned* (*p*::'*a ptr*)
shows *ptr-aligned* (*p* +$_p$ *i*)
⟨*proof*⟩

lemma (**in** *mem-type*) *mem-type-self* [*simp*]:
ptr-val (*p*::'*a ptr*) ∈ {*ptr-val p*..+*size-of TYPE*('*a*)}
⟨*proof*⟩

lemma (**in** *mem-type*) *intvl-Suc-nmem* [*simp*]:
(*p*::*addr*) ∉ {*p* + *1*..+*size-of TYPE*('*a*) − *Suc 0*}
⟨*proof*⟩

lemma (**in** *mem-type*) *wf-size-desc-typ-uinfo-t-simp* [*simp*]:
wf-size-desc (*typ-uinfo-t TYPE*('*a*))
⟨*proof*⟩

lemma *aggregate-map* [*simp*]:
aggregate (*map-td f g t*) = *aggregate t*
⟨*proof*⟩

lemma (**in** *simple-mem-type*) *simple-tag-not-aggregate2* [*simp*]:
typ-uinfo-t TYPE('*a*) ≠ *TypDesc algn* (*TypAggregate ts*) *tn*
⟨*proof*⟩

lemma (**in** *simple-mem-type*) *simple-tag-not-aggregate3* [*simp*]:
typ-info-t TYPE('*a*) ≠ *TypDesc algn* (*TypAggregate ts*) *tn*
⟨*proof*⟩

lemma (**in** *mem-type*) *field-of-t-mem*:
field-of-t (*p*::'*a ptr*) (*q*::'*b*::*mem-type ptr*) ⟹
ptr-val p ∈ {*ptr-val q*..+*size-of TYPE*('*b*)}
⟨*proof*⟩

lemma *map-td-map*:
map-td f p (*map-td g q t*) = *map-td* (λ*n algn*. *f n algn o g n algn*) (*p o q*) *t*
map-td-struct f p (*map-td-struct g q st*) = *map-td-struct* (λ*n algn*. *f n algn o g n algn*) (*p o q*) *st*
map-td-list f p (*map-td-list g q ts*) = *map-td-list* (λ*n algn*. *f n algn o g n algn*) (*p o q*) *ts*
map-td-tuple f p (*map-td-tuple g q x*) = *map-td-tuple* (λ*n algn*. *f n algn o g n algn*) (*p o q*) *x*
⟨*proof*⟩

lemma *field-of-t-simple*:
field-of-t p (x::'a::simple-mem-type ptr) $\Longrightarrow$ *ptr-val p = ptr-val x*
⟨*proof*⟩

lemma *fold-td'-unfold*:
fold-td' t =
(let (f,s) = t in
case s of TypDesc algn' st nm ⇒
(case st of
TypScalar n algn d ⇒ d
| TypAggregate ts ⇒ f nm (map (λx. case x of DTuple t n d ⇒ (fold-td'
(f,t),n)) ts)))
⟨*proof*⟩

lemma *fold-td-alt-def'*:
fold-td f t = (case t of
TypDesc algn' st nm ⇒
(case st of
TypScalar n algn d ⇒ d
| TypAggregate ts ⇒ f nm (map (λx. (fold-td f (dt-fst x),dt-snd
x)) ts)))
⟨*proof*⟩

lemma *fold-td-alt-def*:
fold-td f t ≡ (case t of
TypDesc algn' st nm ⇒
(case st of
TypScalar n algn d ⇒ d
| TypAggregate ts ⇒ f nm (map (λx. (fold-td f (dt-fst x),dt-snd
x)) ts)))
⟨*proof*⟩

lemma *map-td'-map'*:
map-td f g t = (map-td' ((f,g),t))
TypDesc algn (map-td-struct f g st) (typ-name t) = (map-td' ((f,g),TypDesc algn
st (typ-name t)))
TypDesc algn (TypAggregate (map-td-list f g ts)) (typ-name t) = map-td' ((f,g),TypDesc
algn (TypAggregate ts) (typ-name t))
map-td-tuple f g x = DTuple (map-td' ((f,g),dt-fst x)) (dt-snd x) (g (dt-trd x))
⟨*proof*⟩

lemma *map-td'-map*:
map-td f g t = (case t of TypDesc algn st nm ⇒ TypDesc algn (case st of
TypScalar n algn d ⇒ TypScalar n algn (f n algn d) |
TypAggregate ts ⇒ TypAggregate (map (λx. DTuple (map-td f g (dt-fst x))
(dt-snd x) (g (dt-trd x))) ts)) nm)
⟨*proof*⟩

lemma *map-td-alt-def*:
map-td f g t ≡ (*case t of TypDesc algn st nm* ⇒ *TypDesc algn* (*case st of*
TypScalar n algn d ⇒ *TypScalar n algn* (*f n algn d*) |
TypAggregate ts ⇒ *TypAggregate* (*map* (λ*x*. *DTuple* (*map-td f g* (*dt-fst x*))
(*dt-snd x*) (*g* (*dt-trd x*))) *ts*)) *nm*)
⟨*proof*⟩

lemma *size-td-fm'*:
size-td (*t*::('*a*,'*b*) *typ-desc*) = *fold-td tnSum* (*map-td* (λ*n x d*. *n*) *g t*)
size-td-struct (*st*::('*a*,'*b*) *typ-struct*) = *fold-td-struct* (*typ-name t*) *tnSum* (*map-td-struct*
(λ*n x d*. *n*) *g st*)
size-td-list (*ts*::('*a*,'*b*) *typ-tuple list*) = *fold-td-list* (*typ-name t*) *tnSum* (*map-td-list*
(λ*n x d*. *n*) *g ts*)
size-td-tuple (*x*::('*a*,'*b*) *typ-tuple*) = *fold-td-tuple tnSum* (*map-td-tuple* (λ*n x d*.
n) *g x*)
⟨*proof*⟩

lemma *size-td-fm*:
size-td (*t*::('*a*,'*b*) *typ-desc*) ≡ *fold-td tnSum* (*map-td* (λ*n algn d*. *n*) *id t*)
⟨*proof*⟩

lemma *align-td-wo-align-fm'*:
align-td-wo-align (*t*::('*a*,'*b*) *typ-desc*) = *fold-td tnMax* (*map-td* (λ*n x d*. *x*) *g t*)
align-td-wo-align-struct (*st*::('*a*,'*b*) *typ-struct*) = *fold-td-struct* (*typ-name t*) *tn-*
Max (*map-td-struct* (λ*n x d*. *x*) *g st*)
align-td-wo-align-list (*ts*::('*a*,'*b*) *typ-tuple list*) = *fold-td-list* (*typ-name t*) *tnMax*
(*map-td-list* (λ*n x d*. *x*) *g ts*)
align-td-wo-align-tuple (*x*::('*a*,'*b*) *typ-tuple*) = *fold-td-tuple tnMax* (*map-td-tuple*
(λ*n x d*. *x*) *g x*)
⟨*proof*⟩

lemma *align-td-wo-align-fm*:
align-td-wo-align (*t*::('*a*,'*b*) *typ-desc*) ≡ *fold-td tnMax* (*map-td* (λ*n algn d*. *algn*)
id t)
⟨*proof*⟩

thm *case-dt-tuple-def*

lemma *case-dt-tuple*:
snd ‘ *case-dt-tuple* (λ*t n d*. *Pair* (*f t*) *n*) ‘ *X* = *dt-snd* ‘ *X*
⟨*proof*⟩

lemma *map-DTuple-dt-snd*:
map-td-tuple f g x = *DTuple a b c* ⟹ *b* = *dt-snd x*
⟨*proof*⟩

lemma *wf-desc-fm'*:
wf-desc (*t*::('*a*,'*b*) *typ-desc*) = *fold-td wfd* (*map-td* (λ*n x d*. *True*) *g t*)

wf-desc-struct (*st*::('*a*,'*b*) *typ-struct*) = *fold-td-struct* (*typ-name t*) *wfd* (*map-td-struct* (λ*n x d*. *True*) *g st*)
wf-desc-list (*ts*::('*a*,'*b*) *typ-tuple list*) = *fold-td-list* (*typ-name t*) *wfd* (*map-td-list* (λ*n x d*. *True*) *g ts*)
wf-desc-tuple (*x*::('*a*,'*b*) *typ-tuple*) = *fold-td-tuple wfd* (*map-td-tuple* (λ*n x d*. *True*) *g x*)
⟨*proof*⟩

lemma *wf-desc-fm*:
wf-desc (*t*::('*a*,'*b*) *typ-desc*) ≡ *fold-td wfd* (*map-td* (λ*n algn d*. *True*) *id t*)
⟨*proof*⟩

lemma *update-tag-list-empty* [*simp*]:
(*map-td-list f g xs* = []) = (*xs* = [])
⟨*proof*⟩

lemma *wf-size-desc-fm'*:
wf-size-desc (*t*::('*a*,'*b*) *typ-desc*) = *fold-td wfsd* (*map-td* (λ*n x d*. 0 < *n*) *g t*)
wf-size-desc-struct (*st*::('*a*,'*b*) *typ-struct*) = *fold-td-struct* (*typ-name t*) *wfsd* (*map-td-struct* (λ*n x d*. 0 < *n*) *g st*)
ts ≠ [] ⟶ *wf-size-desc-list* (*ts*::('*a*,'*b*) *typ-tuple list*) = *fold-td-list* (*typ-name t*) *wfsd* (*map-td-list* (λ*n x d*. 0 < *n*) *g ts*)
wf-size-desc-tuple (*x*::('*a*,'*b*) *typ-tuple*) = *fold-td-tuple wfsd* (*map-td-tuple* (λ*n x d*. 0 < *n*) *g x*)
⟨*proof*⟩

lemma *wf-size-desc-fm*:
wf-size-desc (*t*::('*a*,'*b*) *typ-desc*) ≡ *fold-td wfsd* (*map-td* (λ*n algn d*. 0 < *n*) *id t*)
⟨*proof*⟩

typedef *stack-byte* = *UNIV*::*byte set*
⟨*proof*⟩

definition *stack-byte-name* = ''*stack-byte*''

instantiation *stack-byte* :: *c-type*
begin

definition *typ-name-itself* (*x*::*stack-byte itself*) = *stack-byte-name*

definition
typ-info-stack-byte: *typ-info-t* (*x*::*stack-byte itself*) ≡
TypDesc 0 (*TypScalar 1 0*
(⦇*field-access* = λ*v bs*. [*Rep-stack-byte v*],
field-update = λ*bs v*. *if length bs* ≥ *1 then Abs-stack-byte* (*hd bs*) *else Abs-stack-byte* (*0*::*byte*),
field-sz = *1*⦈))
stack-byte-name

instance
⟨*proof*⟩
end

lemma *size-of-stack-byte* [*simp*]:
size-of TYPE(*stack-byte*) = *1*
size-td (*typ-info-t TYPE*(*stack-byte*)) = *1*
size-td (*typ-uinfo-t TYPE*(*stack-byte*)) = *1*
⟨*proof*⟩

lemma *align-of-stack-byte* [*simp*]:
align-of TYPE(*stack-byte*) = *1*
align-td (*typ-info-t TYPE*(*stack-byte*)) = *0*
align-td (*typ-uinfo-t TYPE*(*stack-byte*)) = *0*
⟨*proof*⟩

lemma *length-1-conv*: *length bs* = *Suc 0* ⟷ (∃ *b*. *bs* = [*b*])
⟨*proof*⟩
lemma *padding-lense-stack-byte*: *padding-lense* (λ*v bs*. [*Rep-stack-byte v*])
(λ*bs v*.
if Suc 0 ≤ *length bs then Abs-stack-byte* (*hd bs*) *else Abs-stack-byte 0*)
(*Suc 0*)
⟨*proof*⟩

instantiation *stack-byte*:: *xmem-contained-type*
begin
instance
⟨*proof*⟩

end

end

theory *Vanilla32-Preliminaries*
imports *CTypes*
begin

11.12 Words and Pointers

instantiation *unit* :: *c-type*
begin
definition [*simp*]: *typ-name-itself* (*x*::*unit itself*) = ″*unit*″
definition *typ-info-unit*[*simp*]:
typ-info-t (*x*::*unit itself*) ≡
TypDesc 0 (*TypScalar 1 0*
(⦇*field-access* = (λ*v bs*. [*0*]), *field-update* = (λ*bs v*. ()), *field-sz* = *1*⦈))

```
        ''unit''
        :: unit xtyp-info
instance ⟨proof⟩
end

lemma typ-name-unit[simp]:
  typ-name (typ-info-t (TYPE(unit))) = ''unit''
  typ-name (typ-uinfo-t TYPE(unit)) = ''unit''
  ⟨proof⟩

instantiation unit :: mem-type
begin
  definition
    to-bytes-unit :: unit ⇒ byte list where
    to-bytes-unit a ≡ [0]

  definition
    from-bytes-unit :: byte list ⇒ unit where
    from-bytes-unit bs ≡ ()

  definition
    size-of-unit :: unit itself ⇒ nat where
    size-of-unit x ≡ 0

  definition
    align-of-unit :: unit itself ⇒ nat where
    align-of-unit x ≡ 1

  instance
    ⟨proof⟩
end

definition
  bogus-log2lessthree (n::nat) ==
          if n = 128 then 4
          else if n = 64 then (3::nat)
          else if n = 32 then 2
          else if n = 16 then 1
          else if n = 8 then 0
          else undefined
definition
  len-exp (x::('a::len) itself) ≡ bogus-log2lessthree (len-of TYPE('a))

lemma lx8' [simp] : len-exp (x::8 itself) = 0
⟨proof⟩

lemma lx16' [simp]: len-exp (x::16 itself) = 1
⟨proof⟩
```

lemma *lx32'* [*simp*]: *len-exp* (*x*::*32 itself*) = *2*
⟨*proof*⟩

lemma *lx64'* [*simp*]: *len-exp* (*x*::*64 itself*) = *3*
⟨*proof*⟩

lemma *lx128'* [*simp*]: *len-exp* (*x*::*128 itself*) = *4*
⟨*proof*⟩

lemma *lx-signed'* [*simp*]: *len-exp* (*x*::(*'a*::*len*) *signed itself*) = *len-exp* (*TYPE*(*'a*))
⟨*proof*⟩

class *len8* = *len* +

assumes *len8-bytes*: *len-of TYPE*(*'a*::*len*) = *8* ∗ (*2^len-exp TYPE*(*'a*))

assumes *len8-width*: *len-of TYPE*(*'a*::*len*) ≤ *128*
begin

lemma *len8-size*:
len-of TYPE(*'a*) *div 8* < *addr-card*
⟨*proof*⟩

lemma *len8-dv8*:
8 dvd len-of TYPE(*'a*)
⟨*proof*⟩

lemma *len8-pow*:
∃*k*. *len-of TYPE*(*'a*) *div 8* = *2^k*
⟨*proof*⟩

end

fun
nat-to-bin-string :: *nat* ⇒ *char list*
where
ntbs: *nat-to-bin-string n* = (*if* (*n* = *0*) *then* *''0''* *else* (*if n mod 2* = *1 then CHR* *''1''* *else CHR* *''0''*) # *nat-to-bin-string* (*n div 2*))

declare *nat-to-bin-string.simps* [*simp del*]

lemma *nat-to-bin-string-simps*:
nat-to-bin-string 0 = *''0''*
n > *0* ⟹ *nat-to-bin-string n* =
(*if n mod 2* = *1 then CHR* *''1''* *else CHR* *''0''*) # *nat-to-bin-string* (*n div 2*)
⟨*proof*⟩

instance *signed* :: (*len8*) *len8*
⟨*proof*⟩

end

theory *Word-Mem-Encoding-ARM*
imports *../Vanilla32-Preliminaries*
begin

if-architecture-context (*ARM*)
begin

definition
word-tag :: *'a*::*len8 word itself* ⇒ *'a word xtyp-info*
where
word-tag (*w*::*'a*::*len8 word itself*) ≡
TypDesc (*len-exp TYPE*(*'a*)) (*TypScalar* (*len-of TYPE*(*'a*) *div* 8) (*len-exp TYPE*(*'a*))
(⦇*field-access* = λ*v bs*. (*rev o word-rsplit*) *v*,
field-update = λ*bs v*. (*word-rcat* (*rev* (*take* (*len-of TYPE*(*'a*) *div* 8) *bs*))::*'a*::*len8 word*),
field-sz = (*len-of TYPE*(*'a*) *div* 8)⦈)
(*''word''* @ *nat-to-bin-string* (*len-of TYPE*(*'a*)))
end

end

theory *Word-Mem-Encoding-ARM64*
imports *Vanilla32-Preliminaries*
begin

if-architecture-context (*ARM64*)
begin

definition
word-tag :: *'a*::*len8 word itself* ⇒ *'a word xtyp-info*
where
word-tag (*w*::*'a*::*len8 word itself*) ≡
TypDesc (*len-exp TYPE*(*'a*)) (*TypScalar* (*len-of TYPE*(*'a*) *div* 8)
(*len-exp TYPE*(*'a*))
(⦇*field-access* = λ*v bs*. (*rev o word-rsplit*) *v*,
field-update = λ*bs v*. (*word-rcat* (*rev* (*take* (*len-of TYPE*(*'a*) *div* 8) *bs*))::*'a*::*len8 word*),
field-sz = (*len-of TYPE*(*'a*) *div* 8)⦈)
(*''word''* @ *nat-to-bin-string* (*len-of TYPE*(*'a*)))
end

end

theory *Word-Mem-Encoding-ARM-HYP*
imports *../Vanilla32-Preliminaries*
begin

if-architecture-context (*ARM-HYP*)
begin

definition
word-tag :: *'a::len8 word itself* ⇒ *'a word xtyp-info*
where
word-tag (*w::'a::len8 word itself*) ≡
TypDesc (*len-exp TYPE('a)*) (*TypScalar* (*len-of TYPE('a) div 8*) (*len-exp TYPE('a)*)
(⦇*field-access* = λ*v bs*. (*rev o word-rsplit*) *v*,
field-update = λ*bs v*. (*word-rcat* (*rev* (*take* (*len-of TYPE('a) div 8*) *bs*))::*'a::len8 word*),
field-sz = (*len-of TYPE('a) div 8*)⦈)
(*''word''* @ *nat-to-bin-string* (*len-of TYPE('a)*))
end

end

theory *Word-Mem-Encoding-RISCV64*
imports *../Vanilla32-Preliminaries*
begin

if-architecture-context (*RISCV64*)
begin

definition
word-tag :: *'a::len8 word itself* ⇒ *'a word xtyp-info*
where
word-tag (*w::'a::len8 word itself*) ≡
TypDesc (*len-exp TYPE('a)*) (*TypScalar* (*len-of TYPE('a) div 8*)
(*len-exp TYPE('a)*)
(⦇*field-access* = λ*v bs*. (*rev o word-rsplit*) *v*,
field-update = λ*bs v*. (*word-rcat* (*rev* (*take* (*len-of TYPE('a) div 8*) *bs*))::*'a::len8 word*),
field-sz = (*len-of TYPE('a) div 8*)⦈)
(*''word''* @ *nat-to-bin-string* (*len-of TYPE('a)*)))
end

end

theory *Word-Mem-Encoding-X64*
imports *../Vanilla32-Preliminaries*
begin

if-architecture-context (*X64*)
begin

definition
word-tag :: *'a::len8 word itself* ⇒ *'a word xtyp-info*
where
word-tag (*w::'a::len8 word itself*) ≡
TypDesc (*len-exp TYPE('a)*) (*TypScalar* (*len-of TYPE('a) div 8*)
(*len-exp TYPE('a)*)
(⦇*field-access* = λ*v bs*. (*rev o word-rsplit*) *v*,
field-update = λ*bs v*. (*word-rcat* (*rev* (*take* (*len-of TYPE('a)*
div 8) *bs*))::*'a::len8 word*),
field-sz = (*len-of TYPE('a) div 8*)⦈))
(*''word''* @ *nat-to-bin-string* (*len-of TYPE('a)*))
end

end

theory *Word-Mem-Encoding*
imports
ARM/Word-Mem-Encoding-ARM
ARM64/Word-Mem-Encoding-ARM64
ARM-HYP/Word-Mem-Encoding-ARM-HYP
RISCV64/Word-Mem-Encoding-RISCV64
X64/Word-Mem-Encoding-X64
begin

end

theory *Vanilla32*
imports *Word-Mem-Encoding CTypes*
begin

type-synonym *exit-status* = *32 sword*

instantiation *word* :: (*len8*) *c-type*
begin
definition *typ-name-itself* (*w::'a::len8 word itself*) =
''word'' @ *nat-to-bin-string* (*len-of TYPE('a)*)
definition
typ-info-word: *typ-info-t* (*w::'a::len8 word itself*) ≡ *word-tag w*

instance ⟨*proof*⟩
end

lemma *align-of-word*:
align-of TYPE('*a*::*len8 word*) = *len-of TYPE*('*a*) *div 8*
⟨*proof*⟩

instantiation *word* :: (*len8*) *mem-type*
begin

instance
⟨*proof*⟩

end

instantiation *word* :: (*len8*) *simple-mem-type*
begin

instance
⟨*proof*⟩

end

class *len-eq1* = *len* +
assumes *len-eq1*: *len-of TYPE*('*a*::*len*) = *1*
instance *num1* :: *len-eq1*
⟨*proof*⟩

class *len-lg01* = *len* +
assumes *len-lg01*: *len-of TYPE*('*a*::*len*) ∈ {*1*,*2*}
instance *bit0* :: (*len-eq1*) *len-lg01*
⟨*proof*⟩
instance *num1* :: *len-lg01*
⟨*proof*⟩

class *len-lg02* = *len* +
assumes *len-lg02*: *len-of TYPE*('*a*::*len*) ∈ {*1*,*2*,*4*}
instance *bit0* :: (*len-lg01*) *len-lg02*
⟨*proof*⟩
instance *num1* :: *len-lg02*
⟨*proof*⟩

class *len-lg03* = *len* +
assumes *len-lg03*: *len-of TYPE*('*a*::*len*) ∈ {*1*,*2*,*4*,*8*}
instance *bit0* :: (*len-lg02*) *len-lg03*
⟨*proof*⟩
instance *num1* :: *len-lg03*
⟨*proof*⟩

class *len-lg04* = *len* +
  assumes *len-lg04*: *len-of TYPE*('*a*::*len*) ∈ {*1*,*2*,*4*,*8*,*16*}
instance *bit0* :: (*len-lg03*) *len-lg04*
⟨*proof*⟩

instance *num1* :: *len-lg04*
⟨*proof*⟩
class *len-lg15* = *len* +
  assumes *len-lg15*: *len-of TYPE*('*a*::*len*) ∈ {*2*,*4*,*8*,*16*,*32*}
instance *bit0* :: (*len-lg04*) *len-lg15*
⟨*proof*⟩

class *len-lg26* = *len* +
  assumes *len-lg26*: *len-of TYPE*('*a*::*len*) ∈ {*4*,*8*,*16*,*32*,*64*}

instance *bit0* :: (*len-lg15*) *len-lg26*
⟨*proof*⟩

instance *bit0* :: (*len-lg26*) *len8*
⟨*proof*⟩

instance *signed* :: (*len-eq1*) *len-eq1* ⟨*proof*⟩
instance *signed* :: (*len-lg01*) *len-lg01* ⟨*proof*⟩
instance *signed* :: (*len-lg02*) *len-lg02* ⟨*proof*⟩
instance *signed* :: (*len-lg03*) *len-lg03* ⟨*proof*⟩
instance *signed* :: (*len-lg04*) *len-lg04* ⟨*proof*⟩
instance *signed* :: (*len-lg15*) *len-lg15* ⟨*proof*⟩
instance *signed* :: (*len-lg26*) *len-lg26* ⟨*proof*⟩

lemma
  to-bytes (*1***256***256***256* + *2***256***256* + *3***256* + (*4*::*32 word*)) *bs* = [*4*, *3*, *2*, *1*]
  ⟨*proof*⟩

lemma *size-td-words* [*simp*]:
  size-td (*typ-info-t TYPE*(*8 word*)) = *1*
  size-td (*typ-info-t TYPE*(*16 word*)) = *2*
  size-td (*typ-info-t TYPE*(*32 word*)) = *4*
  size-td (*typ-info-t TYPE*(*64 word*)) = *8*
  size-td (*typ-info-t TYPE*(*128 word*)) = *16*
  ⟨*proof*⟩

lemma *align-td-words* [*simp*]:
  align-td (*typ-info-t TYPE*(*8 word*)) = *0*
  align-td (*typ-info-t TYPE*(*16 word*)) = *1*
  align-td (*typ-info-t TYPE*(*32 word*)) = *2*
  align-td (*typ-info-t TYPE*(*64 word*)) = *3*
  align-td (*typ-info-t TYPE*(*128 word*)) = *4*

⟨*proof*⟩

lemma *size-of-words* [*simp*]:
size-of TYPE(*8 word*) = *1*
size-of TYPE(*16 word*) = *2*
size-of TYPE(*32 word*) = *4*
size-of TYPE(*64 word*) = *8*
size-of TYPE(*128 word*) = *16*
⟨*proof*⟩

lemma *align-of-words* [*simp*]:
align-of TYPE(*8 word*) = *1*
align-of TYPE(*16 word*) = *2*
align-of TYPE(*32 word*) = *4*
align-of TYPE(*64 word*) = *8*
align-of TYPE(*128 word*) = *16*
⟨*proof*⟩

lemma *size-td-swords* [*simp*]:
size-td (*typ-info-t TYPE*(*8 sword*)) = *1*
size-td (*typ-info-t TYPE*(*16 sword*)) = *2*
size-td (*typ-info-t TYPE*(*32 sword*)) = *4*
size-td (*typ-info-t TYPE*(*64 sword*)) = *8*
size-td (*typ-info-t TYPE*(*128 sword*)) = *16*
⟨*proof*⟩

lemma *align-td-swords* [*simp*]:
align-td (*typ-info-t TYPE*(*8 sword*)) = *0*
align-td (*typ-info-t TYPE*(*16 sword*)) = *1*
align-td (*typ-info-t TYPE*(*32 sword*)) = *2*
align-td (*typ-info-t TYPE*(*64 sword*)) = *3*
align-td (*typ-info-t TYPE*(*128 sword*)) = *4*
⟨*proof*⟩

lemma *size-of-swords* [*simp*]:
size-of TYPE(*8 sword*) = *1*
size-of TYPE(*16 sword*) = *2*
size-of TYPE(*32 sword*) = *4*
size-of TYPE(*64 sword*) = *8*
size-of TYPE(*128 sword*) = *16*
⟨*proof*⟩

lemma *align-of-swords* [*simp*]:
align-of TYPE(*8 sword*) = *1*
align-of TYPE(*16 sword*) = *2*
align-of TYPE(*32 sword*) = *4*
align-of TYPE(*64 sword*) = *8*
align-of TYPE(*128 sword*) = *16*
⟨*proof*⟩

instantiation *ptr* :: (*c-type-name*) *c-type*
begin
definition *typ-name-itself* (*p*::'*a*::*c-type-name ptr itself*) = *typ-name-itself TYPE*('*a*) @ ''+*ptr*''
definition
typ-info-ptr:
typ-info-t (*p*::'*a*::*c-type-name ptr itself*) ≡ *TypDesc addr-align*
(*TypScalar* (*addr-bitsize div 8*) *addr-align* ⦇
field-access = λ*p bs*. *rev* (*word-rsplit* (*ptr-val p*)),
field-update = λ*bs v*. *Ptr* (*word-rcat* (*rev* (*take* (*addr-bitsize div 8*) *bs*))::*addr*),
field-sz = (*addr-bitsize div 8*) ⦈)
(*typ-name-itself TYPE*('*a*) @ ''+*ptr*'')

instance ⟨*proof*⟩
end

lemma *align-of-ptr* [*simp*]:
align-of (*p*::'*a*::*c-type ptr itself*) = 2 ^ *addr-align*
⟨*proof*⟩

instantiation *ptr* :: (*c-type*) *mem-type*
begin
instance
⟨*proof*⟩
end

lemma *div-eq*: *8 dvd n* ⟹ (*7* + *n*::*nat*) *div 8* = *n div 8*
⟨*proof*⟩

lemma *length-word-rsplit-len8*: *length* ((*word-rsplit*::('*a*::*len8 word*) ⇒ *8 word list*) *w*) = (*LENGTH*('*a*) *div* (*8*::*nat*))
⟨*proof*⟩

lemma *all-value-byte-word*: *i* < *LENGTH*('*a*::*len8*) *div 8* ⟹
padding-base.is-value-byte (λ*v bs*. *rev* (*word-rsplit v*))
(λ*bs v*. *word-rcat* (*rev* (*take* (*LENGTH*('*a*) *div 8*) *bs*))::'*a word*)
(*LENGTH*('*a*) *div 8*) *i*
⟨*proof*⟩

lemma *no-padding-byte-word*: *i* < *LENGTH*('*a*::*len8*) *div 8* ⟹
¬ *padding-base.is-padding-byte* (λ*v bs*. *rev* (*word-rsplit v*))
(λ*bs v*. *word-rcat* (*rev* (*take* (*LENGTH*('*a*) *div 8*) *bs*))::'*a word*) (*LENGTH*('*a*) *div 8*)
i
⟨*proof*⟩

instantiation *word* :: (*len8*) *xmem-contained-type*
begin
instance
⟨*proof*⟩
end

instantiation *ptr* :: (*c-type*) *simple-mem-type*
begin
instance
⟨*proof*⟩
end

lemma *all-value-byte-ptr*: $\bigwedge i.\ i < (addr\text{-}bitsize\ div\ 8) \Longrightarrow$
padding-base.is-value-byte (λ*p bs. rev* (*word-rsplit* (*ptr-val p*)))
(λ*bs v. PTR*('*a*) (*word-rcat* (*rev* (*take* (*addr-bitsize div 8*) *bs*))))
(*addr-bitsize div 8*) *i*
⟨*proof*⟩

lemma *no-padding-byte-ptr*: $\bigwedge i.\ i < (addr\text{-}bitsize\ div\ 8) \Longrightarrow$
¬ *padding-base.is-padding-byte* (λ*p bs. rev* (*word-rsplit* (*ptr-val p*)))
(λ*bs v. PTR*('*a*::*c-type*) (*word-rcat* (*rev* (*take* (*addr-bitsize div 8*) *bs*))))
(*addr-bitsize div 8*) *i*
⟨*proof*⟩

instantiation *ptr* :: (*c-type*) *xmem-contained-type*
begin
instance
⟨*proof*⟩
end

lemma *size-td-ptr* [*simp*]:
size-td (*typ-info-t TYPE*('*a*::*c-type ptr*)) = *addr-bitsize div 8*
⟨*proof*⟩

lemma *size-of-ptr* [*simp*]:
size-of TYPE('*a*::*c-type ptr*) = *addr-bitsize div 8*
⟨*proof*⟩

lemma *align-td-ptr* [*simp*]: *align-td* (*typ-info-t TYPE*('*a*::*c-type ptr*)) = *addr-align*
⟨*proof*⟩

lemma *ptr-add-word32-signed* [*simp*]:
fixes *a* :: *32 word ptr*
shows *ptr-val* (*a* +$_p$ *x*) = *ptr-val a* + *4* ∗ *of-int x*
⟨*proof*⟩

lemma *ptr-add-word32* [*simp*]:

fixes *a* :: *32 word ptr*
shows *ptr-val* (*a* +$_p$ *uint x*) = *ptr-val a* + *4* ∗ *x*
⟨*proof*⟩

lemma *ptr-add-word64-signed* [*simp*]:
fixes *a* :: *64 word ptr*
shows *ptr-val* (*a* +$_p$ *x*) = *ptr-val a* + *8* ∗ *of-int x*
⟨*proof*⟩

lemma *ptr-add-word64* [*simp*]:
fixes *a* :: *64 word ptr*
shows *ptr-val* (*a* +$_p$ *uint x*) = *ptr-val a* + *8* ∗ *x*
⟨*proof*⟩

lemma *ptr-add-0-id*[*simp*]:*x* +$_p$ *0* = *x*
⟨*proof*⟩

lemma *from-bytes-ptr-to-bytes-ptr*:
from-bytes (*to-bytes* (*v*::*addr-bitsize word*) *bs*) = (*Ptr v* :: *'a*::*c-type ptr*)
⟨*proof*⟩

lemma *ptr-aligned-coerceI*:
ptr-aligned (*ptr-coerce x*::*addr ptr*) ⟹ *ptr-aligned* (*x*::*'a*::*mem-type ptr ptr*)
⟨*proof*⟩

lemma *lift-ptr-ptr* [*simp*]:
⋀*p*::*'a*::*mem-type ptr ptr*.
lift h (*ptr-coerce p*) = *ptr-val* (*lift h p*)
⟨*proof*⟩
thm *typ-name.simps*

lemma *typ-name-itself-ptr*:
typ-name-itself (*T*::*'a*::*c-type ptr itself*) = *typ-name-itself TYPE*(*'a*) @ *''+ptr''*
⟨*proof*⟩

lemma *typ-name-itself-word*:
typ-name-itself (*T*::*'a*::*len8 word itself*) = *typ-name* (*typ-info-t T*)
⟨*proof*⟩

lemmas *Vanilla32-tags* [*simp*] =
typ-info-word
typ-info-ptr
typ-name-itself-ptr
word-tag-def
typ-name-itself-word

lemma *ptr-typ-name* [*simp*]:
typ-name (*typ-info-t TYPE*((*'a* :: *c-type*) *ptr*)) = *typ-name-itself TYPE*(*'a*) @ *''+ptr''*

⟨*proof*⟩

lemma *word-typ-name* [*simp*]:
typ-name (*typ-info-t TYPE*('*a*::*len8 word*)) = ''*word*'' @ *nat-to-bin-string* (*len-of TYPE*('*a*))
⟨*proof*⟩

lemma *typ-name-words* [*simp*]:
typ-name (*typ-uinfo-t TYPE*(*8 word*)) = ''*word00010*''
typ-name (*typ-uinfo-t TYPE*(*16 word*)) = ''*word000010*''
typ-name (*typ-uinfo-t TYPE*(*32 word*)) = ''*word0000010*''
typ-name (*typ-uinfo-t TYPE*(*64 word*)) = ''*word00000010*''
typ-name (*typ-uinfo-t TYPE*(*128 word*)) = ''*word000000010*''
typ-name (*typ-info-t TYPE*(*8 word*)) = ''*word00010*''
typ-name (*typ-info-t TYPE*(*16 word*)) = ''*word000010*''
typ-name (*typ-info-t TYPE*(*32 word*)) = ''*word0000010*''
typ-name (*typ-info-t TYPE*(*64 word*)) = ''*word00000010*''
typ-name (*typ-info-t TYPE*(*128 word*)) = ''*word000000010*''
⟨*proof*⟩

lemma *typ-name-swords* [*simp*]:
typ-name (*typ-uinfo-t TYPE*(*8 sword*)) = ''*word00010*''
typ-name (*typ-uinfo-t TYPE*(*16 sword*)) = ''*word000010*''
typ-name (*typ-uinfo-t TYPE*(*32 sword*)) = ''*word0000010*''
typ-name (*typ-uinfo-t TYPE*(*64 sword*)) = ''*word00000010*''
typ-name (*typ-uinfo-t TYPE*(*128 sword*)) = ''*word000000010*''
typ-name (*typ-info-t TYPE*(*8 sword*)) = ''*word00010*''
typ-name (*typ-info-t TYPE*(*16 sword*)) = ''*word000010*''
typ-name (*typ-info-t TYPE*(*32 sword*)) = ''*word0000010*''
typ-name (*typ-info-t TYPE*(*64 sword*)) = ''*word00000010*''
typ-name (*typ-info-t TYPE*(*128 sword*)) = ''*word000000010*''
⟨*proof*⟩

lemma *ptr-arith*[*simp*]:
(*x* +$_p$ *a* = *y* +$_p$ *a*) = ((*x*::('*a*::*c-type*) *ptr*) = (*y*::'*a ptr*))
⟨*proof*⟩

lemma *ptr-arith'*[*simp*]:
(*ptr-coerce* (*x* +$_p$ *a*) = *ptr-coerce* (*y* +$_p$ *a*)) = ((*x*::('*a*::*c-type*) *ptr*) = (*y*::'*a ptr*))
⟨*proof*⟩

lemma *typ-uinfo-t-signed-word-word-conv*: *typ-uinfo-t TYPE*(('*a*::*len8*) *signed word*) = *typ-uinfo-t TYPE*('*a word*)
⟨*proof*⟩

end

theory *ML-Infer-Instantiate*
imports *Main*
begin

11.13 Instantiation, inferring type instantiation from term instantiation.

⟨*ML*⟩

This is a variant of of the instantiate antiquotation, e.g.: `\<^instantiate>\<open>'a=\<^typ>\<o`
`and x=\<^term>\<open>1::nat\<close> in term \<open>x + x\<close>`
`for x::\<open>'a::plus\<close>\<close>`.

The instantiate antiquotation requires the explicit instantiation of type-variables, including those that can be inferred from the term instantiation. This makes the code very explicit, predictable and efficient as no types have to be calculated at runtime or be 'guessed' by the reader. On the other hand it can make the resulting code quite verbose when the number of type variables increases.

In this theory we provide a variant, that allows to omit those type-instantiations that are already fixed by the given term instantiations. This follows the same idea as e.g. `Drule.infer_instantiate`.

Moreover, we provide a variant that also takes a morphism into account. The use-case here is that the instantiate expression is written within a locale context but later on is used in an interpretation of the locale (given by morphism phi). Before instantiation the term is exported (according to morphism phi).

⟨*ML*⟩

Here an example with explicit instantiation.

⟨*ML*⟩

With the new variant the type instantiation can be omitted as it is already inferred from the term instantiations. Note that the result of the antiquotation is function that expects a `Proof.context`. This is used to perform the necessary type inference and provide context sensitive error messages.

⟨*ML*⟩

This can be shortened:

⟨*ML*⟩

Here is an example were not all type variables can be inferred from the term instantiation. Those types can be explicitly instantiated.

⟨*ML*⟩

context
fixes *x*::*'a*::*plus*

begin
⟨*ML*⟩
end

Note that the antiquotation only makes some local type-inference on the term parameters alone. There might be further use cases to perform a type inference on the whole term instead. This might be further fine-tuned by inserting explicit dummy types to be inferred: -.

⟨*ML*⟩

end

theory *Arrays*
imports
HOL−Library.Numeral-Type
Match-Cterm
ML-Infer-Instantiate
begin

definition *has-size* :: *'a set* ⇒ *nat* ⇒ *bool* **where**
has-size s n = (*finite s* ∧ *card s* = *n*)

— If *'a* is not finite, there is no *n* < *CARD*(*'a*)
definition *finite-index* :: *nat* ⇒ *'a*::*finite* **where**
finite-index = (*SOME f*. ∀ *x*. ∃!*n*. *n* < *CARD*(*'a*) ∧ *f n* = *x*)

lemma *card-image-inj*:
⟦ *finite S*; ⋀*x y*. ⟦ *x* ∈ *S*; *y* ∈ *S*; *f x* = *f y* ⟧ ⟹ *x* = *y* ⟧ ⟹ *card* (*f* ‘ *S*) = *card S*
⟨*proof*⟩

lemma *has-size-image-inj*:
⟦ *has-size S n*; ⋀*x y*. *x* ∈ *S* ∧ *y* ∈ *S* ∧ *f x* = *f y* ⟹ *x* = *y* ⟧ ⟹ *has-size* (*f* ‘ *S*) *n*
⟨*proof*⟩

lemma *has-size-0*[*simp*]:
has-size S 0 = (*S* = {})
⟨*proof*⟩

lemma *has-size-suc*:
has-size S (*Suc n*) = (*S* ≠ {} ∧ (∀ *a*. *a* ∈ *S* ⟶ *has-size* (*S* − {*a*}) *n*))
⟨*proof*⟩

lemma *has-index*:
⟦ *finite S*; *card S* = *n* ⟧ ⟹

$(\exists f.\ (\forall m.\ m < n \longrightarrow f\ m \in S) \wedge (\forall x.\ x{\in}S \longrightarrow (\exists!m.\ m < n \wedge f\ m = x)))$
⟨*proof*⟩

lemma *finite-index-works*:
$\exists!n.\ n < CARD('n{::}finite) \wedge finite\text{-}index\ n = (i{::}'n)$
⟨*proof*⟩

lemma *finite-index-inj*:
$[\![\ i < CARD('a{::}finite);\ j < CARD('a)\ ]\!] \Longrightarrow$
$((finite\text{-}index\ i :: 'a) = finite\text{-}index\ j) = (i = j)$
⟨*proof*⟩

lemma *forall-finite-index*:
$(\forall k{::}'a{::}finite.\ P\ k) = (\forall i.\ i < CARD('a) \longrightarrow P\ (finite\text{-}index\ i))$
⟨*proof*⟩

11.14 Finite Cartesian Products

typedef $('a,'n{::}finite)$ *array*
(‹(‹*open-block notation=‹mixfix array››-[-]*)› [*30*,*0*] *31*) = *UNIV* :: $('n \Rightarrow 'a)$ *set*
⟨*proof*⟩
setup-lifting *type-definition-array*

lift-definition *index* :: $('a,'n{::}finite)$ *array* ⇒ *nat* ⇒ $'a$
(‹(‹*open-block notation=‹mixfix array index››-.[-]*)› [*900*,*0*] *901*) **is**
$\lambda f\ i.\ f\ (finite\text{-}index\ i)$⟨*proof*⟩

lemma *index-legacy-def*: *index x i* = *Rep-array x* (*finite-index i*)
⟨*proof*⟩

theorem *array-index-eq*:
$((x{::}'a['n{::}finite]) = y) = (\forall i.\ i < CARD('n) \longrightarrow x.[i] = y.[i])$
⟨*proof*⟩

lemmas *cart-eq* = *array-index-eq*

lemma *array-ext*:
fixes $x :: 'a['n{::}finite]$
shows $(\bigwedge i.\ i < CARD('n) \Longrightarrow x.[i] = y.[i]) \Longrightarrow x = y$
⟨*proof*⟩

definition *FCP* :: $(nat \Rightarrow 'a) \Rightarrow 'a['b{::}finite]$ (**binder** ‹*ARRAY* › *10*) **where**
$FCP \equiv \lambda g.\ SOME\ a.\ \forall i.\ i < CARD('b) \longrightarrow a.[i] = g\ i$

definition *update* :: $'a['n{::}finite] \Rightarrow nat \Rightarrow 'a \Rightarrow 'a['n]$ **where**
$update\ f\ i\ x \equiv FCP\ ((index\ f)(i := x))$

definition *fupdate* :: *nat* ⇒ (*'a* ⇒ *'a*) ⇒ *'a*['*b*::*finite*] ⇒ *'a*['*b*] **where**
fupdate i f x ≡ *update x i* (*f* (*index x i*))

lemma *fcp-beta*[*simp*]: *i* < *CARD*('*n* :: *finite*) ⟹ (*FCP g* :: *'a*['*n*]).[*i*] = *g i*
⟨*proof*⟩

lemma *fcp-unique*:
(∀ *i*. *i* < *CARD*('*b*::*finite*) ⟶ *index f i* = *g i*) =
(*FCP g* = (*f* :: ('*a*,'*b*) *array*))
⟨*proof*⟩

lemma *fcp-eta*[*simp*]:
(*ARRAY i*. *g*.[*i*]) = *g*
⟨*proof*⟩

lemma *index-update*[*simp*]:
n < *CARD*('*b*::*finite*) ⟹ *index* (*update* (*f*::*'a*['*b*]) *n x*) *n* = *x*
⟨*proof*⟩

lemma *index-update2*[*simp*]:
⟦ *k* < *CARD*('*b*::*finite*); *n* ≠ *k* ⟧ ⟹ *index* (*update* (*f*::*'a*['*b*]) *n x*) *k* = *index f k*
⟨*proof*⟩

lemma *update-update*[*simp*]:
update (*update f n x*) *n y* = *update f n y*
⟨*proof*⟩

lemma *update-comm*[*simp*]:
n ≠ *m* ⟹ *update* (*update f m v*) *n v'* = *update* (*update f n v'*) *m v*
⟨*proof*⟩

lemma *update-index-same* [*simp*]:
update v n (*index v n*) = *v*
⟨*proof*⟩

function *foldli0* :: (*nat* ⇒ *'acc* ⇒ *'a* ⇒ *'acc*) ⇒ *'acc* ⇒ *nat* ⇒ *'a*['*index*::*finite*]
⇒ *'acc* **where**
foldli0 f a i arr = (*if CARD*('*index*) ≤ *i then a else foldli0 f* (*f i a* (*index arr i*))
(*i*+*1*) *arr*)
⟨*proof*⟩

termination
⟨*proof*⟩

definition *foldli* :: (*nat* ⇒ *'acc* ⇒ *'a* ⇒ *'acc*) ⇒ *'acc* ⇒ ('*a*,'*i*::*finite*) *array* ⇒
'acc **where**
foldli f a arr = *foldli0 f a 0 arr*

definition *map-array* :: $('a \Rightarrow 'b) \Rightarrow 'a['n::finite] \Rightarrow 'b['n]$ **where**
map-array f a $\equiv$ *ARRAY i. f* (*a.*[*i*])

definition *map-array2* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a['n::finite] \Rightarrow 'b['n] \Rightarrow 'c['n]$ **where**
map-array2 f a b $\equiv$ *ARRAY i. f* (*a.*[*i*]) (*b.*[*i*])

definition *zip-array* :: $'a['b::finite] \Rightarrow 'c['b] \Rightarrow ('a \times 'c)['b]$ **where**
zip-array $\equiv$ *map-array2 Pair*

definition *list-array* :: ('*a*,'*n*::*finite*) *array* $\Rightarrow$ '*a list* **where**
list-array a = *map* (*index a*) [*0*..<*CARD*('*n*)]

lift-definition *set-array* :: $'a['n::finite] \Rightarrow 'a$ *set* **is** *range* ⟨*proof*⟩

lemma *set-array-list*:
set-array a = *set* (*list-array a*)
⟨*proof*⟩

definition *rel-array* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a['n::finite] \Rightarrow 'b['n] \Rightarrow bool$ **where**
rel-array f a b $\equiv \forall i <$ *CARD*('*n*). *f* (*a.*[*i*]) (*b.*[*i*])

lemma *map-array-index*:
fixes *a* :: $'a['n::finite]$
shows $n <$ *CARD*('*n*) $\Longrightarrow$ (*map-array f a*).[*n*] = *f* (*a.*[*n*])
⟨*proof*⟩

lemma *map-array2-index*:
fixes *a* :: $'a['n::finite]$
shows $n <$ *CARD*('*n*) $\Longrightarrow$ (*map-array2 f a b*).[*n*] = *f* (*a.*[*n*]) (*b.*[*n*])
⟨*proof*⟩

lemma *zip-array-index*:
fixes *a* :: $'a['n::finite]$
shows $n <$ *CARD*('*n*) $\Longrightarrow$ (*zip-array a b*).[*n*] = (*a.*[*n*],*b.*[*n*])
⟨*proof*⟩

lemma *map-array-id*[*simp*]:
map-array id = *id*
⟨*proof*⟩

lemma *map-array-comp*:
map-array (*g* ∘ *f*) = *map-array g* ∘ *map-array f*
⟨*proof*⟩

lemma *list-array-nth*:
fixes *a* :: $'a['n::finite]$
shows $n <$ *CARD*('*n*) $\Longrightarrow$ *list-array a* ! *n* = *index a n*
⟨*proof*⟩

lemma *list-array-length* [*simp*]:
length (*list-array* (*a* :: *'a*['*n*::*finite*])) = *CARD*('*n*)
⟨*proof*⟩

lemma *in-set-array-index-conv*:
(*z* ∈ *set-array* (*a* :: *'a*['*n*::*finite*])) = (∃ *n* < *CARD*('*n*). *z* = *a*.[*n*])
⟨*proof*⟩

lemma *in-set-arrayE* [*elim*!]:
⟦ *z* ∈ *set-array* (*a* :: *'a*['*n*::*finite*]); ⋀*n* . ⟦*n* < *CARD*('*n*); *z* = *a*.[*n*]⟧ ⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

lemma *map-array-setI*:
(⋀*z*. *z* ∈ *set-array x* ⟹ *f z* = *g z*) ⟹ *map-array f x* = *map-array g x*
⟨*proof*⟩

lemma *list-array-map-array*:
list-array (*map-array f a*) = *map f* (*list-array a*)
⟨*proof*⟩

lemma *list-array-FCP* [*simp*]:
list-array (*FCP f* :: *'a*['*n*]) = *map f* [*0*..<*CARD*('*n*::*finite*)]
⟨*proof*⟩

lemma *set-array-FCP* [*simp*]:
set-array (*FCP f* :: *'a*['*n*]) = *f* ‘ {*0*..< *CARD*('*n*::*finite*)}
⟨*proof*⟩

lemma *map-array-set-img*:
set-array ∘ *map-array f* = (‘) *f* ∘ *set-array*
⟨*proof*⟩

lemma *fcp-cong* [*cong*]:
(⋀*i*. *i* < *CARD*('*n*::*finite*) ⟹ *f i* = *g i*) ⟹ *FCP f* = (*FCP g* :: *'a*['*n*])
⟨*proof*⟩

bnf (*'a*,*'n*::*finite*) *array*
map: *map-array*
sets: *set-array*
bd: *card-suc* (*BNF-Cardinal-Arithmetic.csum natLeq* (*card-of* (*UNIV* :: *'n set*)))
rel: *rel-array*
⟨*proof*⟩

lemma *arr-fupdate-same*: *i* < *CARD*('*n*) ⟹
index (*fupdate i f* (*x*::*'a*['*n*::*finite*])) *i* = *f* (*index x i*)
⟨*proof*⟩

lemma *arr-fupdate-other*: $j < CARD('n) \Longrightarrow i \neq j \Longrightarrow$
index (*fupdate i f* (x::$'a['n$::*finite*])) j = (*index x j*)
⟨*proof*⟩

lemmas *arr-fupdate-simps* = *arr-fupdate-same arr-fupdate-other*

lemma *surj-finite-index*: *surj finite-index*
⟨*proof*⟩

lemma *finite-index-inj-on*: *inj-on* (*finite-index*::*nat*⇒$'a$::*finite*) {i. $i < CARD('a)$}
⟨*proof*⟩

lemma *finite-index-bij-betw*: *bij-betw* (*finite-index*::*nat*⇒$'a$::*finite*) {i. $i < CARD('a)$}
UNIV
⟨*proof*⟩

⟨*ML*⟩

experiment
begin
lemma *Arrays.update a n* (k − *a*.[*n*] + x) = *fupdate n* (λv. k − v + x) *a*
⟨*proof*⟩
end
end

theory *Padding*
imports *Main*
begin

lemma (*0*::*nat*) *mod 2* = *XXX*
⟨*proof*⟩

lemma (*2* − (*0*::*nat*) *mod 2*) = *XXX*
⟨*proof*⟩

lemma (*2* − (*0*::*nat*) *mod 2*) *mod 2* = *XXX*
⟨*proof*⟩

definition *padup* :: *nat* ⇒ *nat* ⇒ *nat* **where**
padup align n ≡ (*align* − *n mod align*) *mod align*

lemma *padup-dvd*:
$0 < b \Longrightarrow$ (*padup b n* = *0*) = (*b dvd n*)
⟨*proof*⟩

lemma *dvd-padup-add*:

$0 < x \Longrightarrow x$ *dvd* $y +$ *padup* $x\ y$
⟨*proof*⟩

end

theory *Lens*
imports *Main Distinct-Prop*
begin

11.15 Auxiliary Theorems

lemma *distinct-prop-cons*:
list-all $(R\ x)\ xs \Longrightarrow$ *distinct-prop* $R\ xs \Longrightarrow$ *distinct-prop* $R\ (x \#\ xs)$
⟨*proof*⟩

lemma *distinct-prop-distrib-all*:
distinct-prop $(\lambda x\ y.\ \forall\, a.\ R\ a\ x\ y)\ xs \longleftrightarrow (\forall\, a.$ *distinct-prop* $(R\ a)\ xs)$
⟨*proof*⟩

lemma *pairwise-set-of-distinct-prop*:
$(\bigwedge a\ b.\ R\ a\ b \longleftrightarrow R\ b\ a) \Longrightarrow$ *distinct-prop* $R\ xs \Longrightarrow$ *pairwise* R (*set* xs)
⟨*proof*⟩

lemma *distinct-prop-iff-nth*:
distinct-prop $R\ xs \longleftrightarrow (\forall\, i\ j.\ i < j \longrightarrow j <$ *length* $xs \longrightarrow R\ (xs!i)\ (xs!j))$
⟨*proof*⟩

lemma *distinct-prop-mono*:
$(\bigwedge x\ y.\ x \in$ *set* $xs \Longrightarrow y \in$ *set* $xs \Longrightarrow P\ x\ y \Longrightarrow R\ x\ y) \Longrightarrow$
distinct-prop $P\ xs \Longrightarrow$ *distinct-prop* $R\ xs$
⟨*proof*⟩

lemma *distinct-prop-nil*: *distinct-prop* R []
⟨*proof*⟩

lemma *list-all-concat*: *list-all* p (*concat* xs) = *list-all* (*list-all* p) xs
⟨*proof*⟩

lemma *list-all-conj*: *list-all* $P\ xs \Longrightarrow$ *list-all* $Q\ xs \Longrightarrow$ *list-all* $(\lambda x.\ P\ x \wedge Q\ x)\ xs$
⟨*proof*⟩

lemma *list-all-zip-iff-list-all2*:
length as = *length* $bs \Longrightarrow$ *list-all* R (*zip* $as\ bs) \longleftrightarrow$ *list-all2* $(\lambda a\ b.\ R\ (a,\ b))\ as\ bs$
⟨*proof*⟩

lemma *disjnt-comm*: *disjnt* $A\ B \longleftrightarrow$ *disjnt* $B\ A$
⟨*proof*⟩

lemma *disjnt-image*: *disjnt* (*f* ‘ *x*) (*f* ‘ *y*) ⟹ *disjnt x y*
⟨*proof*⟩

lemma *disjnt-of-nat*:
s ⊆ {*0*..<*2*^*LENGTH*('*a*::*len*)} ⟹ *t* ⊆ {*0*..<*2*^*LENGTH*('*a*)} ⟹
disjnt ((*of-nat* :: *nat* ⇒ '*a word*) ‘ *s*) (*of-nat* ‘ *t*) ⟷ *disjnt s t*
⟨*proof*⟩

lemma *list-all-zip-zip-cons*:
R a b c ⟹
list-all (λ(*a*, *b*, *c*). *R a b c*) (*zip as* (*zip bs cs*)) ⟹
list-all (λ(*a*, *b*, *c*). *R a b c*) (*zip* (*a*#*as*) (*zip* (*b*#*bs*) (*c*#*cs*)))
⟨*proof*⟩

lemma *list-all-zip-zip-empty*:
list-all (λ(*a*, *b*, *c*). *R a b c*) (*zip* [] (*zip* [] []))
⟨*proof*⟩

lemma *list-all-cons*: *P x* ⟹ *list-all P xs* ⟹ *list-all P* (*x* # *xs*)
⟨*proof*⟩

lemma *list-all-nil*: *list-all P* []
⟨*proof*⟩

lemma *fold-functor*:
F id = *id* ⟹ (⋀*a b*. *F* (*a* ∘ *b*) = *F a* ∘ *F b*) ⟹ *F* (*fold a xs*) = *fold* (λ*x*. *F* (*a x*)) *xs*
⟨*proof*⟩

11.16 Legacy definition *fg-cons*

definition *fg-cons* :: ('*a* ⇒ '*b*) ⇒ ('*b* ⇒ '*a* ⇒ '*a*) ⇒ *bool* **where**
fg-cons acc upd ≡
(∀ *bs v*. *acc* (*upd bs v*) = *bs*) ∧ (∀ *bs bs*' *v*. *upd bs* (*upd bs*' *v*) = *upd bs v*) ∧ (∀ *v*. *upd* (*acc v*) *v* = *v*)

11.17 Lense definition using an update function *lense*

type-synonym '*a upd* = '*a* ⇒ '*a*

named-theorems *update-compose*

locale *lense* =
fixes
get :: '*s* ⇒ '*a* **and**
upd :: '*a upd* ⇒ '*s upd*

assumes *get-upd* [*simp*]: *get* (*upd f s*) = *f* (*get s*)
assumes *upd-same* : *f* (*get s*) = *get s* ⟹ *upd f s* = *s*
assumes *upd-compose*[*update-compose*, *simp*]: *upd f* (*upd g s*) = *upd* (*f o g*) *s*
begin

lemma *upd-comp*[*simp*]: *upd f* ∘ *upd g* = *upd* (*f* ∘ *g*)
⟨*proof*⟩
lemma *upd-get* [*simp*]: *upd* (λ-. *get s*) *s* = *s*
⟨*proof*⟩
lemma *upd-id* [*simp*]: *upd* (λ*x*. *x*) *s* = *s*
⟨*proof*⟩
lemma *upd-cong*: *f* (*get s*) = *f'* (*get s*) ⟹ *upd f s* = *upd f'* *s*
⟨*proof*⟩

lemma *upd-comp-fold*: *upd* (*fold f xs*) = *fold* (*upd* ∘ *f*) *xs*
⟨*proof*⟩

lemma *fg-cons*: *fg-cons get* (*upd* ∘ (λ*x* -. *x*))
⟨*proof*⟩

end

lemma *lenseI*:
fixes *get upd*
assumes *1*:(⋀*s f*. *get* (*upd f s*) = *f* (*get s*)) (⋀*s*. *upd* (λ*x*. *get s*) *s* = *s*)
and *2*[*cong*]: (⋀*s f g*. *f* (*get s*) = *g* (*get s*) ⟹ *upd g s* = *upd f s*)
and *3*: (⋀*s f g*. *upd f* (*upd g s*) = *upd* (*f o g*) *s*)
shows *lense get upd*
⟨*proof*⟩

lemma *lenseI-equiv*:
(⋀*x*. *f* (*g x*) = *x*) ⟹ (⋀*x*. *g* (*f x*) = *x*) ⟹ *lense g* (λ*v x*. *f* (*v* (*g x*)))
⟨*proof*⟩

lemma *lense-comp*:
assumes *lense1*: *lense sel1 upd1*
assumes *lense2*: *lense sel2 upd2*
shows *lense* (*sel2 o sel1*) (*upd1 o upd2*)
⟨*proof*⟩

lemma *lense-compose*:
assumes *lense sel1 upd1 lense sel2 upd2*
shows *lense* (λ*x*. *sel2* (*sel1 x*)) (λ*f*. *upd1* (*upd2 f*))
⟨*proof*⟩

lemma *lense-of-fg-cons*:
fg-cons g s ⟹ *lense g* (λ*f x*. *s* (*f* (*g x*)) *x*)
⟨*proof*⟩

lemma *lense-of-fg-cons'*:
fg-cons g (*u* ∘ (λ*x* -. *x*)) ⟹
(⋀*f x*. *u f x* = *u* (λ-. *f* (*g x*)) *x*) ⟹
lense g u
⟨*proof*⟩

11.18 Update for functions

definition *upd-fun* :: *'a* ⇒ *'b upd* ⇒ (*'a* ⇒ *'b*) *upd* **where**
upd-fun i f g = *g*(*i* := *f* (*g i*))

global-interpretation *upd-fun*: *lense* λ*f*. *f a upd-fun a*
⟨*proof*⟩

lemma *upd-fun-ne*: *i* ≠ *j* ⟹ *upd-fun i f g j* = *g j*
⟨*proof*⟩

lemma *upd-fun-commute*: *i* ≠ *j* ⟹ *upd-fun i f* ∘ *upd-fun j g* = *upd-fun j g* ∘ *upd-fun i f*
⟨*proof*⟩

11.19 Scenes

Scenes allow us to represent a projection/lense without referring to a second type.

type-synonym *'a scene* = *'a* ⇒ *'a* ⇒ *'a*

locale *is-scene* =
fixes *s* :: *'a scene*
assumes *left*[*simp*]: *s* (*s a b*) *c* = *s a c*
assumes *right*[*simp*]: *s a* (*s b c*) = *s a c*
assumes *idem*[*simp*]: *s a a* = *a*

lemma *is-scene-all*[*simp*]: *is-scene* (λ*a b*. *a*)
⟨*proof*⟩

lemma *is-scene-id*[*simp*]: *is-scene* (λ*a*. *id*)
⟨*proof*⟩

lemma *is-scene-flip*: *is-scene m* ⟹ *is-scene* (λ*a b*. *m b a*)
⟨*proof*⟩

lemma *is-scene-of-lense*: *lense r w* ⟹ *is-scene* (λ*a*. *w* (λ-. *r a*))
⟨*proof*⟩

lemma *is-scene-of-fg-cons*: *fg-cons r w* ⟹ *is-scene* (λ*a*. *w* (*r a*))
⟨*proof*⟩

lemma *scene-comp-idem*: *is-scene m* $\Longrightarrow$ *m a* $\circ$ *m a* = *m a*
⟨*proof*⟩

definition *comm-scene* :: *'a scene* $\Rightarrow$ *'a scene* $\Rightarrow$ *bool* **where**
comm-scene m1 m2 $\longleftrightarrow$ ($\forall$ *a b*. *m1 a* (*m2 a b*) = *m2 a* (*m1 a b*))

lemma *comm-scene-comm*: *comm-scene m1 m2* $\longleftrightarrow$ *comm-scene m2 m1*
⟨*proof*⟩

lemma *comm-scene-sym*[*intro*]: *comm-scene m1 m2* $\Longrightarrow$ *comm-scene m2 m1*
⟨*proof*⟩

lemma *comm-sceneD*: *comm-scene m1 m2* $\Longrightarrow$ *m1 a* (*m2 a c*) = *m2 a* (*m1 a c*)
⟨*proof*⟩

lemma *comm-scene-all-left*[*simp*]: *is-scene m* $\Longrightarrow$ *comm-scene* (λ*a b*. *a*) *m*
⟨*proof*⟩

lemma *comm-scene-all-right*[*simp*]: *is-scene m* $\Longrightarrow$ *comm-scene m* (λ*a b*. *a*)
⟨*proof*⟩

lemma *comm-scene-id-left*[*simp*]: *comm-scene* (λ*a*. *id*) *m*
⟨*proof*⟩

lemma *comm-scene-id-right*[*simp*]: *comm-scene m* (λ*a*. *id*)
⟨*proof*⟩

lemma *comm-scene-refl*[*intro*, *simp*]: *comm-scene m m*
⟨*proof*⟩

lemma *is-scene-comp*:
is-scene m1 $\Longrightarrow$ *is-scene m2* $\Longrightarrow$ *comm-scene m1 m2* $\Longrightarrow$ *is-scene* (λ*a*. *m1 a* $\circ$ *m2 a*)
⟨*proof*⟩

lemma *comm-scene-comp-left*:
comm-scene m1 m2 $\Longrightarrow$ *comm-scene m1 m* $\Longrightarrow$ *comm-scene m2 m* $\Longrightarrow$ *comm-scene* (λ*a*. *m1 a* $\circ$ *m2 a*) *m*
⟨*proof*⟩

lemma *comm-scene-comp-right*:
comm-scene m m1 $\Longrightarrow$ *comm-scene m m2* $\Longrightarrow$ *comm-scene m1 m2* $\Longrightarrow$ *comm-scene m* (λ*a*. *m1 a* $\circ$ *m2 a*)
⟨*proof*⟩

lemma *comm-scene-fold*:
pairwise comm-scene (*insert m* (*set ms*)) $\Longrightarrow$ *comm-scene* (λ*a*. *fold* (λ*m*. *m a*) *ms*) *m*

⟨*proof*⟩

lemma *is-scene-fold*:
list-all is-scene ms ⟹ *pairwise comm-scene* (*set ms*) ⟹ *is-scene* (λ*a*. *fold* (λ*m*. *m a*) *ms*)
⟨*proof*⟩

lemma *is-scene-fold′*:
list-all (λ*x*. *is-scene* (*m x*)) *ms* ⟹ *pairwise comm-scene* (*m* ‘ *set ms*) ⟹
is-scene (λ*a*. *fold* (λ*x*. *m x a*) *ms*)
⟨*proof*⟩

This expresses "disjointness" on scenes, saying that two scenes occupy disjnt parts of the type.

It is stronger than "commutativity" ∀ *a c*. *m1 a* (*m2 a b*) = *m2 a* (*m1 a b*) which is enough to show composition, but in our cases we are interested in disjointness. Important difference: a scene *m* "commutes" with itself, but only λ*a*. *id* is disjoint with itself.

definition *disjnt-scene* :: *′a scene* ⇒ *′a scene* ⇒ *bool* **where**
disjnt-scene m1 m2 ⟷ (∀ *a b c*. *m1 a* (*m2 b c*) = *m2 b* (*m1 a c*))

lemma *comm-scene-of-disjnt-scene*[*intro*, *simp*]: *disjnt-scene m1 m2* ⟹ *comm-scene m1 m2*
⟨*proof*⟩

lemma *disjnt-scene-comm*: *disjnt-scene m1 m2* ⟷ *disjnt-scene m2 m1*
⟨*proof*⟩

lemma *disjnt-scene-sym*[*intro*]: *disjnt-scene m1 m2* ⟹ *disjnt-scene m2 m1*
⟨*proof*⟩

lemma *disjnt-sceneD*: *disjnt-scene m1 m2* ⟹ *m1 a* (*m2 b c*) = *m2 b* (*m1 a c*)
⟨*proof*⟩

lemma *disjnt-sceneD-left*: *is-scene m1* ⟹ *disjnt-scene m1 m2* ⟹ *m1* (*m2 a b*) *c* = *m1 b c*
⟨*proof*⟩

lemma *disjnt-sceneD-right*: *is-scene m2* ⟹ *disjnt-scene m1 m2* ⟹ *m2* (*m1 a b*) *c* = *m2 b c*
⟨*proof*⟩

lemma *disjnt-scene-all-left*[*simp*]: *disjnt-scene* (λ*a b*. *a*) *m* ⟷ *m* = (λ*a*. *id*)
⟨*proof*⟩

lemma *disjnt-scene-all-right*[*simp*]: *disjnt-scene m* (λ*a b*. *a*) ⟷ *m* = (λ*a*. *id*)
⟨*proof*⟩

lemma *disjnt-scene-id-left*[*simp*]: *disjnt-scene* (λ*a*. *id*) *m*

⟨*proof*⟩

lemma *disjnt-scene-id-right*[*simp*]: *disjnt-scene m* (λ*a*. *id*)
⟨*proof*⟩

lemma *disjnt-scene-comp-left*:
comm-scene m1 m2 ⟹ *disjnt-scene m1 m* ⟹ *disjnt-scene m2 m* ⟹
disjnt-scene (λ*a*. *m1 a* ∘ *m2 a*) *m*
⟨*proof*⟩

lemma *disjnt-scene-comp-right*:
disjnt-scene m m1 ⟹ *disjnt-scene m m2* ⟹ *comm-scene m1 m2* ⟹
disjnt-scene m (λ*a*. *m1 a* ∘ *m2 a*)
⟨*proof*⟩

lemma *disjnt-scene-fold*:
list-all (*disjnt-scene m*) *ms* ⟹ *pairwise comm-scene* (*set ms*) ⟹
disjnt-scene (λ*a*. *fold* (λ*m*. *m a*) *ms*) *m*
⟨*proof*⟩

lemma *fold-disjnt-scene*:
list-all is-scene ms ⟹ *pairwise comm-scene* (*set ms*) ⟹
list-all (λ*m*. *disjnt-scene m m'* ∨ *m* = *m'*) *ms* ⟹ *m'* ∈ *set ms* ⟹
fold (λ*m*. *m* (*m' a b*)) *ms c* = *m' a* (*fold* (λ*m*. *m b*) *ms c*)
⟨*proof*⟩

end

theory *CompoundCTypes*
imports
Vanilla32
Padding
Lens
begin

lemma *simple-type-dest*: ¬*aggregate t* ⟹ ∃ *n sz align align' d*. *t* = *TypDesc align'* (*TypScalar sz align d*) *n*
⟨*proof*⟩

definition *empty-typ-info* :: *nat* ⇒ *typ-name* ⇒ ('*a*,'*b*) *typ-desc* **where**
empty-typ-info algn tn ≡ *TypDesc algn* (*TypAggregate* []) *tn*

primrec
extend-ti :: '*a xtyp-info* ⇒ '*a xtyp-info* ⇒ *nat* ⇒ *field-name* ⇒ '*a field-desc* ⇒ '*a xtyp-info* **and**
extend-ti-struct :: '*a xtyp-info-struct* ⇒ '*a xtyp-info* ⇒ *field-name* ⇒ '*a field-desc* ⇒ '*a xtyp-info-struct*

where
et0: *extend-ti* (*TypDesc algn' st nm*) *t algn fn d* = *TypDesc* (*max algn'* (*max algn* (*align-td t*))) (*extend-ti-struct st t fn d*) *nm*

| *et1*: *extend-ti-struct* (*TypScalar n sz algn*) *t fn d* = *TypAggregate* [*DTuple t fn d*]
| *et2*: *extend-ti-struct* (*TypAggregate ts*) *t fn d* = *TypAggregate* (*ts*@[*DTuple t fn d*])

lemma *aggregate-empty-typ-info* [*simp*]:
aggregate (*empty-typ-info algn tn*)
⟨*proof*⟩

lemma *aggregate-extend-ti* [*simp*]:
aggregate (*extend-ti tag t algn f d*)
⟨*proof*⟩

lemma *typ-name-extend-ti* [*simp*]: *typ-name* (*extend-ti T t algn fn d*) = *typ-name T*
⟨*proof*⟩

definition *update-desc* :: ('*a* ⇒ '*b*) ⇒ ('*b* ⇒ '*a* ⇒ '*a*) ⇒ '*b field-desc* ⇒ '*a field-desc*
where
update-desc acc upd d ≡ (|*field-access* = (*field-access d*) ∘ *acc*,
field-update = λ*bs v*. *upd* (*field-update d bs* (*acc v*)) *v*, *field-sz* = *field-sz d* |)

lemma *update-desc-id*[*simp*]: *update-desc id* (λ*x* -. *x*) = *id*
⟨*proof*⟩

term *map-td* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *t*

definition *adjust-ti* :: '*b xtyp-info* ⇒ ('*a* ⇒ '*b*) ⇒ ('*b* ⇒ '*a* ⇒ '*a*) ⇒ '*a xtyp-info*
where
adjust-ti t acc upd ≡ *map-td* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *t*

lemma *adjust-ti-adjust-ti*:
adjust-ti (*adjust-ti t g s*) *g' s'* =
adjust-ti t (*g* ∘ *g'*) (λ*v x*. *s'* (*s v* (*g' x*)) *x*)
⟨*proof*⟩

lemma *typ-desc-size-update-ti* [*simp*]:
(*size-td* (*adjust-ti t acc upd*) = *size-td t*)
⟨*proof*⟩

lemma *export-tag-adjust-ti*[*rule-format*]:
∀ *bs*. *fg-cons acc upd* ⟶ *wf-fd t* ⟶
export-uinfo (*adjust-ti t acc upd*) = *export-uinfo t*

$\forall$ *bs. fg-cons acc upd* $\longrightarrow$ *wf-fd-struct st* $\longrightarrow$
map-td-struct field-norm (λ-. ()) (*map-td-struct* (λn *algn. update-desc acc upd*) (*update-desc acc upd*) *st*) =
map-td-struct field-norm (λ-. ()) *st*
$\forall$ *bs. fg-cons acc upd* $\longrightarrow$ *wf-fd-list ts* $\longrightarrow$
map-td-list field-norm (λ-. ()) (*map-td-list* (λn *algn. update-desc acc upd*) (*update-desc acc upd*) *ts*) =
map-td-list field-norm (λ-. ()) *ts*
$\forall$ *bs. fg-cons acc upd* $\longrightarrow$ *wf-fd-tuple x* $\longrightarrow$
map-td-tuple field-norm (λ-. ()) (*map-td-tuple* (λn *algn. update-desc acc upd*) (*update-desc acc upd*) *x*) =
map-td-tuple field-norm (λ-. ()) *x*
⟨*proof*⟩

definition (**in** *c-type*) *ti-typ-combine* ::
$'a$ *itself* $\Rightarrow$ ($'b \Rightarrow 'a$) $\Rightarrow$ ($'a \Rightarrow 'b \Rightarrow 'b$) $\Rightarrow$ *nat* $\Rightarrow$ *field-name* $\Rightarrow$ $'b$ *xtyp-info* $\Rightarrow$ $'b$ *xtyp-info*
where
ti-typ-combine t-b acc upd algn fn tag $\equiv$
let ft = *adjust-ti* (*typ-info-t TYPE*($'a$)) *acc upd*
in extend-ti tag ft algn fn (⦇*field-access* = *xto-bytes o acc*, *field-update* = *upd o xfrom-bytes*, *field-sz* = *size-of TYPE*($'a$)⦈)

primrec
padding-fields :: ($'a$,$'b$) *typ-desc* $\Rightarrow$ *field-name list* **and**
padding-fields-struct :: ($'a$,$'b$) *typ-struct* $\Rightarrow$ *field-name list*
where
pf0: *padding-fields* (*TypDesc algn st tn*) = *padding-fields-struct st*

| *pf1*: *padding-fields-struct* (*TypScalar n algn d*) = []
| *pf2*: *padding-fields-struct* (*TypAggregate xs*) = *filter* (λx. *hd x* = *CHR* ''!'') (*map dt-snd xs*)

primrec
non-padding-fields :: ($'a$,$'b$) *typ-desc* $\Rightarrow$ *field-name list* **and**
non-padding-fields-struct :: ($'a$,$'b$) *typ-struct* $\Rightarrow$ *field-name list*
where
npf0: *non-padding-fields* (*TypDesc algn st tn*) = *non-padding-fields-struct st*

| *npf1*: *non-padding-fields-struct* (*TypScalar n algn d*) = []
| *npf2*: *non-padding-fields-struct* (*TypAggregate xs*) = *filter* (λx. *hd x* $\neq$ *CHR* ''!'') (*map dt-snd xs*)

definition *field-names-list* :: ($'a$,$'b$) *typ-desc* $\Rightarrow$ *field-name list* **where**
field-names-list tag $\equiv$ *non-padding-fields tag* @ *padding-fields tag*

definition *ti-pad-combine* :: *nat* ⇒ *'a xtyp-info* ⇒ *'a xtyp-info* **where**
ti-pad-combine n tag ≡
let
fn = *foldl* (@) *''!pad-''* (*field-names-list tag*);
td = *padding-desc n*;
nf = *TypDesc 0* (*TypScalar n 0 td*) *''!pad-typ''*
in extend-ti tag nf 0 fn td

lemma *aggregate-ti-pad-combine* [*simp*]:
aggregate (*ti-pad-combine n tag*)
⟨*proof*⟩

definition (**in** *c-type*) *ti-typ-pad-combine* ::
'a itself ⇒ (*'b* ⇒ *'a*) ⇒ (*'a* ⇒ *'b* ⇒ *'b*) ⇒ *nat* ⇒ *field-name* ⇒ *'b xtyp-info* ⇒ *'b xtyp-info*
where
ti-typ-pad-combine t acc upd algn fn tag ≡
let
pad = *padup* (*max* (*2* ^ *algn*) (*align-of TYPE*(*'a*))) (*size-td tag*);
ntag = *if 0* < *pad then ti-pad-combine pad tag else tag*
in
ti-typ-combine t acc upd algn fn ntag

definition *map-align f t* = (*case t of TypDesc algn st n* ⇒ *TypDesc* (*f algn*) *st n*)
lemma *map-align-simp* [*simp*]: *map-align f* (*TypDesc algn st n*) = *TypDesc* (*f algn*) *st n*
⟨*proof*⟩

definition *final-pad* :: *nat* ⇒ *'a xtyp-info* ⇒ *'a xtyp-info* **where**
final-pad algn tag ≡
let n = *padup* (*2*^(*max algn* (*align-td tag*))) (*size-td tag*)
in map-align (*max algn*) (*if 0* < *n then ti-pad-combine n tag else tag*)

lemma *field-names-list-empty-typ-info* [*simp*]:
set (*field-names-list* (*empty-typ-info algn tn*)) = {}
⟨*proof*⟩

lemma *field-names-list-extend-ti* [*simp*]:
set (*field-names-list* (*extend-ti tag t algn fn d*)) = *set* (*field-names-list tag*) ∪ {*fn*}
⟨*proof*⟩

lemma (**in** *c-type*) *field-names-list-ti-typ-combine* [*simp*]:
set (*field-names-list* (*ti-typ-combine* (*t*::*'a itself*) *f f-upd algn fn tag*))
= *set* (*field-names-list tag*) ∪ {*fn*}
⟨*proof*⟩

lemma *field-names-list-ti-pad-combine* [*simp*]:

set (*field-names-list* (*ti-pad-combine n tag*))
= *set* (*field-names-list tag*) ∪ {*foldl* (@) ''!pad-'' (*field-names-list tag*)}
⟨*proof*⟩

lemma *hd-string-hd-fold-eq* [*simp*]:
⟦ *s* ≠ []; *hd s* = *CHR* ''!'' ⟧ ⟹ *hd* (*foldl* (@) *s xs*) = *CHR* ''!''
⟨*proof*⟩

lemma *field-names-list-ti-typ-pad-combine* [*simp*]:
hd x ≠ *CHR* ''!'' ⟹
x ∈ *set* (*field-names-list* (*ti-typ-pad-combine align t-b f-ab f-upd-ab fn tag*))
= (*x* ∈ *set* (*field-names-list tag*) ∪ {*fn*})
⟨*proof*⟩

lemma *wf-desc-empty-typ-info* [*simp*]:
wf-desc (*empty-typ-info algn tn*)
⟨*proof*⟩

lemma *wf-desc-extend-ti*:
⟦ *wf-desc tag*; *wf-desc t*; *f* ∉ *set* (*field-names-list tag*) ⟧ ⟹
wf-desc (*extend-ti tag t algn f d*)
⟨*proof*⟩

lemma *foldl-append-length*:
length (*foldl* (@) *s xs*) ≥ *length s*
⟨*proof*⟩

lemma *foldl-append-nmem*:
s ≠ [] ⟹ *foldl* (@) *s xs* ∉ *set xs*
⟨*proof*⟩

lemma *wf-desc-ti-pad-combine*:
wf-desc tag ⟹ *wf-desc* (*ti-pad-combine n tag*)
⟨*proof*⟩

lemma *wf-desc-adjust-ti* [*simp*]:
wf-desc (*adjust-ti t f g*) = *wf-desc* (*t*::'*a xtyp-info*)
⟨*proof*⟩

lemma (**in** *wf-type*) *wf-desc-ti-typ-combine*:
⟦ *wf-desc tag*; *fn* ∉ *set* (*field-names-list tag*) ⟧ ⟹
wf-desc (*ti-typ-combine* (*t-b*::'*a itself*) *acc upd-ab algn fn tag*)
⟨*proof*⟩

lemma (**in** *wf-type*) *wf-desc-ti-typ-pad-combine*:
⟦ *wf-desc tag*; *fn* ∉ *set* (*field-names-list tag*); *hd fn* ≠ *CHR* ''!'' ⟧ ⟹
wf-desc (*ti-typ-pad-combine* (*t-b*::'*a itself*) *acc upd algn fn tag*)
⟨*proof*⟩

lemma *wf-desc-map-align*: *wf-desc tag* ⟹ *wf-desc* (*map-align f tag*)

⟨*proof*⟩

lemma *wf-desc-final-pad*:
wf-desc tag ⟹ *wf-desc* (*final-pad algn tag*)
⟨*proof*⟩

lemma *wf-size-desc-extend-ti*:
⟦ *wf-size-desc tag*; *wf-size-desc t* ⟧ ⟹ *wf-size-desc* (*extend-ti tag t algn fn d*)
⟨*proof*⟩

lemma *wf-size-desc-ti-pad-combine*:
⟦ *wf-size-desc tag*; *0* < *n* ⟧ ⟹ *wf-size-desc* (*ti-pad-combine n tag*)
⟨*proof*⟩

lemma *wf-size-desc-adjust-ti*:
wf-size-desc (*adjust-ti t f g*) = *wf-size-desc* (*t*::*'a xtyp-info*)
⟨*proof*⟩

lemma (**in** *wf-type*) *wf-size-desc-ti-typ-combine*:
wf-size-desc tag ⟹ *wf-size-desc* (*ti-typ-combine* (*t-b*::*'a itself*) *acc upd-ab algn fn tag*)
⟨*proof*⟩

lemma (**in** *wf-type*) *wf-size-desc-ti-typ-pad-combine*:
wf-size-desc tag ⟹
wf-size-desc (*ti-typ-pad-combine* (*t-b*::*'a itself*) *acc upd algn fn tag*)
⟨*proof*⟩

lemma (**in** *wf-type*) *wf-size-desc-ti-typ-combine-empty* [*simp*]:
wf-size-desc (*ti-typ-combine* (*t-b*::*'a itself*) *acc upd algn fn* (*empty-typ-info algn' tn*))
⟨*proof*⟩

lemma (**in** *wf-type*) *wf-size-desc-ti-typ-pad-combine-empty* [*simp*]:
wf-size-desc (*ti-typ-pad-combine* (*t-b*::*'a itself*) *acc upd algn fn*
(*empty-typ-info algn' tn*))
⟨*proof*⟩

lemma *wf-size-desc-msp-align*:
wf-size-desc tag ⟹ *wf-size-desc* (*map-align f tag*)
⟨*proof*⟩

lemma *wf-size-desc-final-pad*:
wf-size-desc tag ⟹ *wf-size-desc* (*final-pad algn tag*)
⟨*proof*⟩

lemma *wf-fdp-set-comp-simp* [*simp*]:
wf-fdp {(*a*, *n* # *b*) |*a b*. (*a*, *b*) ∈ *tf-set t*} = *wf-fdp* (*tf-set t*)
⟨*proof*⟩

lemma *lf-set-adjust-ti'*:
$\forall\, d\ fn.\ d \in$ *lf-set* (*map-td* ($\lambda n\ algn.$ *update-desc acc upd*) (*update-desc acc upd*) *t*) *fn* $\longrightarrow$
(∃ *y*. *lf-fd d*=*update-desc acc upd* (*lf-fd y*) ∧ *lf-sz d*=*lf-sz y* ∧ *lf-fn d*=*lf-fn y* ∧ *y* ∈ *lf-set t fn*)
$\forall\, d\ fn.\ d \in$ *lf-set-struct* (*map-td-struct* ($\lambda n\ algn.$ *update-desc acc upd*) (*update-desc acc upd*) *st*) *fn* $\longrightarrow$
(∃ *y*. *lf-fd d*=*update-desc acc upd* (*lf-fd y*) ∧ *lf-sz d*=*lf-sz y* ∧ *lf-fn d*=*lf-fn y* ∧ *y* ∈ *lf-set-struct st fn*)
$\forall\, d\ fn.\ d \in$ *lf-set-list* (*map-td-list* ($\lambda n\ algn.$ *update-desc acc upd*) (*update-desc acc upd*) *ts*) *fn* $\longrightarrow$
(∃ *y*. *lf-fd d*=*update-desc acc upd* (*lf-fd y*) ∧ *lf-sz d*=*lf-sz y* ∧ *lf-fn d*=*lf-fn y* ∧ *y* ∈ *lf-set-list ts fn*)
$\forall\, d\ fn.\ d \in$ *lf-set-tuple* (*map-td-tuple* ($\lambda n\ algn.$ *update-desc acc upd*) (*update-desc acc upd*) *x*) *fn* $\longrightarrow$
(∃ *y*. *lf-fd d*=*update-desc acc upd* (*lf-fd y*) ∧ *lf-sz d*=*lf-sz y* ∧ *lf-fn d*=*lf-fn y* ∧ *y* ∈ *lf-set-tuple x fn*)
⟨*proof*⟩

lemma *lf-set-adjust-ti*:
⟦ *d* ∈ *lf-set* (*adjust-ti t acc upd*) *fn*; $\bigwedge$*y*. *upd* (*acc y*) *y* = *y* ⟧ ⟹
(∃ *y*. *lf-fd d*=*update-desc acc upd* (*lf-fd y*) ∧ *lf-sz d*=*lf-sz y* ∧ *lf-fn d*=*lf-fn y* ∧ *y* ∈ *lf-set t fn*)
⟨*proof*⟩

lemma *fd-cons-struct-id-simp* [*simp*]:
fd-cons-struct (*TypScalar n algn* (|*field-access* = $\lambda v.$ *id*, *field-update* = $\lambda bs.$ *id*, *field-sz* = *m*|))
⟨*proof*⟩

lemma *field-desc-adjust-ti*:
fg-cons acc upd $\longrightarrow$
field-desc (*adjust-ti* (*t*::'*a xtyp-info*) *acc upd*) =
update-desc acc upd (*field-desc t*)
fg-cons acc upd $\longrightarrow$
field-desc-struct (*map-td-struct* ($\lambda n\ algn.$ *update-desc acc upd*) (*update-desc acc upd*) *st*) =
update-desc acc upd (*field-desc-struct st*)
fg-cons acc upd $\longrightarrow$
field-desc-list (*map-td-list* ($\lambda n\ algn.$ *update-desc acc upd*) (*update-desc acc upd*) *ts*) =
update-desc acc upd (*field-desc-list ts*)
fg-cons acc upd $\longrightarrow$
field-desc-tuple (*map-td-tuple* ($\lambda n\ algn.$ *update-desc acc upd*) (*update-desc acc upd*) *x*) =
update-desc acc upd (*field-desc-tuple x*)
⟨*proof*⟩

lemma *update-ti-t-adjust-ti*:
fg-cons acc upd $\Longrightarrow$ *update-ti-t* (*adjust-ti t acc upd*) *bs v* = *upd* (*update-ti-t t bs* (*acc v*)) *v*
⟨*proof*⟩

declare *field-desc-def* [*simp del*]

lemma (**in** *c-type*) *aggregate-ti-typ-combine* [*simp*]:
aggregate (*ti-typ-combine* (*t-b*::'*a itself*) *acc upd algn fn tag*)
⟨*proof*⟩

lemma (**in** *c-type*) *aggregate-ti-typ-pad-combine* [*simp*]:
aggregate (*ti-typ-pad-combine* (*t-b*::'*a itself*) *acc upd algn fn tag*)
⟨*proof*⟩

lemma *align-of-empty-typ-info* [*simp*]:
align-td-wo-align (*empty-typ-info algn tn*) = *0*
⟨*proof*⟩

lemma *align-of-empty-typ-info'* [*simp*]:
align-td (*empty-typ-info algn tn*) = *algn*
⟨*proof*⟩

lemma *align-of-tag-list* [*simp*]:
align-td-wo-align-list (*xs* @ [*DTuple t fn d*]) = *max* (*align-td-wo-align-list xs*) (*align-td-wo-align t*)
⟨*proof*⟩

lemma *align-of-tag-list'* [*simp*]:
align-td-list (*xs* @ [*DTuple t fn d*]) = *max* (*align-td-list xs*) (*align-td t*)
⟨*proof*⟩

lemma *align-of-extend-ti* [*simp*]:
aggregate ti $\Longrightarrow$ *align-td-wo-align* (*extend-ti ti t algn fn d*) = *max* (*align-td-wo-align ti*) (*align-td-wo-align t*)
⟨*proof*⟩

lemma *align-of-extend-ti'* [*simp*]:
aggregate ti $\Longrightarrow$ *align-td* (*extend-ti ti t algn fn d*) = *max* (*align-td ti*) (*max algn* (*align-td t*))
⟨*proof*⟩

lemma *align-of-adjust-ti* [*simp*]:
align-td-wo-align (*adjust-ti t acc upd*) = *align-td-wo-align* (*t*::'*a xtyp-info*)
⟨*proof*⟩

lemma *align-of-adjust-ti'* [*simp*]:
align-td (*adjust-ti t acc upd*) = *align-td* (*t*::'*a xtyp-info*)

⟨*proof*⟩

lemma (**in** *c-type*) *align-of-ti-typ-combine* [*simp*]:
aggregate ti ⟹
align-td-wo-align (*ti-typ-combine* (*t*::′*a itself*) *acc upd algn fn ti*) =
max (*align-td-wo-align ti*) (*align-td-wo-align* (*typ-info-t* (*TYPE*(′*a*))))
⟨*proof*⟩

lemma (**in** *c-type*) *align-of-ti-typ-combine*′ [*simp*]:
aggregate ti ⟹
align-td (*ti-typ-combine* (*t*::′*a itself*) *acc upd algn fn ti*) =
max (*align-td ti*) (*max algn* (*align-td* (*typ-info-t TYPE*(′*a*))))
⟨*proof*⟩

lemma *align-of-ti-pad-combine* [*simp*]:
aggregate ti ⟹ *align-td-wo-align* (*ti-pad-combine n ti*) = (*align-td-wo-align ti*)
⟨*proof*⟩

lemma *align-of-ti-pad-combine*′ [*simp*]:
aggregate ti ⟹ *align-td* (*ti-pad-combine n ti*) = (*align-td ti*)
⟨*proof*⟩

lemma *max-2-exp*: *max* ((*2*::*nat*) ^ *a*) (*2* ^ *b*) = *2* ^ (*max a b*)
⟨*proof*⟩

lemma *align-td-wo-align-map-align*: *align-td-wo-align* (*map-align f t*) = *align-td-wo-align t*
⟨*proof*⟩

lemma *align-td-wo-align-final-pad*:
aggregate ti ⟹
align-td-wo-align (*final-pad algn ti*) = (*align-td-wo-align ti*)
⟨*proof*⟩

lemma *align-td-map-align* [*simp*]: *align-td* (*map-align f t*) = *f* (*align-td t*)
⟨*proof*⟩

lemma *align-of-final-pad*:
aggregate ti ⟹
align-td (*final-pad algn ti*) = *max algn* (*align-td ti*)
⟨*proof*⟩

lemma (**in** *c-type*) *align-td-wo-align-ti-typ-pad-combine* [*simp*]:
aggregate ti ⟹
align-td-wo-align (*ti-typ-pad-combine* (*t*::′*a itself*) *acc upd algn fn ti*) =
max (*align-td-wo-align ti*) (*align-td-wo-align* (*typ-info-t TYPE*(′*a*)))
⟨*proof*⟩

lemma (**in** *c-type*) *align-td-ti-typ-pad-combine* [*simp*]:
aggregate ti $\Longrightarrow$
align-td (*ti-typ-pad-combine* (*t*::'*a itself*) *acc upd algn fn ti*) =
max (*align-td ti*) (*max algn* (*align-td* (*typ-info-t TYPE*('*a*))))
⟨*proof*⟩

definition *fu-s-comm-set* :: (*byte list* ⇒ '*a* ⇒ '*a*) *set* ⇒ (*byte list* ⇒ '*a* ⇒ '*a*) *set* ⇒ *bool*
where
fu-s-comm-set X Y ≡ ∀ *x y*. *x* ∈ *X* ∧ *y* ∈ *Y* ⟶ (∀ *v bs bs'*. *x bs* (*y bs' v*) = *y bs'* (*x bs v*))

lemma *fc-empty-ti* [*simp*]:
fu-commutes (*update-ti-t* (*empty-typ-info algn tn*)) *f*
⟨*proof*⟩

lemma *fc-extend-ti*:
⟦ *fu-commutes* (*update-ti-t s*) *h*; *fu-commutes* (*update-ti-t t*) *h* ⟧
$\Longrightarrow$ *fu-commutes* (*update-ti-t* (*extend-ti s t algn fn d*)) *h*
⟨*proof*⟩

lemma *fc-update-ti*:
⟦ *fu-commutes* (*update-ti-t ti*) *h*; *fg-cons acc upd*;
∀ *v bs bs'*. *upd bs* (*h bs' v*) = *h bs'* (*upd bs v*); ∀ *bs v*. *acc* (*h bs v*) = *acc v* ⟧
$\Longrightarrow$ *fu-commutes* (*update-ti-t* (*adjust-ti t acc upd*)) *h*
⟨*proof*⟩

lemma (**in** *c-type*) *fc-ti-typ-combine*:
⟦ *fu-commutes* (*update-ti-t ti*) *h*; *fg-cons acc upd*;
∀ *v bs bs'*. *upd bs* (*h bs' v*) = *h bs'* (*upd bs v*); ∀ *bs v*. *acc* (*h bs v*) = *acc v* ⟧
$\Longrightarrow$ *fu-commutes* (*update-ti-t* (*ti-typ-combine* (*t*::'*a itself*) *acc upd algn fn ti*)) *h*
⟨*proof*⟩

lemma *fc-ti-pad-combine*:
fu-commutes (*update-ti-t ti*) *f* $\Longrightarrow$ *fu-commutes* (*update-ti-t* (*ti-pad-combine n ti*)) *f*
⟨*proof*⟩

lemma (**in** *c-type*) *fc-ti-typ-pad-combine*:
⟦ *fu-commutes* (*update-ti-t ti*) *h*; *fg-cons acc upd*;
∀ *v bs bs'*. *upd bs* (*h bs' v*) = *h bs'* (*upd bs v*); ∀ *bs v*. *acc* (*h bs v*) = *acc v* ⟧
$\Longrightarrow$ *fu-commutes* (*update-ti-t* (*ti-typ-pad-combine* (*t*::'*a itself*) *acc upd algn fn ti*)) *h*
⟨*proof*⟩

definition *fu-eq-mask* :: '*a xtyp-info* ⇒ ('*a* ⇒ '*a*) ⇒ *bool* **where**
fu-eq-mask ti f ≡
∀ *bs v v'*. *length bs* = *size-td ti* ⟶ *update-ti-t ti bs* (*f v*) = *update-ti-t ti bs* (*f v'*)

lemma *fu-eq-mask*:
⟦ *length bs* = *size-td ti*; *fu-eq-mask ti id* ⟧ ⟹
update-ti-t ti bs v = *update-ti-t ti bs w*
⟨*proof*⟩

lemma *fu-eq-mask-ti-pad-combine*:
⟦ *fu-eq-mask ti f*; *aggregate ti* ⟧ ⟹ *fu-eq-mask* (*ti-pad-combine n ti*) *f*
⟨*proof*⟩

lemma *fu-eq-mask-map-align*: *fu-eq-mask t f* ⟹ *fu-eq-mask* (*map-align g t*) *f*
⟨*proof*⟩

lemma *fu-eq-mask-final-pad*:
⟦ *fu-eq-mask ti f*; *aggregate ti* ⟧ ⟹ *fu-eq-mask* (*final-pad algn ti*) *f*
⟨*proof*⟩

definition *upd-local* :: (′*b* ⇒ ′*a* ⇒ ′*a*) ⇒ *bool* **where**
upd-local g ≡ ∀ *j k v v*′. *g k v* = *g k v*′ ⟶ *g j v* = *g j v*′

lemma *fg-cons-upd-local*:
fg-cons f g ⟹ *upd-local g*
⟨*proof*⟩

lemma (**in** *mem-type*) *fu-eq-mask-ti-typ-combine*:
⟦ *fu-eq-mask ti* (λ*v*. (*upd* (*acc undefined*) (*h v*))); *fg-cons acc upd*;
fu-commutes (*update-ti-t ti*) *upd*; *aggregate ti* ⟧ ⟹
fu-eq-mask (*ti-typ-combine* (*t*::′*a itself*) *acc upd algn fn ti*) *h*
⟨*proof*⟩

lemma (**in** *mem-type*) *fu-eq-mask-ti-typ-pad-combine*:
⟦ *fu-eq-mask ti* (λ*v*. (*upd* (*acc undefined*) (*h v*))); *fg-cons acc upd*;
fu-commutes (*update-ti-t ti*) *upd*; *aggregate ti* ⟧ ⟹
fu-eq-mask (*ti-typ-pad-combine* (*t*::′*a itself*) *acc upd algn fn ti*) *h*
⟨*proof*⟩

lemma *fu-eq-mask-empty-typ-info-g*:
∃ *k*. ∀ *v*. *f v* = *k* ⟹ *fu-eq-mask t f*
⟨*proof*⟩

lemma *fu-eq-mask-empty-typ-info*:
∀ *v*. *f v* = *undefined* ⟹ *fu-eq-mask t f*
⟨*proof*⟩

lemma *size-td-extend-ti*:
aggregate s ⟹ *size-td* (*extend-ti s t algn fn d*) = *size-td s* + *size-td t*
⟨*proof*⟩

lemma *size-td-ti-pad-combine*:

aggregate ti $\Longrightarrow$ *size-td* (*ti-pad-combine n ti*) = *n* + *size-td ti*
⟨*proof*⟩

lemma *size-td-map-align* [*simp*]: *size-td* (*map-align f ti*) = *size-td ti*
⟨*proof*⟩

lemma *align-of-dvd-size-of-final-pad* [*simp*]:
aggregate ti $\Longrightarrow$ *2^align-td* (*final-pad algn ti*) *dvd size-td* (*final-pad algn ti*)
⟨*proof*⟩

lemma *size-td-lt-ti-pad-combine*:
aggregate t $\Longrightarrow$ *size-td* (*ti-pad-combine n t*) = *size-td t* + *n*
⟨*proof*⟩

lemma (**in** *c-type*) *size-td-lt-ti-typ-combine*:
aggregate ti $\Longrightarrow$
size-td (*ti-typ-combine* (*t*::*'a itself*) *f g algn fn ti*) =
size-td ti + *size-td* (*typ-info-t TYPE*(*'a*))
⟨*proof*⟩

lemma (**in** *c-type*) *size-td-lt-ti-typ-pad-combine*:
aggregate ti $\Longrightarrow$
size-td (*ti-typ-pad-combine* (*t*::*'a itself*) *f g algn fn ti*) =
(*let k* = *size-td ti in*
k + *size-td* (*typ-info-t TYPE*(*'a*)) + *padup* (*2^*(*max algn* (*align-td* (*typ-info-t TYPE*(*'a*))))) *k*)
⟨*proof*⟩

lemma *size-td-lt-final-pad*:
aggregate tag $\Longrightarrow$
size-td (*final-pad align tag*) = (*let k*=*size-td tag in k* + *padup* (*2^*(*max align* (*align-td tag*))) *k*)
⟨*proof*⟩

lemma *size-td-empty-typ-info* [*simp*]:
size-td (*empty-typ-info algn tn*) = *0*
⟨*proof*⟩

lemma *wf-lf-empty-typ-info* [*simp*]:
wf-lf {}
⟨*proof*⟩

lemma *lf-fn-disj-fn*:
fn $\notin$ *set* (*field-names-list* (*TypDesc algn* (*TypAggregate xs*) *tn*)) $\Longrightarrow$
lf-fn ‘ *lf-set-list xs* [] ∩ *lf-fn* ‘ *lf-set t* [*fn*] = {}
⟨*proof*⟩

lemma *wf-lf-extend-ti*:

⟦ *wf-lf* (*lf-set t* []); *wf-lf* (*lf-set ti* []); *wf-desc t*; *fn* ∉ *set* (*field-names-list ti*);
ti-ind (*lf-set ti* []) (*lf-set t* []) ⟧ ⟹
wf-lf (*lf-set* (*extend-ti ti t algn fn d*) [])
⟨*proof*⟩

lemma *wf-lf-ti-pad-combine*:
wf-lf (*lf-set ti* []) ⟹ *wf-lf* (*lf-set* (*ti-pad-combine n ti*) [])
⟨*proof*⟩

lemma *lf-set-map-align* [*simp*]: *lf-set* (*map-align f ti*) = *lf-set ti*
⟨*proof*⟩

lemma *wf-lf-final-pad*:
wf-lf (*lf-set ti* []) ⟹ *wf-lf* (*lf-set* (*final-pad algn ti*) [])
⟨*proof*⟩

lemma *wf-lf-adjust-ti*:
⟦ *wf-lf* (*lf-set t* []); ⋀*v*. *g* (*f v*) *v* = *v*;
⋀*bs bs*′ *v*. *g bs* (*g bs*′ *v*) = *g bs v*; ⋀*bs v*. *f* (*g bs v*) = *bs* ⟧
⟹ *wf-lf* (*lf-set* (*adjust-ti t f g*) [])
⟨*proof*⟩

lemma *ti-ind-empty-typ-info* [*simp*]:
ti-ind (*lf-set* (*empty-typ-info algn tn*) []) (*lf-set* (*adjust-ti k f g*) [])
⟨*proof*⟩

lemma *ti-ind-extend-ti*:
⟦ *ti-ind* (*lf-set t* []) (*lf-set* (*adjust-ti k acc upd*) []);
ti-ind (*lf-set ti* []) (*lf-set* (*adjust-ti k acc upd*) []) ⟧
⟹ *ti-ind* (*lf-set* (*extend-ti ti t algn n d*) []) (*lf-set* (*adjust-ti k acc upd*) [])
⟨*proof*⟩

lemma *ti-ind-ti-pad-combine*:
ti-ind (*lf-set ti* []) (*lf-set* (*adjust-ti k acc upd*) []) ⟹
ti-ind (*lf-set* (*ti-pad-combine n ti*) []) (*lf-set* (*adjust-ti k acc upd*) [])
⟨*proof*⟩

definition *acc-ind* :: (′*a* ⇒ ′*b*) ⇒ ′*a field-desc set* ⇒ *bool* **where**
acc-ind acc X ≡ ∀ *x bs v*. *x* ∈ *X* ⟶ *acc* (*field-update x bs v*) = *acc v*

definition *fu-s-comm-k* :: ′*a leaf-desc set* ⇒ (′*b* ⇒ ′*a* ⇒ ′*a*) ⇒ *bool* **where**
fu-s-comm-k X upd ≡ ∀ *x*. *x* ∈ *field-update* ‘ *lf-fd* ‘ *X* ⟶ *fu-commutes x upd*

definition *upd-ind* :: ′*a leaf-desc set* ⇒ (′*b* ⇒ ′*a* ⇒ ′*a*) ⇒ *bool* **where**
upd-ind X upd ≡ *fu-s-comm-k X upd*

definition *fa-ind* :: ′*a field-desc set* ⇒ (′*b* ⇒ ′*a* ⇒ ′*a*) ⇒ *bool* **where**
fa-ind X upd ≡ ∀ *x bs v*. *x* ∈ *X* ⟶ *field-access x* (*upd bs v*) = *field-access x v*

lemma *lf-fd-fn*:
∀*fn*. *lf-fd* ‘ (*lf-set* (*t*::′*a xtyp-info*) *fn*) = *lf-fd* ‘ (*lf-set t* [])
∀*fn*. *lf-fd* ‘ (*lf-set-struct* (*st*::′*a xtyp-info-struct*) *fn*) = *lf-fd* ‘ (*lf-set-struct st* [])
∀*fn*. *lf-fd* ‘ (*lf-set-list* (*ts*::′*a xtyp-info-tuple list*) *fn*) = *lf-fd* ‘ (*lf-set-list ts* [])
∀*fn*. *lf-fd* ‘ (*lf-set-tuple* (*x*::′*a xtyp-info-tuple*) *fn*) = *lf-fd* ‘ (*lf-set-tuple x* [])
⟨*proof*⟩

lemma *lf-set-empty-typ-info* [*simp*]:
lf-set (*empty-typ-info algn tn*) *fn* = {}
⟨*proof*⟩

lemma *upd-ind-empty* [*simp*]:
upd-ind {} *upd*
⟨*proof*⟩

lemma *upd-ind-extend-ti*:
⟦ *upd-ind* (*lf-set s* []) *upd*; *upd-ind* (*lf-set t* []) *upd* ⟧ ⟹
upd-ind (*lf-set* (*extend-ti s t algn fn d*) []) *upd*
⟨*proof*⟩

lemma (**in** *c-type*) *upd-ind-ti-typ-combine*:
⟦ *upd-ind* (*lf-set ti* []) *h*; ⋀*w u v*. *upd w* (*h u v*) = *h u* (*upd w v*);
⋀*w v*. *acc* (*h w v*) = *acc v*; ⋀*v*. *upd* (*acc v*) *v* = *v* ⟧
⟹ *upd-ind* (*lf-set* (*ti-typ-combine* (*t*::′*a itself*) *acc upd algn fn ti*) []) *h*
⟨*proof*⟩

lemma *upd-ind-ti-pad-combine*:
upd-ind ((*lf-set ti* [])) *upd* ⟹ *upd-ind* ((*lf-set* (*ti-pad-combine n ti*) [])) *upd*
⟨*proof*⟩

lemma (**in** *c-type*) *upd-ind-ti-typ-pad-combine*:
⟦ *upd-ind* (*lf-set ti* []) *h*; ⋀*w u v*. *upd w* (*h u v*) = *h u* (*upd w v*);
⋀*w v*. *acc* (*h w v*) = *acc v*; ⋀*v*. *upd* (*acc v*) *v* = *v* ⟧
⟹ *upd-ind* (*lf-set* (*ti-typ-pad-combine* (*t*::′*a itself*) *acc upd algn fn ti*) []) *h*
⟨*proof*⟩

lemma *acc-ind-empty* [*simp*]:
acc-ind acc {}
⟨*proof*⟩

lemma *acc-ind-extend-ti*:
⟦ *acc-ind acc* (*lf-fd* ‘ *lf-set s* []); *acc-ind acc* (*lf-fd* ‘ *lf-set t* []) ⟧ ⟹
acc-ind acc (*lf-fd* ‘ *lf-set* (*extend-ti s t algn fn d*) [])
⟨*proof*⟩

lemma (**in** *c-type*) *acc-ind-ti-typ-combine*:
⟦ *acc-ind h* (*lf-fd* ‘ *lf-set ti* []); ⋀*v w*. *h* (*upd w v*) = *h v*;
⋀*v*. *upd* (*acc v*) *v* = *v* ⟧

$\Longrightarrow$ *acc-ind h* (*lf-fd* ‘ *lf-set* (*ti-typ-combine* (*t*::′*a itself*) *acc upd algn fn ti*) [])
⟨*proof*⟩

lemma *acc-ind-ti-pad-combine*:
acc-ind acc (*lf-fd* ‘ (*lf-set t* [])) $\Longrightarrow$ *acc-ind acc* (*lf-fd* ‘ (*lf-set* (*ti-pad-combine n t*) []))
⟨*proof*⟩

lemma (**in** *c-type*) *acc-ind-ti-typ-pad-combine*:
⟦ *acc-ind h* (*lf-fd* ‘ *lf-set ti* []); $\bigwedge$*v w*. *h* (*upd w v*) = *h v*; $\bigwedge$*v*. *upd* (*acc v*) *v* = *v* ⟧
$\Longrightarrow$ *acc-ind h* (*lf-fd* ‘ *lf-set* (*ti-typ-pad-combine* (*t*::′*a itself*) *acc upd algn fn ti*) [])
⟨*proof*⟩

lemma *fa-ind-empty* [*simp*]:
fa-ind {} *upd*
⟨*proof*⟩

lemma *fa-ind-extend-ti*:
⟦ *fa-ind* (*lf-fd* ‘ *lf-set s* []) *upd*; *fa-ind* (*lf-fd* ‘ *lf-set t* []) *upd* ⟧ $\Longrightarrow$
fa-ind (*lf-fd* ‘ *lf-set* (*extend-ti s t algn fn d*) []) *upd*
⟨*proof*⟩

lemma (**in** *c-type*) *fa-ind-ti-typ-combine*:
⟦ *fa-ind* (*lf-fd* ‘ *lf-set ti* []) *h*; $\bigwedge$*v w*. *acc* (*h w v*) = *acc v*;
$\bigwedge$*v*. *upd* (*acc v*) *v* = *v* ⟧
$\Longrightarrow$ *fa-ind* (*lf-fd* ‘ *lf-set* (*ti-typ-combine* (*t*::′*a itself*) *acc upd algn fn ti*) []) *h*
⟨*proof*⟩

lemma *fa-ind-ti-pad-combine*:
fa-ind (*lf-fd* ‘ (*lf-set ti* [])) *upd* $\Longrightarrow$ *fa-ind* (*lf-fd* ‘ (*lf-set* (*ti-pad-combine n ti*) [])) *upd*
⟨*proof*⟩

lemma (**in** *c-type*) *fa-ind-ti-typ-pad-combine*:
⟦ *fa-ind* (*lf-fd* ‘ *lf-set ti* []) *h*; $\bigwedge$*v w*. *f* (*h w v*) = *f v*;
$\bigwedge$*v*. *g* (*f v*) *v* = *v* ⟧
$\Longrightarrow$ *fa-ind* (*lf-fd* ‘ *lf-set* (*ti-typ-pad-combine* (*t*::′*a itself*) *f g algn fn ti*) []) *h*
⟨*proof*⟩

lemma (**in** *wf-type*) *wf-lf-ti-typ-combine*:
⟦ *wf-lf* (*lf-set ti* []); *fn* ∉ *set* (*field-names-list ti*);
$\bigwedge$*v*. *upd* (*acc v*) *v* = *v*; $\bigwedge$*w u v*. *upd w* (*upd u v*) = *upd w v*;
$\bigwedge$*w v*. *acc* (*upd w v*) = *w*;
upd-ind (*lf-set ti* []) *upd*; *acc-ind acc* (*lf-fd* ‘ *lf-set ti* []);
fa-ind (*lf-fd* ‘ *lf-set ti* []) *upd* ⟧ $\Longrightarrow$
wf-lf (*lf-set* (*ti-typ-combine* (*t*::′*a itself*) *acc upd algn fn ti*) [])
⟨*proof*⟩

lemma (**in** *wf-type*) *wf-lf-ti-typ-pad-combine*:
⟦ *wf-lf* (*lf-set ti* []); *fn* ∉ *set* (*field-names-list ti*); *hd fn* ≠ *CHR* ''!'';
⋀*v*. *upd* (*acc v*) *v* = *v*; ⋀*w u v*. *upd w* (*upd u v*) = *upd w v*;
⋀*w v*. *acc* (*upd w v*) = *w*;
upd-ind (*lf-set ti* []) *upd*; *acc-ind acc* (*lf-fd* ‘ *lf-set ti* []);
fa-ind (*lf-fd* ‘ *lf-set ti* []) *upd* ⟧ ⟹
wf-lf (*lf-set* (*ti-typ-pad-combine* (*t*::''*a itself*) *acc upd algn fn ti*) [])
⟨*proof*⟩

definition *upd-fa-ind X upd* ≡ *upd-ind X upd* ∧ *fa-ind* (*lf-fd* ‘ *X*) *upd*

lemma (**in** *wf-type*) *wf-lf-ti-typ-pad-combine'*:
⟦ *wf-lf* (*lf-set ti* []); *fn* ∉ *set* (*field-names-list ti*); *hd fn* ≠ *CHR* ''!'';
⋀*v*. *upd* (*acc v*) *v* = *v*; ⋀*w u v*. *upd w* (*upd u v*) = *upd w v*;
⋀*w v*. *acc* (*upd w v*) = *w*;
upd-fa-ind (*lf-set ti* []) *upd*; *acc-ind acc* (*lf-fd* ‘ *lf-set ti* [])⟧
⟹
wf-lf (*lf-set* (*ti-typ-pad-combine* (*t*::''*a itself*) *acc upd algn fn ti*) [])
⟨*proof*⟩

lemma (**in** *c-type*) *upd-fa-ind-ti-typ-pad-combine*:
⟦ *upd-fa-ind* (*lf-set ti* []) *h*; ⋀*w u v*. *upd w* (*h u v*) = *h u* (*upd w v*);
⋀*w v*. *acc* (*h w v*) = *acc v*; ⋀*v*. *upd* (*acc v*) *v* = *v* ⟧
⟹ *upd-fa-ind* (*lf-set* (*ti-typ-pad-combine* (*t*::''*a itself*) *acc upd algn fn ti*) []) *h*
⟨*proof*⟩

lemma *upd-fa-ind-empty* [*simp*]:
upd-fa-ind {} *h*
⟨*proof*⟩

lemma *wf-align-empty-typ-info*: *wf-align* (*empty-typ-info algn tn*)
⟨*proof*⟩

lemma *wf-align-list*: *wf-align t* ⟹ *wf-align-list fs* ⟹ *wf-align-list* (*fs* @ [*DTuple t f d*])
⟨*proof*⟩
lemma *wf-align-struct*: *wf-align t* ⟹ *wf-align-struct st* ⟹ *wf-align-struct* (*extend-ti-struct st t f d*)
⟨*proof*⟩

lemma *align-td-extend-ti*: *align-td* (*extend-ti tag t algn f d*) = *max* (*align-td tag*) (*max algn* (*align-td t*))
⟨*proof*⟩

lemma *align-td-struct-extend-ti*: *aggregate-struct st* ⟹
align-td-struct (*extend-ti-struct st t f d*) = *max* (*align-td-struct st*) (*align-td t*)
⟨*proof*⟩
lemma *wf-align-extend-ti'*:

assumes *wf-t*: *wf-align t*
assumes *agg*: *aggregate tag*
assumes *wf-tag*: *wf-align tag*
shows *wf-align* (*extend-ti tag t algn f d*)
⟨*proof*⟩

lemma (**in** *mem-type*) *wf-align-extend-ti*:
assumes *agg*: *aggregate tag*
assumes *wf-tag*: *wf-align tag*
shows *wf-align* (*extend-ti tag* (*typ-info-t* (*TYPE*('*a*))) *algn f d*)
⟨*proof*⟩

lemma *wf-align-map-td* [*simp*]:
wf-align (*map-td f g d*) = *wf-align d*
wf-align-struct (*map-td-struct f g ts*) = *wf-align-struct* (*ts*)
wf-align-list (*map-td-list f g fs*) = *wf-align-list fs*
wf-align-tuple (*map-td-tuple f g fd*) = *wf-align-tuple fd*
⟨*proof*⟩

lemma *wf-align-adjust-ti*[*simp*]: *wf-align* (*adjust-ti t acc upd*) = *wf-align t*
⟨*proof*⟩

lemma (**in** *mem-type*) *wf-align-ti-typ-combine*:
aggregate tag ⟹ *wf-align tag* ⟹ *wf-align* (*ti-typ-combine* (*TYPE*('*a*)) *acc upd algn fn tag*)
⟨*proof*⟩

lemma *wf-align-ti-pad-combine*:
aggregate tag ⟹ *wf-align tag* ⟹ *wf-align* (*ti-pad-combine n tag*)
⟨*proof*⟩

lemma (**in** *mem-type*) *wf-align-ti-typ-pad-combine*:
aggregate tag ⟹ *wf-align tag* ⟹ *wf-align* (*ti-typ-pad-combine* (*TYPE*('*a*)) *acc upd algn fn tag*)
⟨*proof*⟩

lemma *wf-align-map-align*:
assumes *wf-tag*: *wf-align tag*
assumes *mono*: ⋀*a*. *a* ≤ *f a*
shows *wf-align* (*map-align f tag*)
⟨*proof*⟩

lemma *wf-align-final-pad*: *aggregate tag* ⟹ *wf-align tag* ⟹ *wf-align* (*final-pad algn tag*)
⟨*proof*⟩

lemmas *wf-align-simps* =
wf-align-empty-typ-info
wf-align-ti-typ-pad-combine

wf-align-ti-typ-combine
wf-align-ti-pad-combine
wf-align-final-pad

lemma *align-field-empty-typ-info* [*simp*]:
align-field (*empty-typ-info algn tn*)
⟨*proof*⟩

lemma *align-td-wo-align-field-lookup*:
∀ *f m s n*. *field-lookup* (*t*::('*a*,'*b*) *typ-desc*) *f m* = *Some* (*s*,*n*) ⟶ *align-td-wo-align s* ≤ *align-td-wo-align t*
∀ *f m s n*. *field-lookup-struct* (*st*::('*a*,'*b*) *typ-struct*) *f m* = *Some* (*s*,*n*) ⟶ *align-td-wo-align s* ≤ *align-td-wo-align-struct st*
∀ *f m s n*. *field-lookup-list* (*ts*::('*a*,'*b*) *typ-tuple list*) *f m* = *Some* (*s*,*n*) ⟶ *align-td-wo-align s* ≤ *align-td-wo-align-list ts*
∀ *f m s n*. *field-lookup-tuple* (*x*::('*a*,'*b*) *typ-tuple*) *f m* = *Some* (*s*,*n*) ⟶ *align-td-wo-align s* ≤ *align-td-wo-align-tuple x*
⟨*proof*⟩

lemma *align-td-field-lookup*[*rule-format*]:
∀ *f m s n*. *wf-align t* ⟶ *field-lookup* (*t*::('*a*,'*b*) *typ-desc*) *f m* = *Some* (*s*,*n*) ⟶ *align-td s* ≤ *align-td t*
∀ *f m s n*. *wf-align-struct st* ⟶ *field-lookup-struct* (*st*::('*a*,'*b*) *typ-struct*) *f m* = *Some* (*s*,*n*) ⟶ *align-td s* ≤ *align-td-struct st*
∀ *f m s n*. *wf-align-list ts* ⟶ *field-lookup-list* (*ts*::('*a*,'*b*) *typ-tuple list*) *f m* = *Some* (*s*,*n*) ⟶ *align-td s* ≤ *align-td-list ts*
∀ *f m s n*. *wf-align-tuple x* ⟶ *field-lookup-tuple* (*x*::('*a*,'*b*) *typ-tuple*) *f m* = *Some* (*s*,*n*) ⟶ *align-td s* ≤ *align-td-tuple x*
⟨*proof*⟩

lemma (**in** *mem-type*) *align-td-field-lookup-mem-type*: *field-lookup* (*typ-info-t* (*TYPE*('*a*))) *f m* = *Some* (*s*, *n*) ⟹
align-td s ≤ *align-td* (*typ-info-t* (*TYPE*('*a*)))
⟨*proof*⟩

lemma *align-field-extend-ti*:
⟦ *align-field s*; *align-field t*; *wf-align t*; *2*^(*align-td t*) *dvd size-td s* ⟧ ⟹
align-field (*extend-ti s t algn fn d*)
⟨*proof*⟩

lemma *align-field-ti-pad-combine*:
align-field ti ⟹ *align-field* (*ti-pad-combine n ti*)
⟨*proof*⟩

lemma *align-field-map-align* [*simp*]: *align-field* (*map-align f t*) = *align-field t*
⟨*proof*⟩

lemma *align-field-final-pad*:
align-field ti ⟹ *align-field* (*final-pad algn ti*)

⟨*proof*⟩

lemma *field-lookup-adjust-ti-None*:
∀ *fn m s n*. *field-lookup* (*adjust-ti t acc upd*) *fn m* = *None* ⟶
(*field-lookup t fn m* = *None*)
∀ *fn m s n*. *field-lookup-struct* (*map-td-struct* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *st*)
fn m = *None* ⟶
(*field-lookup-struct st fn m* = *None*)
∀ *fn m s n*. *field-lookup-list* (*map-td-list* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *ts*) *fn m* = *None* ⟶
(*field-lookup-list ts fn m* = *None*)
∀ *fn m s n*. *field-lookup-tuple* (*map-td-tuple* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *x*) *fn m* = *None* ⟶
(*field-lookup-tuple x fn m* = *None*)
⟨*proof*⟩

lemma *field-lookup-adjust-ti'* [*rule-format*]:
∀ *fn m s n*. *field-lookup* (*adjust-ti t acc upd*) *fn m* = *Some* (*s*,*n*) ⟶
(∃ *s'*. *field-lookup t fn m* = *Some* (*s'*,*n*) ∧ *align-td-wo-align s* = *align-td-wo-align s'*)
∀ *fn m s n*. *field-lookup-struct* (*map-td-struct* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *st*)
fn m = *Some* (*s*,*n*) ⟶
(∃ *s'*. *field-lookup-struct st fn m* = *Some* (*s'*,*n*) ∧ *align-td-wo-align s* = *align-td-wo-align s'*)
∀ *fn m s n*. *field-lookup-list* (*map-td-list* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *ts*) *fn m* = *Some* (*s*,*n*) ⟶
(∃ *s'*. *field-lookup-list ts fn m* = *Some* (*s'*,*n*) ∧ *align-td-wo-align s* = *align-td-wo-align s'*)
∀ *fn m s n*. *field-lookup-tuple* (*map-td-tuple* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *x*) *fn m* = *Some* (*s*,*n*) ⟶
(∃ *s'*. *field-lookup-tuple x fn m* = *Some* (*s'*,*n*) ∧ *align-td-wo-align s* = *align-td-wo-align s'*)
⟨*proof*⟩

lemma *field-lookup-adjust-ti'''* [*rule-format*]:
∀ *fn m s n*. *field-lookup* (*adjust-ti t acc upd*) *fn m* = *Some* (*s*,*n*) ⟶
(∃ *s'*. *field-lookup t fn m* = *Some* (*s'*,*n*) ∧ *align-td s* = *align-td s'*)
∀ *fn m s n*. *field-lookup-struct* (*map-td-struct* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *st*)
fn m = *Some* (*s*,*n*) ⟶
(∃ *s'*. *field-lookup-struct st fn m* = *Some* (*s'*,*n*) ∧ *align-td s* = *align-td s'*)
∀ *fn m s n*. *field-lookup-list* (*map-td-list* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *ts*) *fn m* = *Some* (*s*,*n*) ⟶
(∃ *s'*. *field-lookup-list ts fn m* = *Some* (*s'*,*n*) ∧ *align-td s* = *align-td s'*)
∀ *fn m s n*. *field-lookup-tuple* (*map-td-tuple* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *x*) *fn m* = *Some* (*s*,*n*) ⟶
(∃ *s'*. *field-lookup-tuple x fn m* = *Some* (*s'*,*n*) ∧ *align-td s* = *align-td s'*)

⟨*proof*⟩

lemma *field-lookup-adjust-ti*:
⟦ *field-lookup* (*adjust-ti t acc upd*) *fn m* = *Some* (*s*,*n*) ⟧ ⟹
(∃ *s'*. *field-lookup t fn m* = *Some* (*s'*,*n*) ∧ *align-td-wo-align s* = *align-td-wo-align s'*)
⟨*proof*⟩

lemma *field-lookup-adjust-ti1*:
⟦ *field-lookup* (*adjust-ti t acc upd*) *fn m* = *Some* (*s*,*n*) ⟧ ⟹
(∃ *s'*. *field-lookup t fn m* = *Some* (*s'*,*n*) ∧ *align-td s* = *align-td s'*)
⟨*proof*⟩

lemma *align-adjust-ti*:
align-field ti ⟹ *align-field* (*adjust-ti ti acc upd*)
⟨*proof*⟩

lemma (**in** *mem-type*) *align-field-ti-typ-combine*:
⟦ *align-field ti*; 2 ^ *align-td* (*typ-info-t TYPE*('*a*)) *dvd size-td ti* ⟧ ⟹
align-field (*ti-typ-combine* (*t*::'*a itself*) *acc upd algn fn ti*)
⟨*proof*⟩

lemma (**in** *mem-type*) *align-td-wo-align-type-info-t-le-align-td*:
align-td-wo-align (*typ-info-t TYPE*('*a*)) ≤ *align-td* (*typ-info-t TYPE*('*a*))
⟨*proof*⟩

lemma (**in** *mem-type*) *align-field-ti-typ-pad-combine*:
⟦*wf-align ti*; *align-field ti*; *aggregate ti*⟧ ⟹
align-field (*ti-typ-pad-combine* (*t*::'*a itself*) *acc upd algn fn ti*)

⟨*proof*⟩

lemma *npf-extend-ti* [*simp*]:
non-padding-fields (*extend-ti s t algn fn d*) = *non-padding-fields s* @
(*if hd fn* = *CHR* ''!'' *then* [] *else* [*fn*])
⟨*proof*⟩

lemma *npf-ti-pad-combine* [*simp*]:
non-padding-fields (*ti-pad-combine n tag*) = *non-padding-fields tag*
⟨*proof*⟩

lemma (**in** *c-type*) *npf-ti-typ-combine* [*simp*]:
hd fn ≠ *CHR* ''!'' ⟹
non-padding-fields (*ti-typ-combine* (*t*::'*a itself*) *acc upd algn fn tag*) = *non-padding-fields tag* @ [*fn*]
⟨*proof*⟩

lemma (**in** *c-type*) *npf-ti-typ-pad-combine* [*simp*]:
hd fn ≠ *CHR* ''!'' ⟹

non-padding-fields (*ti-typ-pad-combine* (*t*::*'a itself*) *acc upd algn fn tag*) = *non-padding-fields tag* @ [*fn*]
⟨*proof*⟩

lemma *non-padding-fields-map-align* [*simp*]:
non-padding-fields (*map-align f t*) = *non-padding-fields t*
⟨*proof*⟩

lemma *npf-final-pad* [*simp*]:
non-padding-fields (*final-pad algn tag*) = *non-padding-fields tag*
⟨*proof*⟩

lemma *npf-empty-typ-info* [*simp*]:
non-padding-fields (*empty-typ-info algn tn*) = []
⟨*proof*⟩

definition *field-fd'* :: *'a xtyp-info* ⇒ *qualified-field-name* ⇀ *'a field-desc* **where**
field-fd' t f ≡ *case field-lookup t f 0 of None* ⇒ *None* | *Some x* ⇒ *Some* (*field-desc* (*fst x*))

lemma *padup-zero* [*simp*]:
padup n 0 = *0*
⟨*proof*⟩

lemma *padup-same* [*simp*]:
padup n n = *0*
⟨*proof*⟩

lemmas *size-td-simps-0* = *size-td-lt-final-pad size-td-lt-ti-typ-pad-combine*

lemmas *size-td-simps-1* = *size-td-simps-0*
aggregate-ti-typ-pad-combine aggregate-empty-typ-info

lemmas *size-td-simps-2* = *padup-def align-of-final-pad align-of-def*

lemmas *size-td-simps* = *size-td-simps-1 size-td-simps-2*

lemmas *size-td-simps-3* = *size-td-simps-0 size-td-simps-2*

lemma *fu-commutes-sym*: *fu-commutes x y* = *fu-commutes y x*
⟨*proof*⟩

lemma *wf-lf-insert-recursion*:
assumes *wf-D*: *wf-lf D*
assumes *cons-x*: *fd-cons-desc* (*lf-fd x*) (*lf-sz x*)
assumes *comm-D*: ⋀*y*. *y* ∈ *D* ⟹ *fu-commutes* (*field-update* (*lf-fd x*)) (*field-update* (*lf-fd y*)) ∧

fa-fu-ind (*lf-fd x*) (*lf-fd y*) (*lf-sz y*) (*lf-sz x*) ∧
fa-fu-ind (*lf-fd y*) (*lf-fd x*) (*lf-sz x*) (*lf-sz y*)
shows *wf-lf* (*insert x D*)
⟨*proof*⟩

lemma *wf-lf-insert-recursion′*:
assumes *cons-x*: *fd-cons-desc* (*lf-fd x*) (*lf-sz x*)
assumes *comm-D*: ⋀*y*. *y* ∈ *D* ⟹ *fu-commutes* (*field-update* (*lf-fd x*)) (*field-update* (*lf-fd y*)) ∧
fa-fu-ind (*lf-fd x*) (*lf-fd y*) (*lf-sz y*) (*lf-sz x*) ∧
fa-fu-ind (*lf-fd y*) (*lf-fd x*) (*lf-sz x*) (*lf-sz y*)
assumes *wf-D*: *wf-lf D*
shows *wf-lf* (*insert x D*)
⟨*proof*⟩

lemma *wf-lf-insert-recursion″*:
assumes *wf-D*: *wf-lf D*
assumes *cons-x*: *fd-cons-desc* (*lf-fd x*) (*lf-sz x*)
assumes *comm-D*: ⋀*y*. *y* ∈ *D* ⟹ *lf-fn y* ≠ *lf-fn x* ⟹ *fu-commutes* (*field-update* (*lf-fd x*)) (*field-update* (*lf-fd y*)) ∧
fa-fu-ind (*lf-fd x*) (*lf-fd y*) (*lf-sz y*) (*lf-sz x*) ∧
fa-fu-ind (*lf-fd y*) (*lf-fd x*) (*lf-sz x*) (*lf-sz y*)
shows *wf-lf* (*insert x D*)
⟨*proof*⟩

lemma *wf-field-desc-empty* [*simp*]:
wf-field-desc (|*field-access* = λ*v bs*. [], *field-update* = λ*bs*. *id*, *field-sz* = *0*|)
⟨*proof*⟩

lemma *field-desc-independent-subset*:
D ⊆ *E* ⟹ *field-desc-independent acc upd E* ⟹ *field-desc-independent acc upd D*
⟨*proof*⟩

lemma *field-desc-independent-union-iff*:
field-desc-independent acc upd (*D* ∪ *E*) =
(*field-desc-independent acc upd D* ∧ *field-desc-independent acc upd E*)
⟨*proof*⟩

lemma *field-desc-independent-unionI*:
field-desc-independent acc upd D ⟹ *field-desc-independent acc upd E* ⟹
field-desc-independent acc upd (*D* ∪ *E*)
⟨*proof*⟩

lemma *field-desc-independent-unionD1*:
field-desc-independent acc upd $(D \cup E) \Longrightarrow$
field-desc-independent acc upd D
⟨*proof*⟩

lemma *field-desc-independent-unionD2*:
field-desc-independent acc upd $(D \cup E) \Longrightarrow$
field-desc-independent acc upd E
⟨*proof*⟩

lemma *field-desc-independent-insert-iff*:
field-desc-independent acc upd (*insert d D*) =
(*field-desc-independent acc upd* $\{d\} \wedge$ *field-desc-independent acc upd D*)
⟨*proof*⟩

lemma *field-desc-independent-insertI*:
field-desc-independent acc upd $\{d\} \Longrightarrow$ *field-desc-independent acc upd D* $\Longrightarrow$
field-desc-independent acc upd (*insert d D*)
⟨*proof*⟩

lemma *field-desc-independent-insertD1*:
assumes *ins*: *field-desc-independent acc upd* (*insert d D*)
shows *field-desc-independent acc upd* $\{d\}$
⟨*proof*⟩

lemma *field-desc-independent-insertD2*:
assumes *ins*: *field-desc-independent acc upd* (*insert d D*)
shows *field-desc-independent acc upd D*
⟨*proof*⟩

lemma *field-descs-independent-append-first*: *field-descs-independent* (*xs* @ *ys*) $\Longrightarrow$
field-descs-independent xs
⟨*proof*⟩

lemma *field-descs-independent-append-second*: *field-descs-independent* (*xs* @ *ys*)
$\Longrightarrow$ *field-descs-independent ys*
⟨*proof*⟩

lemma *field-descs-independent-append-first-ind*:
field-descs-independent (*xs* @ *ys*) $\Longrightarrow$ *x* $\in$ *set xs* $\Longrightarrow$
field-desc-independent (*field-access x*) (*field-update x*) (*set ys*)
⟨*proof*⟩

lemma *field-descs-independent-appendI1*:
field-descs-independent xs $\Longrightarrow$ *field-descs-independent ys* $\Longrightarrow$
($\forall$ *x* $\in$ *set xs*. *field-desc-independent* (*field-access x*) (*field-update x*) (*set ys*)) $\Longrightarrow$
field-descs-independent (*xs* @ *ys*)

⟨*proof*⟩

lemma *field-desc-independent-symmetric*:
(∀ *x* ∈ *X*. *field-desc-independent* (*field-access x*) (*field-update x*) *Y*) ⟹
(∀ *y* ∈ *Y*. *field-desc-independent* (*field-access y*) (*field-update y*) *X*)
⟨*proof*⟩

lemma *field-desc-independent-symmetric-singleton*:
field-desc-independent (*field-access x*) (*field-update x*) *Y* ⟹ *y* ∈ *Y*
⟹ *field-desc-independent* (*field-access y*) (*field-update y*) {*x*}
⟨*proof*⟩

lemma *field-descs-independent-appendI2*:
assumes *ind-xs*: *field-descs-independent xs*
assumes *ind-ys*: *field-descs-independent ys*
assumes *ind-xs-ys*: ∀ *y* ∈ *set ys*. *field-desc-independent* (*field-access y*) (*field-update y*) (*set xs*)
shows *field-descs-independent* (*xs* @ *ys*)
⟨*proof*⟩

lemma *field-desc-indepenent-empty* [*simp*]:
field-desc-independent (*field-access d*) (*field-update d*) {} ⟨*proof*⟩

lemma *field-update-apply-field-updates-commute*:
fixes *d*::′*a field-desc*
fixes *ds*::′*a field-desc list*
assumes *ind-d*:*field-desc-independent* (*field-access d*) (*field-update d*) (*set ds*)
assumes *ind-ds*: *field-descs-independent ds*
shows *field-update d bs* (*fst* (*apply-field-updates ds bs*′ *s*)) =
fst (*apply-field-updates ds bs*′ (*field-update d bs s*))
⟨*proof*⟩

lemma (**in** *wf-field-desc*) *is-padding-byte-acc-upd-compose*:
assumes *acc-upd*: ⋀*v s*. *acc* (*upd v s*) = *v*
shows *padding-base.is-padding-byte* (*field-access d* ∘ *acc*)
(λ*bs v*. *upd* (*field-update d bs* (*acc v*)) *v*) (*field-sz d*) *i* =
is-padding-byte i
⟨*proof*⟩

lemma (**in** *wf-field-desc*) *is-value-byte-acc-upd-compose*:
assumes *acc-upd*: ⋀*v s*. *acc* (*upd v s*) = *v*
shows *padding-base.is-value-byte* (*field-access d* ∘ *acc*)
(λ*bs v*. *upd* (*field-update d bs* (*acc v*)) *v*) (*field-sz d*) *i* =
is-value-byte i
⟨*proof*⟩

lemma (**in** *wf-field-desc*) *wf-field-desc-update-desc*:
assumes *double-update-upd*: $\bigwedge v\ w\ s.\ upd\ v\ (upd\ w\ s) = upd\ v\ s$
assumes *acc-upd*: $\bigwedge v\ s.\ acc\ (upd\ v\ s) = v$
assumes *upd-acc*: $\bigwedge s.\ upd\ (acc\ s)\ s = s$
shows *wf-field-desc* (*update-desc acc upd d*) (**is** *wf-field-desc ?upd*)
⟨*proof*⟩

lemma *toplevel-field-descs-subset*:
fixes *t*::'*a xtyp-info* **and**
st::'*a xtyp-info-struct* **and**
fs::'*a xtyp-info-tuple list* **and**
f::'*a xtyp-info-tuple*
shows
set (*toplevel-field-descs t*) ⊆ *set* (*field-descs t*)
set (*toplevel-field-descs-struct st*) ⊆ *set* (*field-descs-struct st*)
set (*toplevel-field-descs-list fs*) ⊆ *set* (*field-descs-list fs*)
set (*toplevel-field-descs-tuple f*) ⊆ *set* (*field-descs-tuple f*)
⟨*proof*⟩

lemma
fixes *t*::'*a xtyp-info* **and**
st::'*a xtyp-info-struct* **and**
fs::'*a xtyp-info-tuple list* **and**
f::'*a xtyp-info-tuple*
shows
lf-fd ‘ *lf-set t xs* ⊆ *set* (*field-descs t*)
lf-fd ‘ *lf-set-struct st xs* ⊆ *set* (*field-descs-struct st*)
lf-fd ‘ *lf-set-list fs xs* ⊆ *set* (*field-descs-list fs*)
lf-fd ‘ *lf-set-tuple f xs* ⊆ *set* (*field-descs-tuple f*)
⟨*proof*⟩

lemma *lf-set-subset-field-descs*:
fixes *t*::'*a xtyp-info* **and**
st::'*a xtyp-info-struct* **and**
fs::'*a xtyp-info-tuple list* **and**
f::'*a xtyp-info-tuple*
shows
$\bigwedge xs.$ *lf-fd* ‘ *lf-set t xs* ⊆ *set* (*field-descs t*)
$\bigwedge xs.$ *lf-fd* ‘ *lf-set-struct st xs* ⊆ *set* (*field-descs-struct st*)
$\bigwedge xs.$ *lf-fd* ‘ *lf-set-list fs xs* ⊆ *set* (*field-descs-list fs*)
$\bigwedge xs.$ *lf-fd* ‘ *lf-set-tuple f xs* ⊆ *set* (*field-descs-tuple f*)
⟨*proof*⟩

lemma *wf-field-descs-empty-typ-info* [*simp*]: *wf-field-descs* (*set* (*field-descs* (*empty-typ-info algn t*)))
⟨*proof*⟩

lemma *field-descs-map*:
field-descs (*map-td* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *t*) =
map (*update-desc acc upd*) (*field-descs t*)
field-descs-struct (*map-td-struct* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *st*) =
map (*update-desc acc upd*) (*field-descs-struct st*)
field-descs-list (*map-td-list* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *fs*) =
map (*update-desc acc upd*) (*field-descs-list fs*)
field-descs-tuple (*map-td-tuple* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *f*) =
map (*update-desc acc upd*) (*field-descs-tuple f*)
⟨*proof*⟩

lemma *component-update-access*:
assumes *wf*:*wf-field-descs* (*set* (*field-descs t*))
shows *field-update* (*component-desc t*)
(*field-access* (*component-desc t*) *s bs*) *s* = *s*
⟨*proof*⟩

lemma *toplevel-field-descs-tuple-dt-trd*: *toplevel-field-descs-tuple t* = [*dt-trd t*]
⟨*proof*⟩
lemma *toplevel-field-descs-list-map*: *toplevel-field-descs-list fs* = *map dt-trd fs*
⟨*proof*⟩

lemma *component-double-update*:
assumes *wf*:*wf-field-descs* (*set* (*field-descs t*))
assumes *ind*: *field-descs-independent* (*toplevel-field-descs t*)
shows*field-update* (*component-desc t*) *bs* (*field-update* (*component-desc t*) *bs*′ *s*) =
field-update (*component-desc t*) *bs s*
⟨*proof*⟩

lemma *component-access-update*:
assumes *wf*:*wf-field-descs* (*set* (*field-descs t*))
assumes *ind*: *field-descs-independent* (*toplevel-field-descs t*)
shows *field-access* (*component-desc t*) (*field-update* (*component-desc t*) *bs s*) *bs*′ =
field-access (*component-desc t*) (*field-update* (*component-desc t*) *bs s*′) *bs*′
⟨*proof*⟩

lemma *sum-nat-foldl*: (*n*::*nat*) + *foldl* (+) *m xs* = *foldl* (+) (*n* + *m*) *xs*
⟨*proof*⟩

lemma *sum-nat-foldl-le*: (*n*::*nat*) ≤ *foldl* (+) *n xs*
⟨*proof*⟩

lemma *sum-add-sub-foldl*: *foldl* (+) (*m* + *n*) *ns* − *n* = *foldl* (+) (*m*::*nat*) *ns*
⟨*proof*⟩

lemma *sum-sub-zero-foldl*: *foldl* (+) *n ns* − (*n*::*nat*) = *foldl* (+) *0 ns*
⟨*proof*⟩

lemma *drop-take-sub*: *drop n* (*take* (*foldl* (+) *n ns*) *bs*) = *take* (*foldl* (+) *0 ns*) (*drop n bs*)
⟨*proof*⟩

lemma *component-access-size*:
assumes *wf*: *wf-field-descs* (*set* (*field-descs t*))
shows *field-access* (*component-desc t*) *s* (*take* (*field-sz* (*component-desc t*)) *bs*) =
field-access (*component-desc t*) *s bs*
⟨*proof*⟩

lemma *component-access-result-size*:
assumes *wf*: *wf-field-descs* (*set* (*field-descs t*))
shows *length* (*field-access* (*component-desc t*) *s bs*) = *field-sz* (*component-desc t*)
⟨*proof*⟩

lemma *component-update-size*:
assumes *wf*: *wf-field-descs* (*set* (*field-descs t*))
shows *field-update* (*component-desc t*) (*take* (*field-sz* (*component-desc t*)) *bs*) *s* =
field-update (*component-desc t*) *bs s*
⟨*proof*⟩

lemma *apply-field-updates-access-cancel*:
assumes *wf*: *wf-field-descs* (*set* (*field-descs-list fs*))
assumes *ind*: *field-descs-independent* (*toplevel-field-descs-list fs*)
shows *fst* (*apply-field-updates* (*map dt-trd fs*) (*concat* (*split-map* (*component-access-tuple v*) *fs bs*)) *v*) = *v*
⟨*proof*⟩

lemma *apply-field-updates-double*:
assumes *wf*: *wf-field-descs* (*set* (*field-descs-list fs*))
assumes *ind*: *field-descs-independent* (*toplevel-field-descs-list fs*)
shows *fst* (*apply-field-updates* (*map dt-trd fs*) *bs*

(*fst* (*apply-field-updates* (*map dt-trd fs*) *bs'* *v*))) =
fst (*apply-field-updates* (*map dt-trd fs*) *bs* *v*)
⟨*proof*⟩

lemma *list-append-eq-split-conv*: *length xs1* = *length xs2* ⟹ *xs1* @ *ys1* = *xs2* @ *ys2* ⟷ *xs1* = *xs2* ∧ *ys1* = *ys2*
⟨*proof*⟩

lemma *component-complement-padding*:
assumes *wf*: *wf-field-descs* (*set* (*field-descs t*))
assumes *ind*: *field-descs-independent* (*toplevel-field-descs t*)
assumes *i-bound*: *i* < *field-sz* (*component-desc t*)
shows *padding-base.is-padding-byte* (*field-access* (*component-desc t*))
(*field-update* (*component-desc t*)) (*field-sz* (*component-desc t*)) *i* ≠
padding-base.is-value-byte (*field-access* (*component-desc t*))
(*field-update* (*component-desc t*)) (*field-sz* (*component-desc t*)) *i*
⟨*proof*⟩

theorem *wf-field-desc-component-desc*:
assumes *wf*:*wf-field-descs* (*set* (*field-descs t*))
assumes *ind*: *field-descs-independent* (*toplevel-field-descs t*)
shows *wf-field-desc* (*component-desc t*)
⟨*proof*⟩

lemma *field-desc-list-append* [*simp*]: *field-descs-list* (*fs1* @ *fs2*) = *field-descs-list fs1* @ *field-descs-list fs2*
⟨*proof*⟩

lemma *toplevel-field-desc-list-append* [*simp*]:
toplevel-field-descs-list (*fs1* @ *fs2*) = *toplevel-field-descs-list fs1* @ *toplevel-field-descs-list fs2*
⟨*proof*⟩

corollary *wf-field-descs-struct-append-first*:
assumes *wf*:*wf-field-descs* (*set* (*field-descs-struct* (*TypAggregate* (*fs1* @ *fs2*))))
assumes *ind*: *field-descs-independent* (*toplevel-field-descs-struct* (*TypAggregate* (*fs1* @ *fs2*)))
shows *wf-field-desc* (*component-desc-struct* (*TypAggregate fs1*))
⟨*proof*⟩

corollary *wf-field-descs-struct-append-second*:
assumes *wf*:*wf-field-descs* (*set* (*field-descs-struct* (*TypAggregate* (*fs1* @ *fs2*))))
assumes *ind*: *field-descs-independent* (*toplevel-field-descs-struct* (*TypAggregate* (*fs1* @ *fs2*)))

shows *wf-field-desc* (*component-desc-struct* (*TypAggregate fs2*))
⟨*proof*⟩

corollary *wf-field-descs-component-desc*:
assumes *wf*:*wf-field-descs* (*set* (*field-descs t*))
assumes *ind*: *field-descs-independent* (*toplevel-field-descs t*)
shows *wf-field-descs* (*insert* (*component-desc t*) (*set* (*field-descs t*)))
⟨*proof*⟩

lemma *size-td-field-sz*:
fixes
t::'*a xtyp-info* **and**
st::'*a xtyp-info-struct* **and**
fs::'*a xtyp-info-tuple list* **and**
f::'*a xtyp-info-tuple*
shows
wf-component-descs t ⟹ *size-td t* = *field-sz* (*component-desc t*)
wf-component-descs-struct st ⟹ *size-td-struct st* = *field-sz* (*component-desc-struct st*)
wf-component-descs-list fs ⟹ *size-td-list fs* = *field-sz* (*component-desc-struct* (*TypAggregate fs*))
wf-component-descs-tuple f ⟹ *size-td-tuple f* = *field-sz* (*dt-trd f*)
⟨*proof*⟩

lemma (**in** *c-type*) *simple-type-to-xto-eq*:
fixes *v*::'*a*
assumes *simple*: ¬ *aggregate* (*typ-info-t TYPE*('*a*))
shows *to-bytes v* = *xto-bytes v*
⟨*proof*⟩

lemma *field-desc-independent-empty* [*simp*]: *field-desc-independent acc upd* {}
⟨*proof*⟩

lemma *component-descs-field-descs-independent*:
component-descs-independent t ⟹ *field-descs-independent* (*toplevel-field-descs t*)
⟨*proof*⟩

lemma *component-descs-list-field-descs-independent*:
component-descs-independent-list fs ⟹ *field-descs-independent* (*toplevel-field-descs-list fs*)
⟨*proof*⟩

theorem *access-ti-component-desc-compatible*:
⋀*bs v*::'*a*. ⟦*wf-component-descs t*; *component-descs-independent t*;
wf-field-descs (*set* (*field-descs t*))⟧
⟹ *access-ti t v bs* = *field-access* (*component-desc t*) *v bs*
⋀*bs v*::'*a*. ⟦*wf-component-descs-struct st*; *component-descs-independent-struct st*;
wf-field-descs (*set* (*field-descs-struct st*))⟧
⟹ *access-ti-struct st v bs* = *field-access* (*component-desc-struct st*) *v bs*
⋀*bs v*::'*a*. ⟦*wf-component-descs-list fs*; *component-descs-independent-list fs*;
wf-field-descs (*set* (*field-descs-list fs*))⟧
⟹ *access-ti-list fs v bs* = *field-access* (*component-desc-struct* (*TypAggregate fs*))
v bs
⋀*bs v*::'*a*. ⟦*wf-component-descs-tuple f*; *component-descs-independent-tuple f*;
wf-field-descs (*set* (*field-descs* (*dt-fst f*)))⟧
⟹ *access-ti-tuple f v bs* = *field-access* (*dt-trd f*) *v bs*
⟨*proof*⟩

theorem *update-ti-component-desc-compatible*:
⋀*bs v*::'*a*. ⟦*wf-component-descs t*; *component-descs-independent t*;
wf-field-descs (*set* (*field-descs t*))⟧
⟹ *update-ti t bs v* = *field-update* (*component-desc t*) *bs v*
⋀*bs v*::'*a*. ⟦*wf-component-descs-struct st*; *component-descs-independent-struct st*;
wf-field-descs (*set* (*field-descs-struct st*))⟧
⟹ *update-ti-struct st bs v* = *field-update* (*component-desc-struct st*) *bs v*
⋀*bs v*::'*a*. ⟦*wf-component-descs-list fs*; *component-descs-independent-list fs*;
wf-field-descs (*set* (*field-descs-list fs*))⟧
⟹ *update-ti-list fs bs v* = *field-update* (*component-desc-struct* (*TypAggregate*
fs)) *bs v*
⋀*bs v*::'*a*. ⟦*wf-component-descs-tuple f*; *component-descs-independent-tuple f*;
wf-field-descs (*set* (*field-descs* (*dt-fst f*)))⟧
⟹ *update-ti-tuple f bs v* = *field-update* (*dt-trd f*) *bs v*
⟨*proof*⟩

context *xmem-type*
begin

lemma *wf-field-desc-component-desc*: *wf-field-desc* (*component-desc* (*typ-info-t* (*TYPE*('*a*))))
⟨*proof*⟩

lemma *wf-field-descs'*: *wf-field-descs* (*insert* (*component-desc* (*typ-info-t* (*TYPE*('*a*))))
(*set* (*field-descs* (*typ-info-t* (*TYPE*('*a*)))))))
⟨*proof*⟩

lemma *wf-field-desc-t*: *wf-field-desc* (*component-desc* (*typ-info-t* (*TYPE*('*a*))))
⟨*proof*⟩

sublocale *xmem-type-wf-field-desc*: *wf-field-desc component-desc* (*typ-info-t* (*TYPE*('*a*)))
⟨*proof*⟩

lemma *xfrom-bytes-xto-bytes-inv*: *xfrom-bytes* (*xto-bytes* (*v*::'*a*) *bs*) = *v*
⟨*proof*⟩

lemma *xto-bytes-inj*:
xto-bytes (*v*::'*a*) = *xto-bytes* (*v'*::'*a*) ⟹ *v* = *v'*
⟨*proof*⟩

lemma *field-update-update-ti-t*:
assumes *len*: *length bs* = *size-td* (*typ-info-t* (*TYPE*('*a*)))
shows *field-update* (*component-desc* (*typ-info-t* (*TYPE*('*a*)))) *bs* = *update-ti-t* (*typ-info-t* (*TYPE*('*a*))) *bs*
⟨*proof*⟩

lemma *field-update-update-ti*:
shows *field-update* (*component-desc* (*typ-info-t* (*TYPE*('*a*)))) = *update-ti* (*typ-info-t* (*TYPE*('*a*)))
⟨*proof*⟩

lemma *field-access-access-ti*:
shows *field-access* (*component-desc* (*typ-info-t* (*TYPE*('*a*)))) = *access-ti* (*typ-info-t* (*TYPE*('*a*)))
⟨*proof*⟩

lemma *field-sz-size-td*:
field-sz (*component-desc* (*typ-info-t* (*TYPE*('*a*)))) = *size-td* (*typ-info-t* (*TYPE*('*a*)))
⟨*proof*⟩

lemma *field-sz-size-of*:
field-sz (*component-desc* (*typ-info-t* (*TYPE*('*a*)))) = *size-of* (*TYPE*('*a*))
⟨*proof*⟩

lemma *xto-bytes-size*: *xto-bytes* (*v*::'*a*) (*take* (*size-td* (*typ-info-t TYPE*('*a*))) *bs*) = *xto-bytes v bs*
⟨*proof*⟩

lemma *xto-bytes-result-size*: *length* (*xto-bytes* (*v*::'*a*) *bs*) = *size-td* (*typ-info-t TYPE*('*a*))
⟨*proof*⟩

lemma *xfrom-bytes-size*: (*xfrom-bytes*::*byte list* ⇒ '*a*) (*take* (*size-td* (*typ-info-t TYPE*('*a*))) *bs*) = *xfrom-bytes bs*
⟨*proof*⟩

lemma *entire-update*:
shows *field-update* (*component-desc* (*typ-info-t* (*TYPE*('*a*)))) *bs v* = *field-update* (*component-desc* (*typ-info-t* (*TYPE*('*a*)))) *bs w*

⟨*proof*⟩

lemma *update-access-complete*:
shows *field-update* (*component-desc* (*typ-info-t* (*TYPE*('*a*)))) (*field-access* (*component-desc* (*typ-info-t* (*TYPE*('*a*)))) *v bs*) *w* = *v*
⟨*proof*⟩

end

lemma *contained-field-descs-list-all*: *contained-field-descs-list xs* = *list-all contained-field-descs-tuple xs*
⟨*proof*⟩

lemma *toplevel-field-descs-map*:
toplevel-field-descs (*map-td* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *t*) =
map (*update-desc acc upd*) (*toplevel-field-descs t*)
toplevel-field-descs-struct (*map-td-struct* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *st*) =
map (*update-desc acc upd*) (*toplevel-field-descs-struct st*)
toplevel-field-descs-list (*map-td-list* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *fs*) =
map (*update-desc acc upd*) (*toplevel-field-descs-list fs*)
toplevel-field-descs-tuple (*map-td-tuple* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *f*) =
map (*update-desc acc upd*) (*toplevel-field-descs-tuple f*)
⟨*proof*⟩

lemma *toplevel-field-descs-adjust-ti*: *toplevel-field-descs* (*adjust-ti t acc upd*) = *map* (*update-desc acc upd*) (*toplevel-field-descs t*)
⟨*proof*⟩

theorem *contained-field-descs-empty-typ-info* [*simp*]: *contained-field-descs* (*empty-typ-info algn n*)
⟨*proof*⟩

lemma *fields-contained-update-desc*:
assumes *contained-t*: *fields-contained* (*field-access d*) (*field-update d*) (*set* (*toplevel-field-descs t*))
shows *fields-contained* (*field-access* (*update-desc acc upd d*))
(*field-update* (*update-desc acc upd d*))
(*update-desc acc upd* ‘ *set* (*toplevel-field-descs t*))
⟨*proof*⟩

lemma *contained-field-descs-update-desc*:
contained-field-descs t $\Longrightarrow$
contained-field-descs (*map-td* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *t*)
contained-field-descs-struct st $\Longrightarrow$
contained-field-descs-struct (*map-td-struct* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *st*)
contained-field-descs-list fs $\Longrightarrow$
contained-field-descs-list (*map-td-list* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *fs*)
contained-field-descs-tuple f $\Longrightarrow$
contained-field-descs-tuple (*map-td-tuple* (λ- -. *update-desc acc upd*) (*update-desc acc upd*) *f*)
⟨*proof*⟩

theorem *contained-field-descs-adjust-ti*:
fixes *t*::*'a xtyp-info*
assumes *contained-t*: *contained-field-descs t*
shows *contained-field-descs* (*adjust-ti t acc upd*)
⟨*proof*⟩

theorem *contained-field-descs-extend-ti*:
assumes *contained-t*: *contained-field-descs t*
assumes *contained-ft*: *contained-field-descs ft*
assumes *fields-contained-ft*: *fields-contained* (*field-access d*) (*field-update d*) (*set* (*toplevel-field-descs ft*))
shows *contained-field-descs* (*extend-ti t ft algn n d*)
⟨*proof*⟩

theorem *contained-field-descs-ti-pad-combine*:
fixes *t*::*'a xtyp-info*
assumes *cont-t*: *contained-field-descs t*
shows *contained-field-descs* (*ti-pad-combine n t*)
⟨*proof*⟩

lemma *contained-field-descs-simp*[*simp*]:
contained-field-descs (*map-align f t*) = *contained-field-descs t*
⟨*proof*⟩

theorem *contained-field-descs-final-pad*:
fixes *t*::*'a xtyp-info*
assumes *cont-t*: *contained-field-descs t*
shows *contained-field-descs* (*final-pad algn t*)
⟨*proof*⟩

lemma (**in** *xmem-type*) *fields-contained-update-desc-mem-type*:
fixes *acc*:: *'b* ⇒ *'a*
fixes *D*:: *'a field-desc set*
shows *fields-contained* (*xto-bytes* ∘ *acc*) (*upd* ∘ *xfrom-bytes*) (*update-desc acc upd*

‘ *D*)
⟨*proof*⟩

locale *wf-field* =
  fixes *acc*::′*s* ⇒ ′*a*
  fixes *upd*::′*a* ⇒ ′*s* ⇒ ′*s*
  assumes *double-update-upd*: ⋀*v w s*. *upd v* (*upd w s*) = *upd v s*
  assumes *acc-upd*: ⋀*v s*. *acc* (*upd v s*) = *v*
  assumes *upd-acc*: ⋀*s*. *upd* (*acc s*) *s* = *s*

locale *wf-cfield* = *wf-field* +
  constrains *acc*::′*s* ⇒ ′*a*::*c-type*
locale *wf-xfield* = *wf-cfield* +
  constrains *acc*::′*s* ⇒ ′*a*::*xmem-type*

lemma (**in** *wf-field*) *wf-field-descs-adjust-ti*:
  assumes *wf-t*: *wf-field-descs* (*set* (*field-descs t*))
  shows *wf-field-descs* (*set* (*field-descs* (*adjust-ti t acc upd*)))
⟨*proof*⟩

lemma *field-descs-list-append* [*simp*]: *field-descs-list* (*fs* @ *fs*′) = *field-descs-list fs* @ *field-descs-list fs*′
  ⟨*proof*⟩

lemma *wf-field-descs-extend-ti*:
  assumes *wf-t*: *wf-field-descs* (*set* (*field-descs t*))
  assumes *wf-ft*: *wf-field-descs* (*set* (*field-descs ft*))
  assumes *wf-d*: *wf-field-desc d*
  shows *wf-field-descs* (*set* (*field-descs* (*extend-ti t ft algn fn d*)))
⟨*proof*⟩

lemma *padding-pad*: *complement-padding* (λ*v bs*. *if n* ≤ *length bs then take n bs else replicate n 0*) (λ*bs*. *id*) *n*
  ⟨*proof*⟩

lemma *wf-field-desc-pad*:
wf-field-desc
    (|*field-access* = λ*v bs*. *if n* ≤ *length bs then take n bs else replicate n 0*,
     field-update = λ*bs*. *id*, *field-sz* = *n*|)

⟨*proof*⟩

lemma *wf-field-descs-ti-pad-combine*: *wf-field-descs* (*set* (*field-descs t*)) ⟹
wf-field-descs (*set* (*field-descs* (*ti-pad-combine n t*)))
⟨*proof*⟩

lemma *set-field-descs-map-align*[*simp*]: *set* (*field-descs* (*map-align f t*)) = *set* (*field-descs t*)
⟨*proof*⟩

lemma *wf-field-descs-final-pad*: *wf-field-descs* (*set* (*field-descs t*)) ⟹
wf-field-descs (*set* (*field-descs* (*final-pad algn t*)))
⟨*proof*⟩

lemma (**in** *wf-xfield*) *padding-lift*:
shows *complement-padding* (*xto-bytes* ∘ *acc*) (*upd* ∘ *xfrom-bytes*) (*size-of TYPE*('*a*))
⟨*proof*⟩

lemma (**in** *wf-xfield*) *wf-field-desc-lift*:
shows *wf-field-desc* (|*field-access* = *xto-bytes* ∘ *acc*,
field-update = *upd* ∘ *xfrom-bytes*,
field-sz = *size-of* (*TYPE*('*a*))|) (**is** *wf-field-desc ?lift*)
⟨*proof*⟩

lemma (**in** *wf-xfield*) *wf-field-descs-ti-typ-combine*:
assumes *wf-ft*: *wf-field-descs* (*set* (*field-descs ft*))
shows *wf-field-descs* (*set* (*field-descs* (*ti-typ-combine* (*TYPE*('*a*)) *acc upd algn fn ft*)))
⟨*proof*⟩

lemma (**in** *wf-xfield*) *wf-field-descs-ti-typ-pad-combine*:
assumes *wf-ft*: *wf-field-descs* (*set* (*field-descs ft*))
shows *wf-field-descs* (*set* (*field-descs* (*ti-typ-pad-combine* (*TYPE*('*a*)) *acc upd algn fn ft*)))
⟨*proof*⟩

lemma *wf-component-descs-empty-typ-info* [*simp*]: *wf-component-descs* (*empty-typ-info algn n*)
⟨*proof*⟩

lemma *wf-component-descs-list-append* [*simp*]:
wf-component-descs-list (*fs* @ *fs'*) = (*wf-component-descs-list fs* ∧ *wf-component-descs-list*

fs')
⟨*proof*⟩

lemma *wf-component-descs-extend-ti*:
assumes *wf-t*: *wf-component-descs t*
assumes *wf-ft*: *wf-component-descs ft*
assumes *d*: *d* = (*component-desc ft*)
shows *wf-component-descs* (*extend-ti t ft algn fn d*)
⟨*proof*⟩

lemma *wf-component-descs-ti-pad-combine*:
assumes *wf-t*: *wf-component-descs t*
shows *wf-component-descs* (*ti-pad-combine n t*)
⟨*proof*⟩

lemma *wf-component-descs-map-align* [*simp*]: *wf-component-descs* (*map-align f t*)
= *wf-component-descs t*
⟨*proof*⟩

lemma *wf-component-descs-final-pad*:
assumes *wf-t*: *wf-component-descs t*
shows *wf-component-descs* (*final-pad algn t*)
⟨*proof*⟩

lemma *field-sz-update-desc* [*simp*]: *field-sz* (*update-desc acc upd d*) = *field-sz d*
⟨*proof*⟩

lemma *component-sz-struct-update-desc-commutes*:
(*component-sz-struct* (*TypAggregate fs*)) =
component-sz-struct (*map-td-struct* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) (*TypAggregate fs*))
⟨*proof*⟩

lemma *component-access-struct-update-desc-commutes'*:
component-access-struct (*TypAggregate fs*) (*acc v*) *bs* =
component-access-struct (*map-td-struct* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) (*TypAggregate fs*)) *v bs*
⟨*proof*⟩

lemma *component-access-struct-update-desc-commutes*:
(*component-access-struct* (*TypAggregate fs*)) ∘ *acc* =
component-access-struct (*map-td-struct* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) (*TypAggregate fs*))
⟨*proof*⟩

lemma (**in** *wf-field*) *lemma-component-update-struct-update-desc-commutes*:
shows *upd* (*component-update-struct* (*TypAggregate fs*) *bs* (*acc v*)) *v* =
component-update-struct (*map-td-struct* (λ*n algn*. *update-desc acc upd*)
(*update-desc acc upd*) (*TypAggregate fs*)) *bs v*
⟨*proof*⟩

lemma (**in** *wf-field*) *update-desc-component-desc-struct-commutes*:
shows *update-desc acc upd* (*component-desc-struct* (*TypAggregate fs*)) =
component-desc-struct (*map-td-struct* (λ*n algn*. *update-desc acc upd*)
(*update-desc acc upd*) (*TypAggregate fs*))
⟨*proof*⟩

lemma (**in** *wf-field*) *update-desc-component-desc-commutes*:
shows *update-desc acc upd* (*component-desc t*) =
component-desc (*map-td* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *t*)
⟨*proof*⟩

lemma (**in** *wf-field*) *wf-component-descs-map-td-update-desc*:
shows
wf-component-descs t
⟹ *wf-component-descs* (*map-td* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *t*)
wf-component-descs-struct st
⟹ *wf-component-descs-struct* (*map-td-struct* (λ*n algn*. *update-desc acc upd*)
(*update-desc acc upd*) *st*)
wf-component-descs-list fs
⟹ *wf-component-descs-list* (*map-td-list* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *fs*)
wf-component-descs-tuple f
⟹ *wf-component-descs-tuple* (*map-td-tuple* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *f*)
⟨*proof*⟩

lemma (**in** *wf-field*) *wf-component-descs-adjust-ti*:
assumes *wf-t*: *wf-component-descs t*
shows *wf-component-descs* (*adjust-ti t acc upd*)
⟨*proof*⟩

lemma (**in** *wf-xfield*) *update-desc-component-desc*:
shows
update-desc acc upd (*component-desc* (*typ-info-t TYPE*('*a*))) =
(|*field-access* = *xto-bytes* ∘ *acc*, *field-update* = *upd* ∘ *xfrom-bytes*,
field-sz = *size-of TYPE*('*a*)|)
⟨*proof*⟩

lemma (**in** *wf-xfield*) *wf-component-descs-ti-typ-combine*:
assumes *wf-t*: *wf-component-descs t*

shows *wf-component-descs* (*ti-typ-combine ft acc upd algn fn t*)
⟨*proof*⟩

lemma (**in** *wf-xfield*) *wf-component-descs-ti-typ-pad-combine*:
assumes *wf-t*: *wf-component-descs t*
shows *wf-component-descs* (*ti-typ-pad-combine ft acc upd algn fn t*)
⟨*proof*⟩

lemma *component-descs-independent-empty-typ-info* [*simp*]:
component-descs-independent (*empty-typ-info algn n*)
⟨*proof*⟩

lemma *component-descs-independent-list-appendI*:
assumes *ind-xs-ys*: ∀ *x* ∈ *set* (*toplevel-field-descs-list xs*).
field-desc-independent (*field-access x*) (*field-update x*) (*set* (*toplevel-field-descs-list ys*))
assumes *ind-xs*: *component-descs-independent-list xs*
assumes *ind-ys*: *component-descs-independent-list ys*
shows *component-descs-independent-list* (*xs* @ *ys*)
⟨*proof*⟩

lemma *component-descs-independent-extend-ti*:
assumes *ind-t*: *component-descs-independent t*
assumes *ind-ft*: *component-descs-independent ft*
assumes *ind-d-t*: *field-desc-independent* (*field-access d*) (*field-update d*) (*set* (*toplevel-field-descs t*))
shows *component-descs-independent* (*extend-ti t ft algn fn d*)
⟨*proof*⟩

lemma *fu-commutes-id1* [*simp*]: *fu-commutes* (λ*bs*. *id*) *upd*
⟨*proof*⟩

lemma *fu-commutes-id2* [*simp*]: *fu-commutes upd* (λ*bs*. *id*)
⟨*proof*⟩

lemma *field-desc-independent-pad*: *field-desc-independent* (λ*v bs*. *if n* ≤ *length bs then take n bs else replicate n 0*) (λ*bs*. *id*) *D*
⟨*proof*⟩

lemma *component-descs-independent-ti-pad-combine*:
assumes *ind-t*: *component-descs-independent t*
shows *component-descs-independent* (*ti-pad-combine n t*)
⟨*proof*⟩

lemma (**in** *wf-field*) *field-desc-independent-update-desc*:
assumes *ind*: *field-desc-independent* (*field-access d*) (*field-update d*) *D*
shows *field-desc-independent* (*field-access* (*update-desc acc upd d*)) (*field-update* (*update-desc acc upd d*))
(*update-desc acc upd ‘ D*) (**is** *field-desc-independent ?acc ?upd ?U*)
⟨*proof*⟩

lemma (**in** *wf-field*) *field-descs-independent-update-desc*:
field-descs-independent (*toplevel-field-descs t*)
⟹ *field-descs-independent* (*toplevel-field-descs*
(*map-td* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) *t*))
field-descs-independent (*toplevel-field-descs-struct st*)
⟹ *field-descs-independent* (*toplevel-field-descs-struct*
(*map-td-struct* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) *st*))
field-descs-independent (*toplevel-field-descs-list fs*)
⟹ *field-descs-independent* (*toplevel-field-descs-list*
(*map-td-list* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) *fs*))
field-descs-independent (*toplevel-field-descs-tuple f*)
⟹ *field-descs-independent* (*toplevel-field-descs-tuple*
(*map-td-tuple* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) *f*))
⟨*proof*⟩

lemma (**in** *wf-field*) *component-descs-independent-update-desc*:
component-descs-independent t
⟹ *component-descs-independent*
(*map-td* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) *t*)
component-descs-independent-struct st
⟹ *component-descs-independent-struct*
(*map-td-struct* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) *st*)
component-descs-independent-list fs
⟹ *component-descs-independent-list*
(*map-td-list* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) *fs*)
component-descs-independent-tuple f
⟹ *component-descs-independent-tuple*
(*map-td-tuple* (λ*n algn. update-desc acc upd*) (*update-desc acc upd*) *f*)
⟨*proof*⟩

lemma (**in** *wf-field*) *component-descs-independent-adjust-ti*:
assumes *ind-t*: *component-descs-independent t*
shows *component-descs-independent* (*adjust-ti t acc upd*)
⟨*proof*⟩

theorem (**in** *wf-xfield*) *component-descs-independent-ti-typ-combine*:
fixes
ft::*'a*::*xmem-type itself* **and**
t:: *'s xtyp-info*
assumes *ind-t*: *component-descs-independent t*
assumes *ind-acc-upd*: *field-desc-independent* (*xto-bytes* ∘ *acc*) (*upd* ∘ *xfrom-bytes*)
(*set* (*toplevel-field-descs t*))
shows *component-descs-independent* (*ti-typ-combine ft acc upd algn fn t*)
⟨*proof*⟩

lemma (**in** *wf-xfield*) *wf-field-desc-extend-ti*:
assumes *ind-t*: *field-desc-independent* (*xto-bytes* ∘ *acc*) (*upd* ∘ *xfrom-bytes*) (*set* (*toplevel-field-descs t*))
assumes *ind-d*: *field-desc-independent* (*xto-bytes* ∘ *acc*) (*upd* ∘ *xfrom-bytes*) {*d*}
shows
field-desc-independent (*xto-bytes* ∘ *acc*) (*upd* ∘ *xfrom-bytes*)
(*set* (*toplevel-field-descs* (*extend-ti t ft algn fn d*)))
⟨*proof*⟩

lemma (**in** *wf-xfield*) *field-desc-independent-ti-pad-combine*:
assumes *ind-acc-upd*: *field-desc-independent* (*xto-bytes* ∘ *acc*) (*upd* ∘ *xfrom-bytes*)
(*set* (*toplevel-field-descs t*))
shows *field-desc-independent* (*xto-bytes* ∘ *acc*) (*upd* ∘ *xfrom-bytes*)
(*set* (*toplevel-field-descs* (*ti-pad-combine n t*)))
⟨*proof*⟩

theorem (**in** *wf-xfield*) *component-desc-independent-ti-typ-pad-combine*:
fixes
ft::*'a*::*xmem-type itself* **and**
t:: *'s xtyp-info*
assumes *ind-t*: *component-descs-independent t*
assumes *ind-acc-upd*: *field-desc-independent* (*xto-bytes* ∘ *acc*) (*upd* ∘ *xfrom-bytes*)
(*set* (*toplevel-field-descs t*))
shows *component-descs-independent* (*ti-typ-pad-combine ft acc upd algn fn t*)

⟨*proof*⟩

lemma *component-descs-independent-map-align*[*simp*]:
component-descs-independent (*map-align f t*) = *component-descs-independent t*
⟨*proof*⟩

lemma *component-descs-independent-final-pad*:
assumes *ind-t*: *component-descs-independent t*
shows *component-descs-independent* (*final-pad algn t*)
⟨*proof*⟩

theorem (**in** *xmem-contained-type*) *contained-field-descs-ti-typ-combine*:
fixes
ft::*'a itself* **and**
t:: *'b xtyp-info*
assumes *contained-t*: *contained-field-descs t*
shows *contained-field-descs* (*ti-typ-combine ft acc upd algn fn t*)
⟨*proof*⟩

theorem (**in** *xmem-contained-type*) *contained-field-descs-ti-typ-pad-combine*:
fixes
ft::*'a itself* **and**
t:: *'b xtyp-info*
assumes *contained-t*: *contained-field-descs t*
shows *contained-field-descs* (*ti-typ-pad-combine ft acc upd algn fn t*)
⟨*proof*⟩

locale *ti-ind'* =
fixes *X* :: *'a leaf-desc set*
fixes *Y* :: *'a leaf-desc set*
assumes *fu-commutes*: $x \in X \Longrightarrow y \in Y \Longrightarrow$ *fu-commutes* (*field-update* (*lf-fd x*)) (*field-update* (*lf-fd y*))
assumes *fa-fu-ind-X*: $x \in X \Longrightarrow y \in Y \Longrightarrow$ *fa-fu-ind* (*lf-fd x*) (*lf-fd y*) (*lf-sz y*) (*lf-sz x*)
assumes *fa-fu-ind-Y*: $x \in X \Longrightarrow y \in Y \Longrightarrow$ *fa-fu-ind* (*lf-fd y*) (*lf-fd x*) (*lf-sz x*) (*lf-sz y*)

lemma *ti-ind'-ti-ind*: *ti-ind' X Y* = *ti-ind X Y*
⟨*proof*⟩

lemma *fields-contained-transitive*:
assumes *d-in*: $d \in D$
assumes *d-contains* : *fields-contained* (*field-access d*) (*field-update d*) *X*
assumes *contained*: *fields-contained acc upd D*
shows *fields-contained acc upd X*
⟨*proof*⟩

lemma *fields-contained-singleton* [*simp*]:
fields-contained (*field-access d*) (*field-update d*) {*d*}
⟨*proof*⟩

lemma *contained-field-descs-leaf*:
contained-field-descs t $\Longrightarrow$ *ld* $\in$ *lf-fd* ‘ (*lf-set t* []) $\Longrightarrow$
$\exists\, d \in$ *set* (*toplevel-field-descs t*). *fields-contained* (*field-access d*) (*field-update d*) {*ld*}

contained-field-descs-struct st $\Longrightarrow$ *ld* $\in$ *lf-fd* ‘ (*lf-set-struct st* []) $\Longrightarrow$
$\exists$ *d* $\in$ *set* (*toplevel-field-descs-struct st*). *fields-contained* (*field-access d*) (*field-update d*) {*ld*}
contained-field-descs-list fs $\Longrightarrow$ *ld* $\in$ *lf-fd* ‘ (*lf-set-list fs* []) $\Longrightarrow$
$\exists$ *d* $\in$ *set* (*toplevel-field-descs-list fs*). *fields-contained* (*field-access d*) (*field-update d*) {*ld*}
contained-field-descs-tuple f $\Longrightarrow$ *ld* $\in$ *lf-fd* ‘ (*lf-set-tuple f* []) $\Longrightarrow$
$\exists$ *d* $\in$ *set* (*toplevel-field-descs-tuple f*). *fields-contained* (*field-access d*) (*field-update d*) {*ld*}
⟨*proof*⟩

lemma *wf-field-desc-wf-lf*:
wf-field-desc d $\Longrightarrow$ *sz* = *field-sz d* $\Longrightarrow$ *wf-lf* {(| *lf-fd* = *d*, *lf-sz* = *sz*, *lf-fn* = *xs* |)}
⟨*proof*⟩

lemma *wf-lf-unionI*: *wf-lf A* $\Longrightarrow$ *wf-lf B* $\Longrightarrow$ *ti-ind′ A B* $\Longrightarrow$ *wf-lf* (*A* $\cup$ *B*)
⟨*proof*⟩

lemma *fields-contained-subset*:
assumes *cont-X*: *fields-contained acc upd X*
assumes *subs*: *Y* $\subseteq$ *X*
shows *fields-contained acc upd Y*
⟨*proof*⟩

lemma *fields-contained-unionD1*:
assumes *fields-contained acc upd* (*X* $\cup$ *Y*)
shows *fields-contained acc upd X*
⟨*proof*⟩

lemma *fields-contained-unionD2*:
assumes *fields-contained acc upd* (*X* $\cup$ *Y*)
shows *fields-contained acc upd Y*
⟨*proof*⟩

lemma *fields-contained-unionI*:
assumes *cont-X*: *fields-contained acc upd X*
assumes *cont-Y*: *fields-contained acc upd Y*
shows *fields-contained acc upd* (*X* $\cup$ *Y*)
⟨*proof*⟩

lemma *fields-contained-insertI*:

assumes *cont-x*: *fields-contained acc upd* $\{x\}$
assumes *cont-Y*: *fields-contained acc upd Y*
shows *fields-contained acc upd* (*insert x Y*)
⟨*proof*⟩

lemma *fields-contained-toplevel-to-field-descs*:
$\bigwedge$(*acc*::'*a* ⇒ *byte list* ⇒ *byte list*) *upd*.
contained-field-descs t ⟹ *fields-contained acc upd* (*set* (*toplevel-field-descs t*))
⟹ *fields-contained acc upd* (*set* (*field-descs t*))
$\bigwedge$(*acc*::'*a* ⇒ *byte list* ⇒ *byte list*) *upd*.
contained-field-descs-struct st ⟹ *fields-contained acc upd* (*set* (*toplevel-field-descs-struct st*))
⟹ *fields-contained acc upd* (*set* (*field-descs-struct st*))
$\bigwedge$(*acc*::'*a* ⇒ *byte list* ⇒ *byte list*) *upd*.
contained-field-descs-list fs ⟹ *fields-contained acc upd* (*set* (*toplevel-field-descs-list fs*))
⟹ *fields-contained acc upd* (*set* (*field-descs-list fs*))
$\bigwedge$(*acc*::'*a* ⇒ *byte list* ⇒ *byte list*) *upd*.
contained-field-descs-tuple f ⟹ *fields-contained acc upd* (*set* (*toplevel-field-descs-tuple f*))
⟹ *fields-contained acc upd* (*set* (*field-descs-tuple f*))
⟨*proof*⟩

lemma *fu-commutes-intro*:
assumes $\bigwedge$*v bs bs'*. *f bs* (*g bs' v*) = *g bs'* (*f bs v*)
shows *fu-commutes f g*
⟨*proof*⟩

lemma (**in** *wf-field-desc*) *access-same-update-id*:
assumes *field-access d* (*field-update d bs v*) *bs'* = *field-access d v bs'*
shows *field-update d bs v* = *v*
⟨*proof*⟩

lemma (**in** *wf-field-desc*) *access-same-update-id'*:
assumes *upd-inv*: *field-update d bs v* = *v*
assumes *acc-same*: *field-access d v bs* = *field-access d v' bs*
shows *field-update d bs v'* = *v'*
⟨*proof*⟩

lemma (**in** *wf-field-desc*) *update-access-unequal*:
assumes *neq-upd*: *field-update d bs v* ≠ *v*
shows *field-access d v bs'* ≠ *field-access d* (*field-update d bs v*) *bs'*
⟨*proof*⟩

lemma (**in** *wf-field-desc*) *access-eq-update-eq*:
assumes $\bigwedge bs'$. *field-access d* (*field-update d bs v*) *bs'* = *field-access d v bs'*
shows *field-update d bs v* = *v*
⟨*proof*⟩

lemma *field-desc-independent-contained-transitive*:
assumes *cont*: *fields-contained* (*field-access e*) (*field-update e*) *D*
assumes *ind*: *field-desc-independent* (*field-access d*) (*field-update d*) {*e*}
shows *field-desc-independent* (*field-access d*) (*field-update d*) *D*
⟨*proof*⟩

lemma *field-desc-independent-contained-toplevel-to-field-descs*:
assumes *cont-fs*: *contained-field-descs-list fs*
assumes *ind-tl*: *field-desc-independent* (*field-access d*) (*field-update d*)
(*set* (*toplevel-field-descs-list fs*))
shows *field-desc-independent* (*field-access d*) (*field-update d*)
(*set* (*field-descs-list fs*))
⟨*proof*⟩

theorem *component-descs-independent-contained-wf-lf*:
fixes *t*::'*a xtyp-info* **and**
st::'*a xtyp-info-struct* **and**
fs::'*a xtyp-info-tuple list* **and**
f::'*a xtyp-info-tuple*
shows
$\bigwedge ps$. ⟦*wf-desc t*; *wf-component-descs t*; *wf-field-descs* (*set* (*field-descs t*)); *component-descs-independent t*;
contained-field-descs t ⟧
⟹ *wf-lf* (*lf-set t ps*)
$\bigwedge ps$. ⟦*wf-desc-struct st*; *wf-component-descs-struct st*; *wf-field-descs* (*set* (*field-descs-struct st*));
component-descs-independent-struct st; *contained-field-descs-struct st*⟧
⟹ *wf-lf* (*lf-set-struct st ps*)
$\bigwedge ps$. ⟦*wf-desc-list fs*; *wf-component-descs-list fs*; *wf-field-descs* (*set* (*field-descs-list fs*));
component-descs-independent-list fs; *contained-field-descs-list fs*⟧
⟹ *wf-lf* (*lf-set-list fs ps*)
$\bigwedge ps$. ⟦*wf-desc-tuple f*; *wf-component-descs-tuple f*; *wf-field-descs* (*set* (*field-descs-tuple f*));
component-descs-independent-tuple f; *contained-field-descs-tuple f*⟧
⟹ *wf-lf* (*lf-set-tuple f ps*)
⟨*proof*⟩

definition *wf-align-field ti* ≡ *wf-align ti* ∧ *align-field ti*

lemma *wf-align-field-empty-typ-info*: *wf-align-field* (*empty-typ-info align n*)
⟨*proof*⟩

lemma (**in** *mem-type*) *wf-align-field-ti-typ-pad-combine*:
aggregate ti ⟹ *wf-align-field ti* ⟹
wf-align-field (*ti-typ-pad-combine* (*TYPE*('*a*)) *acc upd algn fn ti*)
⟨*proof*⟩

lemma (**in** *mem-type*) *wf-align-field-ti-typ-combine*:
aggregate ti ⟹ *wf-align-field ti* ⟹ *2* ^ *align-td* (*typ-info-t TYPE*('*a*)) *dvd size-td ti* ⟹
wf-align-field (*ti-typ-combine* (*TYPE*('*a*)) *acc upd algn fn ti*)
⟨*proof*⟩

lemma *wf-align-field-ti-pad-combine*: *aggregate ti* ⟹ *wf-align-field ti* ⟹ *wf-align-field* (*ti-pad-combine n ti*)
⟨*proof*⟩

lemma *wf-align-field-final-pad*: *aggregate ti* ⟹ *wf-align-field ti* ⟹ *wf-align-field* (*final-pad algn ti*)
⟨*proof*⟩

lemmas *wf-align-field-simps* =
wf-align-field-empty-typ-info
wf-align-field-ti-typ-pad-combine
wf-align-field-ti-typ-combine
wf-align-field-ti-pad-combine
wf-align-field-final-pad

The following theorem is used to prove that a new type is in class *xmem-contained-type*, for which we have constructed the extended type info with:

- *empty-typ-info*
- *ti-typ-pad-combine* (*ti-typ-combine*, *ti-pad-combine*)
- *final-pad*.

Note that the field-types are already in *xmem-contained-type*.

theorem *tuned-xmem-contained-type-class-intro*:
assumes *wf-desc*: *wf-desc* (*typ-info-t TYPE*('*a*))
assumes *wf-size-desc*: *wf-size-desc* (*typ-info-t TYPE*('*a*))
assumes *align-dvd*: *align-of TYPE*('*a*) *dvd size-of TYPE*('*a*)
assumes *wf-align-field*: *wf-align-field* (*typ-info-t TYPE*('*a*))
assumes *size*: *size-of TYPE*('*a*) < *addr-card*

assumes *entire-update*: $\bigwedge bs\ v\ w.\ length\ bs = size\text{-}of\ TYPE('a) \Longrightarrow field\text{-}update\ (component\text{-}desc\ (typ\text{-}info\text{-}t\ TYPE('a)))\ bs\ v = field\text{-}update\ (component\text{-}desc\ (typ\text{-}info\text{-}t\ TYPE('a)))\ bs\ w$
assumes *wf-component-descs*: *wf-component-descs* (*typ-info-t* $TYPE('a)$)
assumes *ind*: *component-descs-independent* (*typ-info-t* $TYPE('a)$)
assumes *wf-field-descs*: *wf-field-descs* (*set* (*field-descs* (*typ-info-t* $TYPE('a)$)))
assumes *contained-field-descs*: *contained-field-descs* (*typ-info-t* $TYPE('a)$)
assumes *wf-padding*: *wf-padding* (*typ-info-t* $TYPE('a)$)
shows $OFCLASS('a::c\text{-}type, xmem\text{-}contained\text{-}type\text{-}class)$
⟨*proof*⟩

lemma *field-sz-extend-ti*: *aggregate t* ⟹
field-sz (*component-desc* (*extend-ti t ft algn fn d*)) = *field-sz* (*component-desc t*) + *field-sz d*
⟨*proof*⟩

lemma *field-sz-empty-typ-info*: *field-sz* (*component-desc* (*empty-typ-info algn n*)) = *0*
⟨*proof*⟩

lemma (**in** *c-type*) *field-sz-ti-typ-combine*:
fixes *acc*:: *'s* ⇒ *'a*
assumes *agg*: *aggregate t*
shows *field-sz* (*component-desc* (*ti-typ-combine ft acc upd algn fn t*)) = *field-sz* (*component-desc t*) + *size-of* $TYPE('a)$
⟨*proof*⟩

lemma (**in** *c-type*) *field-sz-ti-pad-combine*:
fixes *acc*:: *'s* ⇒ *'a*
assumes *agg*: *aggregate t*
shows *field-sz* (*component-desc* (*ti-pad-combine n t*)) = *field-sz* (*component-desc t*) + *n*
⟨*proof*⟩

lemma (**in** *c-type*) *field-sz-ti-typ-pad-combine*:
fixes *acc*:: *'s* ⇒ *'a*
assumes *agg*: *aggregate t*
shows *field-sz* (*component-desc* (*ti-typ-pad-combine ft acc upd algn fn t*)) = *field-sz* (*component-desc t*) + *size-of* $TYPE('a)$ + *padup* (*max* (*2* ^ *algn*) (*align-of* $TYPE('a)$)) (*size-td t*)
⟨*proof*⟩

lemma *component-desc-map-align*[*simp*]:*component-desc* (*map-align f t*) = *component-desc t*
⟨*proof*⟩

lemma *field-sz-final-pad*:
assumes *agg*: *aggregate t*
shows *field-sz* (*component-desc* (*final-pad algn t*)) =

field-sz (*component-desc t*) + *padup* (*2^*(*max algn* (*align-td t*))) (*size-td t*)
⟨*proof*⟩

lemma *split-fold-append*: *split-fold f* (*xs@ys*) *bs s* =
(*let* (*s'*, *bs'*) = *split-fold f xs bs s in split-fold f ys bs' s'*)
⟨*proof*⟩

lemma *apply-field-updates-append*: *apply-field-updates* (*xs@ys*) *bs s* =
(*let* (*s'*, *bs'*) = *apply-field-updates xs bs s in apply-field-updates ys bs' s'*)
⟨*proof*⟩

lemma *snd-apply-field-updates*: *snd* (*apply-field-updates ds bs s*) = (*drop* (*foldl* (+)
0 (*map field-sz ds*)) *bs*)
⟨*proof*⟩

lemma *field-update-extend-ti*:
assumes *agg*: *aggregate t*
shows *field-update* (*component-desc* (*extend-ti t ft algn fn d*)) *bs v* =
field-update d ((*drop* (*field-sz* (*component-desc t*))) *bs*)
(*field-update* (*component-desc t*) *bs v*)
⟨*proof*⟩

lemma *field-update-empty-typ-info*: *field-update* (*component-desc* (*empty-typ-info*
algn n)) *bs s* = *s*
⟨*proof*⟩

lemma (**in** *c-type*) *field-update-ti-typ-combine*:
assumes *agg*: *aggregate t*
shows *field-update* (*component-desc* (*ti-typ-combine* (*ft*::'*a itself*) *acc upd algn fn*
t)) *bs v* =
(*upd o xfrom-bytes*) ((*drop* (*field-sz* (*component-desc t*))) *bs*)
(*field-update* (*component-desc t*) *bs v*)
⟨*proof*⟩

lemma *field-update-ti-pad-combine*:
assumes *agg*: *aggregate t*
shows *field-update* (*component-desc* (*ti-pad-combine n t*)) *bs v* =
field-update (*component-desc t*) *bs v*
⟨*proof*⟩

lemma (**in** *c-type*) *field-update-ti-typ-pad-combine*:
fixes *acc*:: '*s* ⇒ '*a*
assumes *agg*: *aggregate t*

shows *field-update* (*component-desc* (*ti-typ-pad-combine ft acc upd algn fn t*)) *bs v* =
(*upd o xfrom-bytes*) (*drop* (*field-sz* (*component-desc t*) + *padup* (*max* (2 ^ *algn*) (*align-of TYPE*('*a*))) (*size-td t*)) *bs*)
(*field-update* (*component-desc t*) *bs v*)
⟨*proof*⟩

lemma *field-update-final-pad*:
assumes *agg*: *aggregate t*
shows *field-update* (*component-desc* (*final-pad algn t*)) *bs v* =
field-update (*component-desc t*) *bs v*
⟨*proof*⟩

lemma *set-toplevel-field-descs-extend-ti*:
set (*toplevel-field-descs* (*extend-ti t ft algn fn d*)) ⊆ *insert d* (*set* (*toplevel-field-descs t*))
⟨*proof*⟩

lemma *set-toplevel-field-descs-extend-ti-aggregate*:
aggregate t ⟹ *set* (*toplevel-field-descs* (*extend-ti t ft algn fn d*)) = *insert d* (*set* (*toplevel-field-descs t*))
⟨*proof*⟩

lemma (**in** *c-type*) *set-toplevel-field-descs-ti-typ-combine*:
set (*toplevel-field-descs* (*ti-typ-combine ft* (*acc*::'*b* ⇒ '*a*) *upd algn fn t*)) ⊆
insert (⦇*field-access* = *xto-bytes* ∘ *acc*, *field-update* = *upd* ∘ *xfrom-bytes*, *field-sz* = *size-of TYPE*('*a*)⦈
(*set* (*toplevel-field-descs t*))
⟨*proof*⟩

lemma (**in** *c-type*) *set-toplevel-field-descs-ti-typ-combine-aggregate*:
fixes *acc*::'*s* ⇒ '*a*
assumes *agg*: *aggregate t*
shows *set* (*toplevel-field-descs* (*ti-typ-combine ft acc upd algn fn t*)) =
insert (⦇*field-access* = *xto-bytes* ∘ *acc*, *field-update* = *upd* ∘ *xfrom-bytes*, *field-sz* = *size-of TYPE*('*a*)⦈
(*set* (*toplevel-field-descs t*))
⟨*proof*⟩

lemma *set-field-descs-extend-ti-aggregate*:
aggregate t ⟹ *set* (*field-descs* (*extend-ti t ft algn fn d*)) = *insert d* (*set* (*field-descs ft*) ∪ *set* (*field-descs t*))
⟨*proof*⟩

lemma (**in** *c-type*) *set-field-descs-ti-typ-combine-aggregate*:
fixes *acc*::'*s* ⇒ '*a*
assumes *agg*: *aggregate t*
shows *set* (*field-descs* (*ti-typ-combine ft acc upd algn fn t*)) =
insert (⦇*field-access* = *xto-bytes* ∘ *acc*, *field-update* = *upd* ∘ *xfrom-bytes*,

field-sz = size-of TYPE('a)⟫
(*set* (*field-descs* (*adjust-ti* (*typ-info-t TYPE('a)*) *acc upd*)) ∪ *set* (*field-descs t*))
⟨*proof*⟩

locale *padding*=
fixes *d*::'*a field-desc*
assumes *independent*: *field-desc-independent acc upd* {*d*}

lemma *pad-is-padding*: *padding* (*padding-desc n*)
⟨*proof*⟩

definition *PAD*::'*a field-desc set* **where** *PAD* = {*d*. *padding d*}

lemma *in-PAD-iff*[*iff*]: *d* ∈ *PAD* ⟷ (*padding d*) ⟨*proof*⟩

lemma (**in** *c-type*) *set-toplevel-field-descs-ti-pad-combine-aggregate*:
fixes *acc*::'*s* ⇒ '*a*
assumes *agg*: *aggregate t*
shows *set* (*toplevel-field-descs* (*ti-pad-combine n t*)) ∪ *PAD* =
set (*toplevel-field-descs t*) ∪ *PAD*
⟨*proof*⟩

lemma (**in** *c-type*) *set-toplevel-field-descs-ti-typ-pad-combine-aggregate*:
fixes *acc*::'*s* ⇒ '*a*
assumes *agg*: *aggregate t*
shows *set* (*toplevel-field-descs* (*ti-typ-pad-combine ft acc upd algn fn t*)) ∪ *PAD* =
insert (⟪*field-access* = *xto-bytes* ∘ *acc*, *field-update* = *upd* ∘ *xfrom-bytes*, *field-sz* = *size-of TYPE('a)*⟫
((*set* (*toplevel-field-descs t*)) ∪ *PAD*)
⟨*proof*⟩

lemma *toplevel-field-descs-map-align*[*simp*]: *toplevel-field-descs* (*map-align f t*) = *toplevel-field-descs t*
⟨*proof*⟩

lemma *set-toplevel-field-descs-final-pad-aggregate*:
assumes *agg*: *aggregate t*
shows *set* (*toplevel-field-descs* (*final-pad algn t*)) ∪ *PAD* =
set (*toplevel-field-descs t*) ∪ *PAD*
⟨*proof*⟩

lemma *set-toplevel-field-descs-empty-typ-info*: *set* (*toplevel-field-descs* (*empty-typ-info algn n*)) = {}
⟨*proof*⟩

lemmas *set-toplevel-field-descs-combinator-simps* =
set-toplevel-field-descs-empty-typ-info
set-toplevel-field-descs-ti-typ-combine-aggregate
set-toplevel-field-descs-ti-typ-pad-combine-aggregate
set-toplevel-field-descs-final-pad-aggregate

lemma *field-descs-independent-PAD*:
field-desc-independent acc upd $(D \cup PAD)$ = *field-desc-independent acc upd D*
⟨*proof*⟩

lemma *field-desc-independent-PAD-expand*:
field-desc-independent acc upd $(D \cup PAD) \Longrightarrow$ *field-desc-independent acc upd D*
⟨*proof*⟩

lemma *field-desc-independent-PAD-collapse*:
field-desc-independent acc upd D $\Longrightarrow$ *field-desc-independent acc upd* $(D \cup PAD)$
⟨*proof*⟩

lemma *insert-union-out*: *insert d* $(X \cup Y)$ = *insert d X* $\cup$ *Y*
⟨*proof*⟩

lemmas *field-sz-typ-combinators-simps* =
field-sz-final-pad
field-sz-ti-typ-pad-combine
field-sz-ti-typ-combine
field-sz-empty-typ-info

lemmas *field-update-typ-combinators-simps* =
field-update-final-pad
field-update-ti-typ-pad-combine
field-update-ti-typ-combine
field-update-empty-typ-info

lemma *aggregate-map-align*[*simp*]: *aggregate* (*map-align f t*) = *aggregate t*
⟨*proof*⟩

lemma *aggregate-final-pad*[*simp*]: *aggregate t* $\Longrightarrow$ *aggregate* (*final-pad algn t*)
⟨*proof*⟩

lemmas *aggregate-typ-combinators-simps* =
aggregate-empty-typ-info
aggregate-ti-pad-combine
aggregate-ti-typ-pad-combine
aggregate-final-pad

lemma *wf-padding-empty-typ-info*: *wf-padding* (*empty-typ-info algn tn*)
⟨*proof*⟩

lemma *is-padding-tag-padding-tag*[*simp*]: *is-padding-tag* (*padding-tag n*)
⟨*proof*⟩

lemma *is-padding-tag-pad-typ*[*simp*]: *is-padding-tag* (*TypDesc 0* (*TypScalar n 0* (*padding-desc n*)) ''!*pad-typ*'')
⟨*proof*⟩

lemma *wf-padding-padding-tag*: *wf-padding* (*padding-tag n*)
⟨*proof*⟩

lemma *wf-padding-tuple-padI*:
is-padding-tag s ⟹ *wf-padding-tuple* (*DTuple s* (*CHR* ''!''#*xs*) *d*)
⟨*proof*⟩

lemma *wf-padding-tuple-no-padI*:
¬ *padding-field-name fn* ⟹ *wf-padding s* ⟹ *wf-padding-tuple* (*DTuple s fn d*)
⟨*proof*⟩

lemma *wf-padding-extend-ti*:
fixes
t :: 'a *xtyp-info* **and**
st :: 'a *xtyp-info-struct* **and**
ts :: 'a *xtyp-info-tuple list* **and**
x :: 'a *xtyp-info-tuple*
shows
wf-padding t ⟹ *wf-padding-tuple* (*DTuple s fn d*) ⟹
wf-padding (*extend-ti t s n fn d*)

wf-padding-struct st ⟹ *wf-padding-tuple* (*DTuple s fn d*) ⟹
wf-padding-struct (*extend-ti-struct st s fn d*)

wf-padding-list ts ⟹ *wf-padding-tuple* (*DTuple s fn d*) ⟹
wf-padding-list (*ts* @ [*DTuple s fn d*])

wf-padding-tuple x ⟹ *wf-padding-tuple x*
⟨*proof*⟩

lemma *is-padding-tag-update-desc*: *is-padding-tag t* ⟹
(⋀*x*. *upd* (*acc x*) *x* = *x*) ⟹
is-padding-tag
(*map-td* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *t*)
⟨*proof*⟩

lemma *wf-padding-adjust-ti*:
fixes

$t :: 'a$ *xtyp-info* **and**
$st :: 'a$ *xtyp-info-struct* **and**
$ts :: 'a$ *xtyp-info-tuple list* **and**
$x :: 'a$ *xtyp-info-tuple*
assumes *upd-acc-id*: $(\bigwedge x.\ upd\ (acc\ x)\ x = x)$
shows
wf-padding t $\Longrightarrow$
wf-padding (*map-td* ($\lambda n\ algn.$ *update-desc acc upd*) (*update-desc acc upd*) *t*)

wf-padding-struct st $\Longrightarrow$
wf-padding-struct (*map-td-struct* ($\lambda n\ algn.$ *update-desc acc upd*) (*update-desc acc upd*) *st*)

wf-padding-list ts $\Longrightarrow$
wf-padding-list (*map-td-list* ($\lambda n\ algn.$ *update-desc acc upd*) (*update-desc acc upd*) *ts*)

wf-padding-tuple x $\Longrightarrow$
wf-padding-tuple (*map-td-tuple* ($\lambda n\ algn.$ *update-desc acc upd*) (*update-desc acc upd*) *x*)
⟨*proof*⟩

lemma *wf-padding-ti-pad-combine*: *wf-padding t* $\Longrightarrow$ *wf-padding* (*ti-pad-combine n t*)
⟨*proof*⟩

lemma (**in** *c-type*) *wf-padding-ti-typ-combine*:
wf-padding t $\Longrightarrow$ *wf-padding* (*typ-info-t TYPE*('*a*)) $\Longrightarrow$ $(\bigwedge xs.\ fn \neq CHR\ ''!''\#xs)$ $\Longrightarrow$ $(\bigwedge x.\ upd\ (acc\ x)\ x = x) \Longrightarrow$
wf-padding (*ti-typ-combine* (*TYPE*('*a*)) *acc upd algn fn t*)
⟨*proof*⟩

lemma (**in** *c-type*) *wf-padding-ti-typ-pad-combine*:
wf-padding t $\Longrightarrow$ *wf-padding* (*typ-info-t TYPE*('*a*)) $\Longrightarrow$ $(\bigwedge xs.\ fn \neq CHR\ ''!''\#xs)$ $\Longrightarrow$ $(\bigwedge x.\ upd\ (acc\ x)\ x = x) \Longrightarrow$
wf-padding (*ti-typ-pad-combine* (*TYPE*('*a*)) *acc upd algn fn t*)
⟨*proof*⟩

lemma *wf-padding-map-align*: *wf-padding t* $\Longrightarrow$ *wf-padding* (*map-align f t*)
⟨*proof*⟩

lemma *wf-padding-final-pad*: *wf-padding t* $\Longrightarrow$ *wf-padding* (*final-pad n t*)
⟨*proof*⟩

lemmas *wf-padding-combinator-simps* =
wf-padding-empty-typ-info
wf-padding-final-pad
wf-padding-ti-typ-pad-combine

wf-padding-ti-typ-combine

end

theory *ArraysMemInstance*
imports *Arrays CompoundCTypes*
begin

primrec (**in** *c-type*)
array-tag-n :: *nat* ⇒ (*'a*,*'b*::*finite*) *array xtyp-info*
where
atn-base:
array-tag-n 0 = ((*empty-typ-info* (*align-td* (*typ-uinfo-t TYPE*(*'a*))) (*typ-name* (*typ-uinfo-t TYPE*(*'a*)) @ *''-array-''* @
nat-to-bin-string (*CARD*(*'b*::*finite*))))::(*'a*,*'b*) *array xtyp-info*)
| *atn-rec*:
array-tag-n (*Suc n*) = ((*ti-typ-combine TYPE*(*'a*) (λ*x*. *index x n*) (λ*x f*. *update f n x*) *0* (*replicate n CHR ''1''*) (*array-tag-n n*))::(*'a*,*'b*::*finite*) *array xtyp-info*)

definition (**in** *c-type*) *array-tag* :: (*'a*,*'b*::*finite*) *array itself* ⇒ (*'a*,*'b*) *array xtyp-info*
where
array-tag t ≡ *array-tag-n* (*CARD*(*'b*))

lemma (**in** *c-type*) *typ-name-array-tag*: *typ-name* ((*array-tag-n n*)::(*'a*,*'b*::*finite*) *array xtyp-info*) =
(*typ-name* (*typ-uinfo-t TYPE*(*'a*)) @ *''-array-''* @ *nat-to-bin-string* (*CARD*(*'b*)))
⟨*proof*⟩

instantiation *array* :: (*c-type*, *finite*) *c-type*
begin

definition *typ-info-array*:
typ-info-t (*w*::(*'a*::*c-type*,*'b*::*finite*) *array itself*) ≡ *array-tag w*

definition *typ-name-itself* (*w*::(*'a*::*c-type*,*'b*::*finite*) *array itself*) = *typ-name* (*typ-info-t w*)

instance ⟨*proof*⟩
end

lemma (**in** *c-type*) *field-names-array-tag-length* [*rule-format*]:
x ∈ *set* (*field-names-list* (*array-tag-n n*)) ⟶ *length x* < *n*
⟨*proof*⟩

lemma (**in** *c-type*) *replicate-mem-field-names-array-tag* [*simp*]:
replicate n x ∉ *set* (*field-names-list* (*array-tag-n n*))
⟨*proof*⟩

lemma (**in** *c-type*) *aggregate-array-tag* [*simp*]:
aggregate (*array-tag-n n*)
⟨*proof*⟩

lemma (**in** *mem-type*) *wf-desc-array-tag* [*simp*]:
wf-desc ((*array-tag-n n*)::('*a*,'*b*::*finite*) *array xtyp-info*)
⟨*proof*⟩

lemma (**in** *mem-type*) *wf-size-desc-array-tag* [*simp*]:
0 < *n* ⟹ *wf-size-desc* ((*array-tag-n n*)::('*a*,'*b*::*finite*) *array xtyp-info*)
⟨*proof*⟩

lemma (**in** *mem-type*) *upd-ind-array-tag-udpate* [*simp*]:
⟦ *n* ≤ *m*; *n* ≤ *CARD*('*b*) ⟧ ⟹
upd-ind (*lf-set* ((*array-tag-n n*)::('*a*,'*b*::*finite*) *array xtyp-info*) []) (λ*x f*. *update f m x*)
⟨*proof*⟩

lemma (**in** *mem-type*) *fc-array-tag-udpate* [*simp*]:
⟦ *n* ≤ *m*; *n* ≤ *CARD*('*b*) ⟧ ⟹
fu-commutes (*update-ti-t* ((*array-tag-n n*)::('*a*,'*b*::*finite*) *array xtyp-info*)) (λ*x f*. *update f m x*)
⟨*proof*⟩

lemma (**in** *mem-type*) *acc-ind-array-tag-udpate* [*simp*]:
⟦ *n* ≤ *m*; *m* < *CARD*('*b*) ⟧ ⟹
acc-ind (λ*x*. *index x m*) (*lf-fd* ‘ *lf-set* ((*array-tag-n n*)::('*a*,'*b*::*finite*) *array xtyp-info*) [])
⟨*proof*⟩

lemma (**in** *mem-type*) *fa-fu-g-array-tag-udpate* [*simp*]:
⟦ *n* ≤ *m*; *m* < *CARD*('*b*) ⟧ ⟹
fa-ind (*lf-fd* ‘ *lf-set* ((*array-tag-n n*)::('*a*,'*b*::*finite*) *array xtyp-info*) []) (λ*x f*. *update f m x*)
⟨*proof*⟩

lemma (**in** *mem-type*) *wf-fdp-array-tag* [*simp*]:
n ≤ *CARD*('*b*) ⟹ *wf-lf* (*lf-set* ((*array-tag-n n*)::('*a*,'*b*::*finite*) *array xtyp-info*) [])
⟨*proof*⟩

lemma *upd-local-update* [*simp*]:
upd-local (λ*x f*. *update f n x*)
⟨*proof*⟩

lemma (**in** *mem-type*) *fu-eq-mask-array-tag* [*simp*, *rule-format*]:
$n \leq CARD('b) \longrightarrow (\forall\, m.\ (\forall\, k\ v.\ k < CARD('b) \longrightarrow$
index $((m\ v)::('a,'b)\ array)\ k = (if\ n \leq k\ then$
index $(undefined::('a,'b::finite)\ array)\ k$
else index $v\ k)) \longrightarrow$ *fu-eq-mask* $(array\text{-}tag\text{-}n\ n)\ m)$
⟨*proof*⟩

lemma (**in** *c-type*) *size-td-array-tag* [*simp*]:
size-td $(((array\text{-}tag\text{-}n\ n)::('a,'b::finite)\ array\ xtyp\text{-}info)) =$
$n *$ *size-of* $TYPE('a)$
⟨*proof*⟩

lemma (**in** *c-type*) *align-td-wo-align-array-tag*:
$0 < n \Longrightarrow$
align-td-wo-align $((array\text{-}tag\text{-}n\ n)::('a,'b::finite)\ array\ xtyp\text{-}info) = (align\text{-}td\text{-}wo\text{-}align$
$(typ\text{-}info\text{-}t\ (TYPE('a))))$
⟨*proof*⟩

lemma *align-td-export-uinfo*[*simp*]: *align-td* $(export\text{-}uinfo\ t) = align\text{-}td\ t$
⟨*proof*⟩

lemma (**in** *c-type*) *align-td-uinfo*: *align-td* $(typ\text{-}uinfo\text{-}t\ (TYPE('a))) = align\text{-}td$
$(typ\text{-}info\text{-}t\ (TYPE('a)))$
⟨*proof*⟩

lemma (**in** *c-type*) *align-td-array-tag*:
$0 < n \Longrightarrow$
align-td $((array\text{-}tag\text{-}n\ n)::('a,'b::finite)\ array\ xtyp\text{-}info) = (align\text{-}td\ (typ\text{-}info\text{-}t$
$(TYPE('a))))$
⟨*proof*⟩

lemma *align-of-array* [*simp*]:
$0 < CARD('b) \Longrightarrow align\text{-}of\ TYPE(('a,'b::finite)\ array) = align\text{-}of\ TYPE('a::c\text{-}type)$
⟨*proof*⟩

lemma *align-td-wo-align-array-info*: $0 < CARD('b) \Longrightarrow align\text{-}td\text{-}wo\text{-}align\ (typ\text{-}info\text{-}t$
$TYPE(('a,'b::finite)\ array))$
$= align\text{-}td\text{-}wo\text{-}align\ (typ\text{-}info\text{-}t\ TYPE('a::c\text{-}type))$
⟨*proof*⟩

lemma *align-td-array-info*: $0 < CARD('b) \Longrightarrow align\text{-}td\ (typ\text{-}info\text{-}t\ TYPE(('a,'b::finite)$
$array))$
$= align\text{-}td\ (typ\text{-}info\text{-}t\ TYPE('a::c\text{-}type))$
⟨*proof*⟩

lemma (**in** *mem-type*) *align-field-array* [*simp*]:
align-field $((array\text{-}tag\text{-}n\ n)::('a,'b::finite)\ array\ xtyp\text{-}info)$
⟨*proof*⟩

lemma (**in** *mem-type*) *wf-align-array* [*simp*]:
wf-align ((*array-tag-n n*)::('*a*,'*b*::*finite*) *array xtyp-info*)
⟨*proof*⟩

instance *array* :: (*mem-type*,*finite*) *mem-type-sans-size*
⟨*proof*⟩

declare *atn-base* [*simp del*]
declare *atn-rec* [*simp del*]

lemma *size-of-array* [*simp*]:
size-of TYPE(('*a*,'*b*::*finite*) *array*) = *CARD*('*b*) ∗ *size-of TYPE*('*a*::*c-type*)
⟨*proof*⟩

lemma *size-td-array*:
size-td (*typ-info-t TYPE*(('*a*,'*b*::*finite*) *array*)) = *CARD*('*b*) ∗ *size-of TYPE*('*a*::*c-type*)
⟨*proof*⟩

lemma *align-td-array*:
2^align-td (*typ-info-t TYPE*(('*a*,'*b*::*finite*) *array*)) = *align-of TYPE*('*a*::*c-type*)
⟨*proof*⟩

lemma *wf-field-array*:
n < *CARD*('*b*) ⟹ *wf-field* (λ*x*. *x*.[*n*]) (λ*x f*. *update* (*f*::('*a*,'*b*::*finite*) *array*) *n x*)
⟨*proof*⟩

lemma *wf-cfield-array*:
n < *CARD*('*b*) ⟹ *wf-cfield* (λ*x*. *x*.[*n*]) (λ*x f*. *update* (*f*::('*a*::*c-type*,'*b*::*finite*) *array*) *n x*)
⟨*proof*⟩

lemma *wf-xfield-array*:
n < *CARD*('*b*) ⟹ *wf-xfield* (λ*x*. *x*.[*n*]) (λ*x f*. *update* (*f*::('*a*::*xmem-type*,'*b*::*finite*) *array*) *n x*)
⟨*proof*⟩

lemma *wf-component-descs-array-tag-n*: *n* ≤ *CARD*('*b*)
⟹ *wf-component-descs* ((*array-tag-n*::*nat* ⇒ ('*a*::*xmem-type*,'*b*::*finite*) *array xtyp-info*) *n*)
⟨*proof*⟩

lemma *wf-component-descs-array*: *wf-component-descs* (*typ-info-t TYPE*('*a*::*xmem-type*['*b*::*finite*]))
⟨*proof*⟩

lemma (**in** *c-type*) *set-toplevel-field-descs-array-tag-n*:
(*set* (*toplevel-field-descs* ((*array-tag-n*::*nat* ⇒ ('*a*,'*b*::*finite*) *array xtyp-info*) *n*)))
=
{*d*. ∃ *m*. *m* < *n* ∧ *d* = (|*field-access* = *xto-bytes* ∘ (λ*x*. *index x m*),
field-update = (λ*x f*. *update f m x*) ∘ *xfrom-bytes*,
field-sz = *size-of TYPE*('*a*)|)} (**is** - = *?D n*)
⟨*proof*⟩

lemma (**in** *xmem-type*) *field-desc-independent-extend-array*:
n < *CARD*('*b*) ⟹
field-desc-independent (*xto-bytes* ∘ (λ*x*. *x*.[*n*]))
((λ*x f*. *update* (*f*::('*a*,'*b*::*finite*) *array*) *n x*) ∘ *xfrom-bytes*)
(*set* (*toplevel-field-descs* (*array-tag-n n*)))
⟨*proof*⟩

lemma *component-descs-independent-array-tag-n*: *n* ≤ *CARD*('*b*)
⟹ *component-descs-independent* ((*array-tag-n*::*nat* ⇒ ('*a*::*xmem-type*,'*b*::*finite*)
array xtyp-info) *n*)
⟨*proof*⟩

lemma *component-descs-independent-array*: *component-descs-independent* (*typ-info-t*
TYPE('*a*::*xmem-type*['*b*::*finite*]))
⟨*proof*⟩

lemma *complement-padding-extend-array*: *n* < *CARD*('*b*) ⟹
complement-padding (*xto-bytes* ∘ (λ*x*. *x*.[*n*]))
((λ*x f*. *update* (*f*::('*a*::*xmem-type*,'*b*::*finite*) *array*) *n x*) ∘ *xfrom-bytes*) (*size-of*
TYPE('*a*))
⟨*proof*⟩

lemma *wf-field-desc-extend-array*: *n* < *CARD*('*b*) ⟹ *wf-field-desc*
(|*field-access* = *xto-bytes* ∘ (λ*x*. *x*.[*n*]),
field-update = (λ*x f*. *update* (*f*::('*a*::*xmem-type*,'*b*::*finite*) *array*) *n x*) ∘
xfrom-bytes,
field-sz = *size-of TYPE*('*a*::*xmem-type*)|)
⟨*proof*⟩

lemma (**in** *xmem-type*) *wf-field-desc-adjust-array-field*: *n* < *CARD*('*b*) ⟹
wf-field-descs
(*set* (*field-descs*
(*adjust-ti* (*typ-info-t TYPE*('*a*)) (λ*x*. *x*.[*n*])
(λ*x f*. *update* (*f*::('*a*,'*b*::*finite*) *array*) *n x*))))
⟨*proof*⟩

lemma *wf-field-descs-array-tag-n*: *n* ≤ *CARD*('*b*)
⟹ *wf-field-descs* (*set* (*field-descs* ((*array-tag-n*::*nat* ⇒ ('*a*::*xmem-type*,'*b*::*finite*)

array xtyp-info) *n*)))
⟨*proof*⟩

lemma *wf-field-descs-array*: *wf-field-descs* (*set* (*field-descs* (*typ-info-t TYPE*('*a*::*xmem-type*['*b*::*finite*])))))
⟨*proof*⟩

lemma (**in** *xmem-contained-type*) *contained-field-descs-array-tag-n*:
contained-field-descs ((*array-tag-n*::*nat* ⇒ ('*a*,'*b*::*finite*) *array xtyp-info*) *n*)
⟨*proof*⟩

lemma *contained-field-descs-array*: *contained-field-descs* (*typ-info-t TYPE*('*a*::*xmem-contained-type*['*b*::*finite*])
⟨*proof*⟩

lemma *replicate-1-neq-padding*: *replicate n CHR ''1'' ≠ CHR ''!'' # xs*
⟨*proof*⟩

lemma (**in** *xmem-type*) *wf-padding-array-tag-n*: *n* ≤ *CARD*('*b*)
⟹ *wf-padding* ((*array-tag-n*::*nat* ⇒ ('*a*,'*b*::*finite*) *array xtyp-info*) *n*)
⟨*proof*⟩

lemma *wf-padding-array*: *wf-padding* (*typ-info-t TYPE*('*a*::*xmem-type*['*b*::*finite*]))
⟨*proof*⟩

end

theory *ArchArraysMemInstance*
imports *ArraysMemInstance*
begin

class *array-outer-max-size* = *xmem-contained-type* +
assumes *array-outer-max-size-ax*: *size-of TYPE*('*a*) < *2* ^ *array-outer-max-size-exponent*

class *array-max-count* = *finite* +
assumes *array-max-count-ax*: *CARD* ('*a*) <= *2* ^ *array-outer-max-count-exponent*

instance *array* :: (*array-outer-max-size*, *array-max-count*) *mem-type*
⟨*proof*⟩

instance *array* :: (*array-outer-max-size*, *array-max-count*) *xmem-contained-type*
⟨*proof*⟩

class *array-inner-max-size* = *array-outer-max-size* +
 assumes *array-inner-max-size-ax*: *size-of TYPE*('*a*) < *2* ^ *array-inner-max-size-exponent*

instance *array* :: (*array-inner-max-size*, *array-max-count*) *array-outer-max-size*
⟨*proof*⟩

instance *word* :: (*len8*) *array-outer-max-size*
⟨*proof*⟩

instance *word* :: (*len8*) *array-inner-max-size*
⟨*proof*⟩

instance *ptr* :: (*c-type*) *array-outer-max-size*
⟨*proof*⟩

instance *ptr* :: (*c-type*) *array-inner-max-size*
⟨*proof*⟩

class *lt19* = *finite* +
  assumes *lt19-ax*: *CARD* ('*a*) < *2* ^ *19*
class *lt18* = *lt19* +
  assumes *lt18-ax*: *CARD* ('*a*) < *2* ^ *18*
class *lt17* = *lt18* +
  assumes *lt17-ax*: *CARD* ('*a*) < *2* ^ *17*
class *lt16* = *lt17* +
  assumes *lt16-ax*: *CARD* ('*a*) < *2* ^ *16*
class *lt15* = *lt16* +
  assumes *lt15-ax*: *CARD* ('*a*) < *2* ^ *15*
class *lt14* = *lt15* +
  assumes *lt14-ax*: *CARD* ('*a*) < *2* ^ *14*
class *lt13* = *lt14* +
  assumes *lt13-ax*: *CARD* ('*a*) < *2* ^ *13*
class *lt12* = *lt13* +
  assumes *lt12-ax*: *CARD* ('*a*) < *2* ^ *12*
class *lt11* = *lt12* +
  assumes *lt11-ax*: *CARD* ('*a*) < *2* ^ *11*
class *lt10* = *lt11* +
  assumes *lt10-ax*: *CARD* ('*a*) < *2* ^ *10*
class *lt9* = *lt10* +
  assumes *lt9-ax*: *CARD* ('*a*) < *2* ^ *9*
class *lt8* = *lt9* +
  assumes *lt8-ax*: *CARD* ('*a*) < *2* ^ *8*
class *lt7* = *lt8* +
  assumes *lt7-ax*: *CARD* ('*a*) < *2* ^ *7*
class *lt6* = *lt7* +
  assumes *lt6-ax*: *CARD* ('*a*) < *2* ^ *6*
class *lt5* = *lt6* +
  assumes *lt5-ax*: *CARD* ('*a*) < *2* ^ *5*
class *lt4* = *lt5* +

assumes *lt4-ax*: $CARD\ ('a) < 2\ \hat{}\ 4$
class *lt3* = *lt4* +
assumes *lt3-ax*: $CARD\ ('a) < 2\ \hat{}\ 3$
class *lt2* = *lt3* +
assumes *lt2-ax*: $CARD\ ('a) < 2\ \hat{}\ 2$
class *lt1* = *lt2* +
assumes *lt1-ax*: $CARD\ ('a) < 2\ \hat{}\ 1$

instance *bit0* :: (*lt19*) *array-max-count*
⟨*proof*⟩

instance *bit1* :: (*lt19*) *array-max-count*
⟨*proof*⟩

instance *bit0* :: (*lt18*) *lt19*
⟨*proof*⟩

instance *bit1* :: (*lt18*) *lt19*
⟨*proof*⟩

instance *bit0* :: (*lt17*) *lt18*
⟨*proof*⟩

instance *bit1* :: (*lt17*) *lt18*
⟨*proof*⟩

instance *bit0* :: (*lt16*) *lt17*
⟨*proof*⟩

instance *bit1* :: (*lt16*) *lt17*
⟨*proof*⟩

instance *bit0* :: (*lt15*) *lt16*
⟨*proof*⟩

instance *bit1* :: (*lt15*) *lt16*
⟨*proof*⟩

instance *bit0* :: (*lt14*) *lt15*
⟨*proof*⟩

instance *bit1* :: (*lt14*) *lt15*
⟨*proof*⟩

instance *bit0* :: (*lt13*) *lt14*
⟨*proof*⟩

instance *bit1* :: (*lt13*) *lt14*

⟨*proof*⟩

instance *bit0* :: (*lt12*) *lt13*
⟨*proof*⟩

instance *bit1* :: (*lt12*) *lt13*
⟨*proof*⟩

instance *bit0* :: (*lt12*) *array-max-count*
⟨*proof*⟩

instance *bit1* :: (*lt12*) *array-max-count*
⟨*proof*⟩

instance *bit0* :: (*lt11*) *lt12*
⟨*proof*⟩

instance *bit1* :: (*lt11*) *lt12*
⟨*proof*⟩

instance *bit0* :: (*lt10*) *lt11*
⟨*proof*⟩

instance *bit1* :: (*lt10*) *lt11*
⟨*proof*⟩

instance *bit0* :: (*lt9*) *lt10*
⟨*proof*⟩

instance *bit1* :: (*lt9*) *lt10*
⟨*proof*⟩

instance *bit0* :: (*lt8*) *lt9*
⟨*proof*⟩

instance *bit1* :: (*lt8*) *lt9*
⟨*proof*⟩

instance *bit0* :: (*lt7*) *lt8*
⟨*proof*⟩

instance *bit1* :: (*lt7*) *lt8*
⟨*proof*⟩

instance *bit0* :: (*lt6*) *lt7*
⟨*proof*⟩

instance *bit1* :: (*lt6*) *lt7*

⟨*proof*⟩

instance *bit0* :: (*lt5*) *lt6*
⟨*proof*⟩

instance *bit1* :: (*lt5*) *lt6*
⟨*proof*⟩

instance *bit0* :: (*lt4*) *lt5*
⟨*proof*⟩

instance *bit1* :: (*lt4*) *lt5*
⟨*proof*⟩

instance *bit0* :: (*lt3*) *lt4*
⟨*proof*⟩

instance *bit1* :: (*lt3*) *lt4*
⟨*proof*⟩

instance *bit0* :: (*lt2*) *lt3*
⟨*proof*⟩

instance *bit1* :: (*lt2*) *lt3*
⟨*proof*⟩

instance *bit0* :: (*lt1*) *lt2*
⟨*proof*⟩

instance *bit1* :: (*lt1*) *lt2*
⟨*proof*⟩

instance *num1* :: *lt1*
⟨*proof*⟩

instance *num1* :: *array-max-count*
⟨*proof*⟩

end

theory *HeapRawState*
imports *CTypes*
begin

type-synonym *typ-base* = *bool*
datatype *s-heap-index* = *SIndexVal* | *SIndexTyp nat*
datatype *s-heap-value* = *SValue byte* | *STyp typ-uinfo* × *typ-base*

primrec (*nonexhaustive*) *s-heap-tag* :: *s-heap-value* ⇒ *typ-uinfo* × *typ-base* **where**
s-heap-tag (*STyp t*) = *t*

type-synonym *typ-slice* = *nat* ⇀ *typ-uinfo* × *typ-base*

type-synonym *s-addr* = *addr* × *s-heap-index*
type-synonym *heap-state* = *s-addr* ⇀ *s-heap-value*
type-synonym *heap-typ-desc* = *addr* ⇒ *bool* × *typ-slice*
type-synonym *heap-raw-state* = *heap-mem* × *heap-typ-desc*

definition *hrs-mem* :: *heap-raw-state* ⇒ *heap-mem* **where**
hrs-mem ≡ *fst*

definition *hrs-mem-update* :: (*heap-mem* ⇒ *heap-mem*) ⇒ *heap-raw-state* ⇒ *heap-raw-state*
where
hrs-mem-update f ≡ λ(*h*,*d*). (*f h*,*d*)

definition *hrs-htd* :: *heap-raw-state* ⇒ *heap-typ-desc* **where**
hrs-htd ≡ *snd*

definition *hrs-htd-update* :: (*heap-typ-desc* ⇒ *heap-typ-desc*) ⇒ *heap-raw-state* ⇒
heap-raw-state
where
hrs-htd-update f ≡ λ(*h*,*d*). (*h*,*f d*)

lemma *hrs-comm*:
hrs-htd-update d (*hrs-mem-update h s*) = *hrs-mem-update h* (*hrs-htd-update d s*)
⟨*proof*⟩

lemma *hrs-htd-update-htd-update*:
(λ*s*. *hrs-htd-update d* (*hrs-htd-update d*′ *s*)) = *hrs-htd-update* (*d* ∘ *d*′)
⟨*proof*⟩

lemma *hrs-htd-mem-update* [*simp*]:
hrs-htd (*hrs-mem-update f s*) = *hrs-htd s*
⟨*proof*⟩

lemma *hrs-mem-htd-update* [*simp*]:
hrs-mem (*hrs-htd-update f s*) = *hrs-mem s*
⟨*proof*⟩

lemma *hrs-mem-update*:

hrs-mem (*hrs-mem-update f s*) = (*f* (*hrs-mem s*))
⟨*proof*⟩

lemma *hrs-htd-update*:
hrs-htd (*hrs-htd-update f s*) = (*f* (*hrs-htd s*))
⟨*proof*⟩

lemmas *hrs-update* = *hrs-mem-update hrs-htd-update*

lemma *hrs-htd-update-comp*: *hrs-htd-update f* ∘ *hrs-htd-update g* = *hrs-htd-update* (*f o g*)
⟨*proof*⟩

lemma *hrs-mem-update-comp*: *hrs-mem-update f* ∘ *hrs-mem-update g* = *hrs-mem-update* (*f o g*)
⟨*proof*⟩

lemma *hrs-update-commute*:
hrs-mem-update f ∘ *hrs-htd-update g* = *hrs-htd-update g o hrs-mem-update f*
⟨*proof*⟩

end

11.20 More properties of maps plus map disjuction.

theory *MapExtra*
imports *Main*
begin

BEWARE: we are not interested in using the *dom x* ∩ *dom y* = {} rules from Map for our separation logic proofs. As such, we overwrite the Map rules where that form of disjointness is in the assumption conflicts with a name we want to use with ⊥.

A note on naming: Anything not involving heap disjuction can potentially be incorporated directly into Map.thy, thus uses *m*. Anything involving heap disjunction is not really mergeable with Map, is destined for use in separation logic, and hence uses *h*

Things that should go into Option Type

Misc option lemmas

lemma *None-not-eq*: (*None* ≠ *x*) = (∃ *y*. *x* = *Some y*) ⟨*proof*⟩

lemma *None-com*: $(None = x) = (x = None)$ ⟨*proof*⟩

lemma *Some-com*: $(Some\ y = x) = (x = Some\ y)$ ⟨*proof*⟩

Things that should go into Map.thy

Map intersection: set of all keys for which the maps agree.

definition
map-inter :: $('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow 'a\ set$ (**infixl** ‹$\cap_m$› *70*) **where**
$m_1 \cap_m m_2 \equiv \{x \in dom\ m_1.\ m_1\ x = m_2\ x\}$

Map restriction via domain subtraction

definition
sub-restrict-map :: $('a \rightharpoonup 'b) => 'a\ set => ('a \rightharpoonup 'b)$ (**infixl** ‹‘–› *110*)
where
$m\ `-\ S \equiv (\lambda x.\ if\ x \in S\ then\ None\ else\ m\ x)$

11.20.1 Properties of maps not related to restriction

lemma *empty-forall-equiv*: $(m = Map.empty) = (\forall x.\ m\ x = None)$
⟨*proof*⟩

lemma *map-le-empty2* [*simp*]:
$(m \subseteq_m Map.empty) = (m = Map.empty)$
⟨*proof*⟩

lemma *dom-iff*:
$(\exists y.\ m\ x = Some\ y) = (x \in dom\ m)$
⟨*proof*⟩

lemma *non-dom-eval*:
$x \notin dom\ m \Longrightarrow m\ x = None$
⟨*proof*⟩

lemma *non-dom-eval-eq*:
$x \notin dom\ m = (m\ x = None)$
⟨*proof*⟩

lemma *map-add-same-left-eq*:
$m_1 = m_1' \Longrightarrow (m_0 ++ m_1 = m_0 ++ m_1')$
⟨*proof*⟩

lemma *map-add-left-cancelI* [*intro*!]:
$m_1 = m_1' \Longrightarrow m_0 ++ m_1 = m_0 ++ m_1'$
⟨*proof*⟩

lemma *dom-empty-is-empty*:

$(dom\ m = \{\}) = (m = Map.empty)$
⟨*proof*⟩

lemma *map-add-dom-eq*:
$dom\ m = dom\ m' \Longrightarrow m\ ++\ m' = m'$
⟨*proof*⟩

lemma *map-add-right-dom-eq*:
$[\![\ m_0\ ++\ m_1 = m_0'\ ++\ m_1';\ dom\ m_1 = dom\ m_1'\ ]\!] \Longrightarrow m_1 = m_1'$
⟨*proof*⟩

lemma *map-le-same-dom-eq*:
$[\![\ m_0 \subseteq_m m_1\ ;\ dom\ m_0 = dom\ m_1\ ]\!] \Longrightarrow m_0 = m_1$
⟨*proof*⟩

11.20.2 Properties of map restriction

lemma *restrict-map-cancel*:
$(m\ |`\ S = m\ |`\ T) = (dom\ m \cap S = dom\ m \cap T)$
⟨*proof*⟩

lemma *map-add-restricted-self* [*simp*]:
$m\ ++\ m\ |`\ S = m$
⟨*proof*⟩

lemma *map-add-restrict-dom-right* [*simp*]:
$(m\ ++\ m')\ |`\ dom\ m' = m'$
⟨*proof*⟩

lemma *restrict-map-UNIV* [*simp*]:
$m\ |`\ UNIV = m$
⟨*proof*⟩

lemma *restrict-map-dom*:
$S = dom\ m \Longrightarrow m\ |`\ S = m$
⟨*proof*⟩

lemma *restrict-map-subdom*:
$dom\ m \subseteq S \Longrightarrow m\ |`\ S = m$
⟨*proof*⟩

lemma *map-add-restrict*:
$(m_0\ ++\ m_1)\ |`\ S = ((m_0\ |`\ S)\ ++\ (m_1\ |`\ S))$
⟨*proof*⟩

lemma *map-le-restrict*:
$m \subseteq_m m' \Longrightarrow m = m'\ |`\ dom\ m$
⟨*proof*⟩

lemma *restrict-map-le*:
$m \mid` S \subseteq_m m$
⟨*proof*⟩

lemma *restrict-map-remerge*:
$[\![S \cap T = \{\}]\!] \Longrightarrow m \mid` S ++ m \mid` T = m \mid` (S \cup T)$
⟨*proof*⟩

lemma *restrict-map-empty*:
$dom\ m \cap S = \{\} \Longrightarrow m \mid` S = Map.empty$
⟨*proof*⟩

lemma *map-add-restrict-comp-right* [*simp*]:
$(m \mid` S ++ m \mid` (UNIV - S)) = m$
⟨*proof*⟩

lemma *map-add-restrict-comp-right-dom* [*simp*]:
$(m \mid` S ++ m \mid` (dom\ m - S)) = m$
⟨*proof*⟩

lemma *map-add-restrict-comp-left* [*simp*]:
$(m \mid` (UNIV - S) ++ m \mid` S) = m$
⟨*proof*⟩

lemma *restrict-self-UNIV*:
$m \mid` (dom\ m - S) = m \mid` (UNIV - S)$
⟨*proof*⟩

lemma *map-add-restrict-nonmember-right*:
$x \notin dom\ m' \Longrightarrow (m ++ m') \mid` \{x\} = m \mid` \{x\}$
⟨*proof*⟩

lemma *map-add-restrict-nonmember-left*:
$x \notin dom\ m \Longrightarrow (m ++ m') \mid` \{x\} = m' \mid` \{x\}$
⟨*proof*⟩

lemma *map-add-restrict-right*:
$x \subseteq dom\ m' \Longrightarrow (m ++ m') \mid` x = m' \mid` x$
⟨*proof*⟩

lemma *restrict-map-compose*:
$[\![S \cup T = dom\ m\ ;\ S \cap T = \{\}]\!] \Longrightarrow m \mid` S ++ m \mid` T = m$
⟨*proof*⟩

lemma *map-le-dom-subset-restrict*:
$[\![m' \subseteq_m m;\ dom\ m' \subseteq S]\!] \Longrightarrow m' \subseteq_m (m \mid` S)$
⟨*proof*⟩

lemma *map-le-dom-restrict-sub-add*:

$m' \subseteq_m m \Longrightarrow m \mid` (dom\ m - dom\ m') ++ m' = m$
⟨*proof*⟩

lemma *subset-map-restrict-sub-add*:
$T \subseteq S \Longrightarrow m \mid` (S - T) ++ m \mid` T = m \mid` S$
⟨*proof*⟩

lemma *restrict-map-sub-union*:
$m \mid` (dom\ m - (S \cup T)) = (m \mid` (dom\ m - T)) \mid` (dom\ m - S)$
⟨*proof*⟩

lemma *prod-restrict-map-add*:
$[\![S \cup T = U; S \cap T = \{\}]\!] \Longrightarrow m \mid` (X \times S) ++ m \mid` (X \times T) = m \mid` (X \times U)$
⟨*proof*⟩

Things that should NOT go into Map.thy

11.21 Definitions

Map disjuction

definition
$map\text{-}disj :: ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow bool$ (**infix** ‹⊥› *51*) **where**
$h_0 \perp h_1 \equiv dom\ h_0 \cap dom\ h_1 = \{\}$

declare *None-not-eq* [*simp*]

Heap monotonicity and the frame property

definition
$heap\text{-}mono :: (('a \rightharpoonup 'b) \Rightarrow 'c\ option) \Rightarrow bool$ **where**
$heap\text{-}mono\ f \equiv \forall h\ h'\ v.\ h \perp h' \wedge f\ h = Some\ v \longrightarrow f\ (h ++ h') = Some\ v$

lemma *heap-monoE*:
$[\![heap\text{-}mono\ f\ ; f\ h = Some\ v\ ; h \perp h']\!] \Longrightarrow f\ (h ++ h') = Some\ v$
⟨*proof*⟩

lemma *heap-mono-simp*:
$[\![heap\text{-}mono\ f\ ; f\ h = Some\ v\ ; h \perp h']\!] \Longrightarrow f\ (h ++ h') = f\ h$
⟨*proof*⟩

definition
$heap\text{-}frame :: (('a \rightharpoonup 'b) \Rightarrow 'c\ option) \Rightarrow bool$ **where**
$heap\text{-}frame\ f \equiv \forall h\ h'\ v.\ h \perp h' \wedge f\ (h ++ h') = Some\ v \longrightarrow (f\ h = Some\ v \vee f\ h = None)$

lemma *heap-frameE*:
⟦ *heap-frame f* ; *f* (*h* ++ *h'*) = *Some v* ; *h* ⊥ *h'* ⟧
⟹ *f h* = *Some v* ∨ *f h* = *None*
⟨*proof*⟩

11.22 Properties of (‘−)

lemma *restrict-map-sub-disj*: *h* |‘ *S* ⊥ *h* ‘− *S*
⟨*proof*⟩

lemma *restrict-map-sub-add*: *h* |‘ *S* ++ *h* ‘− *S* = *h*
⟨*proof*⟩

11.23 Properties of map disjunction

lemma *map-disj-empty-right* [*simp*]:
h ⊥ *Map.empty*
⟨*proof*⟩

lemma *map-disj-empty-left* [*simp*]:
Map.empty ⊥ *h*
⟨*proof*⟩

lemma *map-disj-com*:
h$_0$ ⊥ *h*$_1$ = *h*$_1$ ⊥ *h*$_0$
⟨*proof*⟩

lemma *map-disjD*:
h$_0$ ⊥ *h*$_1$ ⟹ *dom h*$_0$ ∩ *dom h*$_1$ = {}
⟨*proof*⟩

lemma *map-disjI*:
dom h$_0$ ∩ *dom h*$_1$ = {} ⟹ *h*$_0$ ⊥ *h*$_1$
⟨*proof*⟩

11.23.1 Map associativity-commutativity based on map disjuction

lemma *map-add-com*:
h$_0$ ⊥ *h*$_1$ ⟹ *h*$_0$ ++ *h*$_1$ = *h*$_1$ ++ *h*$_0$
⟨*proof*⟩

lemma *map-add-left-commute*:
h$_0$ ⊥ *h*$_1$ ⟹ *h*$_0$ ++ (*h*$_1$ ++ *h*$_2$) = *h*$_1$ ++ (*h*$_0$ ++ *h*$_2$)
⟨*proof*⟩

lemma *map-add-disj*:
h$_0$ ⊥ (*h*$_1$ ++ *h*$_2$) = (*h*$_0$ ⊥ *h*$_1$ ∧ *h*$_0$ ⊥ *h*$_2$)

⟨*proof*⟩

lemma *map-add-disj'*:
$(h_1 \mathbin{++} h_2) \perp h_0 = (h_1 \perp h_0 \wedge h_2 \perp h_0)$
⟨*proof*⟩

We redefine (++) associativity to bind to the right, which seems to be the more common case. Note that when a theory includes Map again, *map-add-assoc* will return to the simpset and will cause infinite loops if its symmetric counterpart is added (e.g. via *map-ac-simps*)

declare *map-add-assoc* [*simp del*]

Since the associativity-commutativity of (++) relies on map disjunction, we include some basic rules into the ac set.

lemmas *map-ac-simps* =
map-add-assoc[*symmetric*] *map-add-com map-disj-com*
map-add-left-commute map-add-disj map-add-disj'

11.23.2 Basic properties

lemma *map-disj-None-right*:
$[\![h_0 \perp h_1 ; x \in dom\ h_0]\!] \Longrightarrow h_1\ x = None$
⟨*proof*⟩

lemma *map-disj-None-left*:
$[\![h_0 \perp h_1 ; x \in dom\ h_1]\!] \Longrightarrow h_0\ x = None$
⟨*proof*⟩

lemma *map-disj-None-left'*:
$[\![h_0\ x = Some\ y ; h_1 \perp h_0]\!] \Longrightarrow h_1\ x = None$
⟨*proof*⟩

lemma *map-disj-None-right'*:
$[\![h_1\ x = Some\ y ; h_1 \perp h_0]\!] \Longrightarrow h_0\ x = None$
⟨*proof*⟩

lemma *map-disj-common*:
$[\![h_0 \perp h_1 ; h_0\ p = Some\ v ; h_1\ p = Some\ v']\!] \Longrightarrow False$
⟨*proof*⟩

11.23.3 Map disjunction and addition

lemma *map-add-eval-left*:
$[\![x \in dom\ h ; h \perp h']\!] \Longrightarrow (h \mathbin{++} h')\ x = h\ x$
⟨*proof*⟩

lemma *map-add-eval-right*:
$[\![x \in dom\ h' ; h \perp h']\!] \Longrightarrow (h \mathbin{++} h')\ x = h'\ x$
⟨*proof*⟩

lemma *map-add-eval-left'*:
⟦ $x \notin dom\ h'$; $h \perp h'$ ⟧ ⟹ $(h +\!\!+ h')\ x = h\ x$
⟨*proof*⟩

lemma *map-add-eval-right'*:
⟦ $x \notin dom\ h$; $h \perp h'$ ⟧ ⟹ $(h +\!\!+ h')\ x = h'\ x$
⟨*proof*⟩

lemma *map-add-left-dom-eq*:
assumes *eq*: $h_0 +\!\!+ h_1 = h_0' +\!\!+ h_1'$
assumes *etc*: $h_0 \perp h_1\ h_0' \perp h_1'\ dom\ h_0 = dom\ h_0'$
shows $h_0 = h_0'$
⟨*proof*⟩

lemma *map-add-left-eq*:
assumes *eq*: $h_0 +\!\!+ h = h_1 +\!\!+ h$
assumes *disj*: $h_0 \perp h\ h_1 \perp h$
shows $h_0 = h_1$
⟨*proof*⟩

lemma *map-add-right-eq*:
⟦$h +\!\!+ h_0 = h +\!\!+ h_1$; $h_0 \perp h$; $h_1 \perp h$⟧ ⟹ $h_0 = h_1$
⟨*proof*⟩

lemma *map-disj-add-eq-dom-right-eq*:
assumes *merge*: $h_0 +\!\!+ h_1 = h_0' +\!\!+ h_1'$ **and** *d*: $dom\ h_0 = dom\ h_0'$ **and**
ab-disj: $h_0 \perp h_1$ **and** *cd-disj*: $h_0' \perp h_1'$
shows $h_1 = h_1'$
⟨*proof*⟩

lemma *map-disj-add-eq-dom-left-eq*:
assumes *add*: $h_0 +\!\!+ h_1 = h_0' +\!\!+ h_1'$ **and**
dom: $dom\ h_1 = dom\ h_1'$ **and**
disj: $h_0 \perp h_1\ h_0' \perp h_1'$
shows $h_0 = h_0'$
⟨*proof*⟩

lemma *map-add-left-cancel*:
assumes *disj*: $h_0 \perp h_1\ h_0 \perp h_1'$
shows $(h_0 +\!\!+ h_1 = h_0 +\!\!+ h_1') = (h_1 = h_1')$
⟨*proof*⟩

lemma *map-add-lr-disj*:
⟦ $h_0 +\!\!+ h_1 = h_0' +\!\!+ h_1'$; $h_1 \perp h_1'$ ⟧ ⟹ $dom\ h_1 \subseteq dom\ h_0'$
⟨*proof*⟩

11.23.4 Map disjunction and updates

lemma *map-disj-update-left* [*simp*]:
$p \in dom\ h_1 \Longrightarrow h_0 \perp h_1(p \mapsto v) = h_0 \perp h_1$
⟨*proof*⟩

lemma *map-disj-update-right* [*simp*]:
$p \in dom\ h_1 \Longrightarrow h_1(p \mapsto v) \perp h_0 = h_1 \perp h_0$
⟨*proof*⟩

lemma *map-add-update-left*:
$[\![\ h_0 \perp h_1\ ;\ p \in dom\ h_0\ ]\!] \Longrightarrow (h_0 +\!\!+ h_1)(p \mapsto v) = (h_0(p \mapsto v) +\!\!+ h_1)$
⟨*proof*⟩

lemma *map-add-update-right*:
$[\![\ h_0 \perp h_1\ ;\ p \in dom\ h_1\ ]\!] \Longrightarrow (h_0 +\!\!+ h_1)(p \mapsto v) = (h_0 +\!\!+ h_1\ (p \mapsto v))$
⟨*proof*⟩

lemma *map-add3-update*:
$[\![\ h_0 \perp h_1\ ;\ h_1\ \perp h_2\ ;\ h_0 \perp h_2\ ;\ p \in dom\ h_0\ ]\!]$
$\Longrightarrow (h_0 +\!\!+ h_1 +\!\!+ h_2)(p \mapsto v) = h_0(p \mapsto v) +\!\!+ h_1 +\!\!+ h_2$
⟨*proof*⟩

11.23.5 Map disjunction and $(\subseteq_m)$

lemma *map-le-override* [*simp*]:
$[\![\ h \perp h'\ ]\!] \Longrightarrow h \subseteq_m h +\!\!+ h'$
⟨*proof*⟩

lemma *map-leI-left*:
$[\![\ h = h_0 +\!\!+ h_1\ ;\ h_0 \perp h_1\ ]\!] \Longrightarrow h_0 \subseteq_m h$ ⟨*proof*⟩

lemma *map-leI-right*:
$[\![\ h = h_0 +\!\!+ h_1\ ;\ h_0 \perp h_1\ ]\!] \Longrightarrow h_1 \subseteq_m h$ ⟨*proof*⟩

lemma *map-disj-map-le*:
$[\![\ h_0{}' \subseteq_m h_0;\ h_0 \perp h_1\ ]\!] \Longrightarrow h_0{}' \perp h_1$
⟨*proof*⟩

lemma *map-le-on-disj-left*:
$[\![\ h' \subseteq_m h\ ;\ h_0 \perp h_1\ ;\ h' = h_0 +\!\!+ h_1\ ]\!] \Longrightarrow h_0 \subseteq_m h$
⟨*proof*⟩

lemma *map-le-on-disj-right*:
$[\![\ h' \subseteq_m h\ ;\ h_0 \perp h_1\ ;\ h' = h_1 +\!\!+ h_0\ ]\!] \Longrightarrow h_0 \subseteq_m h$
⟨*proof*⟩

lemma *map-le-add-cancel*:
$[\![\ h_0 \perp h_1\ ;\ h_0{}' \subseteq_m h_0\ ]\!] \Longrightarrow h_0{}' +\!\!+ h_1 \subseteq_m h_0 +\!\!+ h_1$
⟨*proof*⟩

lemma *map-le-override-bothD*:
assumes *subm*: $h_0' ++ h_1 \subseteq_m h_0 ++ h_1$
assumes *disj'*: $h_0' \perp h_1$
assumes *disj*: $h_0 \perp h_1$
shows $h_0' \subseteq_m h_0$
⟨*proof*⟩

lemma *map-le-conv*:
$(h_0' \subseteq_m h_0 \wedge h_0' \neq h_0) = (\exists h_1.\ h_0 = h_0' ++ h_1 \wedge h_0' \perp h_1 \wedge h_0' \neq h_0)$
⟨*proof*⟩

lemma *map-le-conv2*:
$h_0' \subseteq_m h_0 = (\exists h_1.\ h_0 = h_0' ++ h_1 \wedge h_0' \perp h_1)$
⟨*proof*⟩

11.23.6 Map disjunction and restriction

lemma *map-disj-comp* [*simp*]:
$h_0 \perp h_1 \mid` (UNIV - dom\ h_0)$
⟨*proof*⟩

lemma *restrict-map-disj*:
$S \cap T = \{\} \Longrightarrow h \mid` S \perp h \mid` T$
⟨*proof*⟩

lemma *map-disj-restrict-dom* [*simp*]:
$h_0 \perp h_1 \mid` (dom\ h_1 - dom\ h_0)$
⟨*proof*⟩

lemma *restrict-map-disj-dom-empty*:
$h \perp h' \Longrightarrow h \mid` dom\ h' = Map.empty$
⟨*proof*⟩

lemma *restrict-map-univ-disj-eq*:
$h \perp h' \Longrightarrow h \mid` (UNIV - dom\ h') = h$
⟨*proof*⟩

lemma *restrict-map-disj-dom*:
$h_0 \perp h_1 \Longrightarrow h \mid` dom\ h_0 \perp h \mid` dom\ h_1$
⟨*proof*⟩

lemma *map-add-restrict-dom-left*:
$h \perp h' \Longrightarrow (h ++ h') \mid` dom\ h = h$
⟨*proof*⟩

lemma *restrict-map-disj-left*:
$h_0 \perp h_1 \Longrightarrow h_0 \mid` S \perp h_1$
⟨*proof*⟩

lemma *restrict-map-disj-right*:
$h_0 \perp h_1 \Longrightarrow h_0 \perp h_1 \mid` S$
⟨*proof*⟩

lemmas *restrict-map-disj-both* = *restrict-map-disj-right restrict-map-disj-left*

lemma *map-dom-disj-restrict-right*:
$h_0 \perp h_1 \Longrightarrow (h_0 ++ h_0{}') \mid` dom\ h_1 = h_0{}' \mid` dom\ h_1$
⟨*proof*⟩

lemma *restrict-map-on-disj*:
$h_0{}' \perp h_1 \Longrightarrow h_0 \mid` dom\ h_0{}' \perp h_1$
⟨*proof*⟩

lemma *restrict-map-on-disj'*:
$h_0 \perp h_1 \Longrightarrow h_0 \perp h_1 \mid` S$
⟨*proof*⟩

lemma *map-le-sub-dom*:
$[\![h_0 ++ h_1 \subseteq_m h\ ;\ h_0 \perp h_1]\!] \Longrightarrow h_0 \subseteq_m h \mid` (dom\ h - dom\ h_1)$
⟨*proof*⟩

lemma *map-submap-break*:
$[\![h \subseteq_m h']\!] \Longrightarrow h' = (h' \mid` (UNIV - dom\ h)) ++ h$
⟨*proof*⟩

lemma *map-add-disj-restrict-both*:
$[\![h_0 \perp h_1;\ S \cap S' = \{\};\ T \cap T' = \{\}]\!]$
$\Longrightarrow (h_0 \mid` S) ++ (h_1 \mid` T) \perp (h_0 \mid` S') ++ (h_1 \mid` T')$
⟨*proof*⟩

end

theory *MapExtraTrans*
imports *MapExtra*
begin

lemma *case-option-None-Some* [*simp*]:
case-option None Some P = *P*
⟨*proof*⟩

lemma *heap-merge-dom-exact2*:
⟦ a ++ b = c ++ d; *dom* a = *dom* c; a ⊥ b; c ⊥ d ⟧ ⟹ a=c ∧ b=d
⟨*proof*⟩

lemma *map-add-restrict-sub*:
⟦ *dom* s = X; *dom* t = X − Y ⟧ ⟹
s ++ (t |` ($X$ − $Y$ − $Z$)) = $s$ ++ $t$ ++ $s$ |` Z
⟨*proof*⟩

lemma *map-add-restrict-UNIV*:
⟦ *dom* g ∩ X = {}; *dom* f = *dom* h ⟧ ⟹ f ++ g = f |` (*UNIV* − $X$) ++ $h$ |` X ++ g ++ f |` X
⟨*proof*⟩

end

theory *TypHeap*
imports
Vanilla32
ArchArraysMemInstance
HeapRawState
MapExtraTrans
begin

declare *map-add-assoc* [*simp del*]

definition *wf-heap-val* :: *heap-state* ⇒ *bool* **where**
wf-heap-val s ≡
∀ x t n v. s (x,*SIndexVal*) ≠ *Some* (*STyp* t) ∧ s (x,*SIndexTyp* n) ≠ *Some* (*SValue* v)

type-synonym *typ-slice-list* = (*typ-uinfo* × *typ-base*) *list*

primrec
typ-slice-t :: *typ-uinfo* ⇒ *nat* ⇒ *typ-slice-list* **and**
typ-slice-struct :: *typ-uinfo-struct* ⇒ *nat* ⇒ *typ-slice-list* **and**
typ-slice-list :: (*typ-uinfo*,*field-name*, *unit*) *dt-tuple list* ⇒ *nat* ⇒ *typ-slice-list* **and**
typ-slice-tuple :: (*typ-uinfo*,*field-name*, *unit*) *dt-tuple* ⇒ *nat* ⇒ *typ-slice-list*
where
tl0: *typ-slice-t* (*TypDesc algn st nm*) m = *typ-slice-struct st m* @
[(*if* m = *0 then* ((*TypDesc algn st nm*),*True*) *else*
((*TypDesc algn st nm*),*False*))]

| *tl1*: *typ-slice-struct* (*TypScalar n algn d*) *m* = []
| *tl2*: *typ-slice-struct* (*TypAggregate xs*) *m* = *typ-slice-list xs m*

| *tl3*: *typ-slice-list* [] *m* = []
| *tl4*: *typ-slice-list* (*x*#*xs*) *m* = (*if m* < *size-td* (*dt-fst x*) ∨ *xs* = [] *then*
typ-slice-tuple x m else typ-slice-list xs (*m* − *size-td* (*dt-fst x*)))

| *tl5*: *typ-slice-tuple* (*DTuple t n d*) *m* = *typ-slice-t t m*

definition *list-map* :: *'a list* ⇒ (*nat* ⇀ *'a*) **where**
list-map xs ≡ *map-of* (*zip* [*0*..<*length xs*] *xs*)

definition *s-footprint-untyped* :: *addr* ⇒ *typ-uinfo* ⇒ (*addr* × *s-heap-index*) *set*
where
s-footprint-untyped p t ≡
{(*p* + *of-nat x*,*k*) | *x k*. *x* < *size-td t* ∧
(*k*=*SIndexVal* ∨ (∃ *n*. *k*=*SIndexTyp n* ∧ *n* < *length*
(*typ-slice-t t x*)))}

definition (**in** *c-type*) *s-footprint* :: *'a ptr* ⇒ (*addr* × *s-heap-index*) *set* **where**
s-footprint p ≡ *s-footprint-untyped* (*ptr-val p*) (*typ-uinfo-t TYPE*(*'a*))

definition *empty-htd* :: *heap-typ-desc* **where**
empty-htd ≡ λ*x*. (*False*,*Map.empty*)

definition *dom-s* :: *heap-typ-desc* ⇒ *s-addr set* **where**
dom-s d ≡ {(*x*,*SIndexVal*) | *x*. *fst* (*d x*)} ∪
{(*x*,*SIndexTyp n*) | *x n*. *snd* (*d x*) *n* ≠ *None*}

definition *restrict-s* :: *heap-typ-desc* ⇒ *s-addr set* ⇒ *heap-typ-desc* **where**
restrict-s d X ≡
λ*x*. ((*x*,*SIndexVal*) ∈ *X* ∧ *fst* (*d x*), (λ*y*. *if* (*x*,*SIndexTyp y*) ∈ *X then snd* (*d*
x) *y else None*))

definition *valid-footprint* :: *heap-typ-desc* ⇒ *addr* ⇒ *typ-uinfo* ⇒ *bool* **where**
valid-footprint d x t ≡
let n = *size-td t in*
0 < *n* ∧ (∀ *y*. *y* < *n* ⟶
list-map (*typ-slice-t t y*) ⊆*m* *snd* (*d* (*x* + *of-nat y*)) ∧ *fst* (*d* (*x* +
of-nat y)))

definition (**in** *c-type*) *h-t-valid* ::
heap-typ-desc ⇒ *'a ptr-guard* ⇒ *'a ptr* ⇒ *bool*
(‹(‹*open-block notation*=‹*mixfix heap-typ-desc*››-,- ⊨*t* -)› [*99*,*0*,*99*] *100*)
where *d*,*g* ⊨*t p* ≡ *valid-footprint d* (*ptr-val* (*p*::*'a ptr*)) (*typ-uinfo-t TYPE*(*'a*))
∧ *g p*

type-synonym *'a typ-heap = 'a ptr ⇀ 'a*

definition *proj-h :: heap-state ⇒ heap-mem* **where**
proj-h s ≡ λx. case-option undefined (case-s-heap-value id undefined) (s (x,SIndexVal))

definition *lift-state :: heap-raw-state ⇒ heap-state* **where**
lift-state ≡ λ(h,d) (x,y).
case y of
SIndexVal ⇒ if fst (d x)then Some (SValue (h x)) else None
| SIndexTyp n ⇒ case-option None (Some ∘ STyp) (snd (d x) n)

definition *fun2list :: (nat ⇒ 'a) ⇒ nat ⇒ 'a list* **where**
fun2list f n ≡ if n=0 then [] else map f [0..<n]

definition *null-d :: heap-state ⇒ addr ⇒ nat ⇒ bool* **where**
null-d s x y ≡ s (x,SIndexTyp y) = None

definition *max-d :: heap-state ⇒ addr ⇒ nat* **where**
max-d s x ≡ 1 + (GREATEST y. ¬ null-d s x y)

definition *proj-d :: heap-state ⇒ heap-typ-desc* **where**
proj-d s ≡ λx. (s (x,SIndexVal) ≠ None,
λn. case-option None (Some ∘ s-heap-tag) (s (x,SIndexTyp n)))

definition (**in** *c-type*) *s-valid ::*
heap-state ⇒ 'a ptr-guard ⇒ 'a ptr ⇒ bool
(‹(‹open-block notation=‹mixfix heap-state››-,- ⊨ₛ -)› [100,0,100] 100)
where *s,g ⊨ₛ p ≡ proj-d s,g ⊨ₜ p*

definition *heap-list-s :: heap-state ⇒ nat ⇒ addr ⇒ byte list* **where**
heap-list-s s n p ≡ heap-list (proj-h s) n p

definition (**in** *c-type*) *lift-typ-heap :: 'a ptr-guard ⇒ heap-state ⇒ 'a typ-heap* **where**
lift-typ-heap g s ≡
(Some ∘ from-bytes ∘ heap-list-s s (size-of TYPE('a)) ∘ ptr-val) |` {p. s,g ⊨ₛ p}

definition (**in** *c-type*) *heap-update-s :: 'a ptr ⇒ 'a ⇒ heap-state ⇒ heap-state* **where**
heap-update-s n p s ≡ lift-state (heap-update n p (proj-h s), proj-d s)

definition (**in** *c-type*) *lift-t :: 'a ptr-guard ⇒ heap-raw-state ⇒ 'a typ-heap* **where**
lift-t g ≡ lift-typ-heap g ∘ lift-state

definition *tag-disj :: ('a, 'b) typ-desc ⇒ ('a,'b) typ-desc ⇒ bool*
(‹(‹open-block notation=‹infix tag-disj››- ⊥ₜ -)› [90,90] 90)

where $f \perp_t g \equiv \neg (f \leq g \vee g \leq f)$

definition *ladder-set* :: *typ-uinfo* ⇒ *nat* ⇒ *nat* ⇒ (*typ-uinfo* × *nat*) *set* **where**
ladder-set s n p ≡ {(*t*,*n*+*p*) | *t*. ∃ *k*. (*t*,*k*) ∈ *set* (*typ-slice-t s n*)}

primrec
field-names :: ('*a*,'*b*) *typ-info* ⇒ *typ-uinfo* ⇒
(*qualified-field-name*) *list* **and**
field-names-struct :: ('*a field-desc*,'*b*) *typ-struct* ⇒ *typ-uinfo* ⇒
(*qualified-field-name*) *list* **and**
field-names-list :: (('*a*,'*b*) *typ-info*,*field-name*, '*b*) *dt-tuple list* ⇒ *typ-uinfo* ⇒
(*qualified-field-name*) *list* **and**
field-names-tuple :: (('*a*,'*b*) *typ-info*,*field-name*, '*b*) *dt-tuple* ⇒ *typ-uinfo* ⇒
(*qualified-field-name*) *list*
where
tfs0: *field-names* (*TypDesc algn st nm*) *t* = (*if t*=*export-uinfo* (*TypDesc algn st nm*) *then*
[[]] *else field-names-struct st t*)

| *tfs1*: *field-names-struct* (*TypScalar m algn d*) *t* = []
| *tfs2*: *field-names-struct* (*TypAggregate xs*) *t* = *field-names-list xs t*

| *tfs3*: *field-names-list* [] *t* = []
| *tfs4*: *field-names-list* (*x*#*xs*) *t* = *field-names-tuple x t*@*field-names-list xs t*

| *tfs5*: *field-names-tuple* (*DTuple s f d*) *t* = *map* (λ*fs*. *f*#*fs*) (*field-names s t*)

definition *field-typ-untyped* :: ('*a*,'*b*) *typ-desc* ⇒ *qualified-field-name* ⇒ ('*a*,'*b*) *typ-desc* **where**
field-typ-untyped t n ≡ (*fst* (*the* (*field-lookup t n 0*)))

definition (**in** *c-type*) *field-typ* :: '*a itself* ⇒ *qualified-field-name* ⇒ '*a xtyp-info* **where**
field-typ t n ≡ *field-typ-untyped* (*typ-info-t TYPE*('*a*)) *n*

definition (**in** *c-type*) *fs-consistent* ::
qualified-field-name list ⇒ '*a itself* ⇒ '*b*::*c-type itself* ⇒ *bool* **where**
fs-consistent fs a b ≡ *set fs* ⊆ *set* (*field-names* (*typ-info-t TYPE*('*a*)) (*typ-uinfo-t TYPE*('*b*)))

definition (**in** *c-type*) *field-offset-footprint* :: '*a ptr* ⇒ (*qualified-field-name*) *list* ⇒ '*b ptr set*
where
field-offset-footprint p fs ≡ {*Ptr* &(*p*→*k*) | *k*. *k* ∈ *set fs*}

definition *sub-typ* :: *'a*::*c-type itself* ⇒ *'b*::*c-type itself* ⇒ *bool*
(‹(‹*open-block notation*=‹*infix sub-typ*››- ≤$_\tau$ -)› [*51*, *51*] *50*)
where *s* ≤$_\tau$ *t* ≡ *typ-uinfo-t s* ≤ *typ-uinfo-t t*

definition *sub-typ-proper* :: *'a*::*c-type itself* ⇒ *'b*::*c-type itself* ⇒ *bool*
(‹(‹*open-block notation*=‹*infix sub-typ-proper*››- <$_\tau$ -)› [*51*, *51*] *50*)
where *s* <$_\tau$ *t* ≡ *typ-uinfo-t s* < *typ-uinfo-t t*

definition *peer-typ* :: *'a*::*c-type itself* ⇒ *'b* :: *c-type itself* ⇒ *bool*
where
peer-typ a b ≡ *typ-uinfo-t TYPE*(*'a*) = *typ-uinfo-t TYPE*(*'b*) ∨
typ-uinfo-t TYPE(*'a*) ⊥$_t$ *typ-uinfo-t TYPE*(*'b*)

definition (**in** *c-type*) *guard-mono* :: *'a ptr-guard* ⇒ *'b*::*c-type ptr-guard* ⇒ *bool*
where
guard-mono g g' ≡
∀ *n f p*. *g p* ∧
field-lookup (*typ-uinfo-t TYPE*(*'a*)) *f 0* = *Some* (*typ-uinfo-t TYPE*(*'b*),*n*)
⟶
g' (*Ptr* (*ptr-val p* + *of-nat n*))

primrec (**in** *c-type*) *sub-field-update-t* ::
qualified-field-name list ⇒ *'a ptr* ⇒ *'a* ⇒ *'b*::*c-type typ-heap* ⇒ *'b typ-heap*
where
sft0: *sub-field-update-t* [] *p v s* = *s*
| *sft1*: *sub-field-update-t* (*f*#*fs*) *p* (*v*::*'a*) *s* =
(*let s'* = *sub-field-update-t fs p v s in*
s'(*Ptr* &(*p*→*f*) ↦ *from-bytes* (*access-ti*$_0$ (*field-typ TYPE*(*'a*) *f*) *v*))) |‘
dom (*s*::*'b*::*c-type typ-heap*)

primrec (**in** *c-type*) *update-value-t* :: (*qualified-field-name*) *list* ⇒ *'a* ⇒ *'b* ⇒ *nat*
⇒ *'b*::*c-type*
where
uvt0: *update-value-t* [] *v w x* = *w*
| *uvt1*: *update-value-t* (*f*#*fs*) *v* (*w*::*'b*) *x* = (*if x*=*field-offset TYPE*(*'b*) *f then*
field-update (*field-desc* (*field-typ TYPE*(*'b*) *f*)) (*to-bytes-p* (*v*::*'a*)) (*w*::*'b*::*c-type*)
else update-value-t fs v w x)

definition (**in** *c-type*) *super-field-update-t* :: *'a ptr* ⇒ *'a* ⇒ *'b*::*c-type typ-heap* ⇒
'b typ-heap **where**
super-field-update-t p v s ≡ λ*q*.
if field-of-t p q
then
case-option None
(λ*w*. *Some* (*update-value-t* (*field-names* (*typ-info-t TYPE*(*'b*))
(*typ-uinfo-t TYPE*(*'a*)))
v w (*unat* (*ptr-val p* − *ptr-val q*))))

$(s\ q)$
else s q

definition *heap-footprint* :: *heap-typ-desc* ⇒ *typ-uinfo* ⇒ *addr set* **where**
heap-footprint d t ≡ {*x*. ∃ *y*. *valid-footprint d y t* ∧ *x* ∈ {*y*} ∪ {*y*..+*size-td t*}}

definition (**in** *c-type*) *ptr-safe* :: *'a ptr* ⇒ *heap-typ-desc* ⇒ *bool* **where**
ptr-safe p d ≡ *s-footprint p* ⊆ *dom-s d*

primrec
htd-update-list :: *addr* ⇒ *typ-slice list* ⇒ *heap-typ-desc* ⇒ *heap-typ-desc*
where
hul0: *htd-update-list p* [] *d* = *d*
| *hul1*: *htd-update-list p* (*x*#*xs*) *d* = *htd-update-list* (*p*+*1*) *xs* (*d*(*p* := (*True*,*snd* (*d p*) ++ *x*)))

definition *dom-tll* :: *addr* ⇒ *typ-slice list* ⇒ *s-addr set* **where**
dom-tll p xs ≡ {(*p* + *of-nat x*,*SIndexVal*) | *x*. *x* < *length xs*} ∪
{(*p* + *of-nat x*,*SIndexTyp n*) | *x n*. *x* < *length xs* ∧ (*xs* ! *x*) *n* ≠ *None*}

definition (**in** *c-type*) *typ-slices* :: *'a itself* ⇒ *typ-slice list* **where**
typ-slices t ≡ *map* (λ*n*. *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*('*a*)) *n*)) [*0*..<*size-of TYPE*('*a*)]

definition (**in** *c-type*) *ptr-retyp* :: *'a ptr* ⇒ *heap-typ-desc* ⇒ *heap-typ-desc* **where**
ptr-retyp p ≡ *htd-update-list* (*ptr-val p*) (*typ-slices TYPE*('*a*))

definition (**in** *c-type*) *field-fd* :: *'a itself* ⇒ *qualified-field-name* ⇒ *'a field-desc* **where**
field-fd t n ≡ *field-desc* (*field-typ t n*)

definition *tag-disj-typ* :: *'a*::*c-type itself* ⇒ *'b*::*c-type itself* ⇒ *bool*
(‹(‹*open-block notation*=‹*infix tag-disj-typ*››- ⊥$_\tau$ -)›)
where *s* ⊥$_\tau$ *t* ≡ *typ-uinfo-t s* ⊥$_t$ *typ-uinfo-t t*

—-

lemma *wf-heap-val-SIndexVal-STyp-simp* [*simp*]:
wf-heap-val s ⟹ *s* (*x*,*SIndexVal*) ≠ *Some* (*STyp t*)
⟨*proof*⟩

lemma *wf-heap-val-SIndexTyp-SValue-simp* [*simp*]:
wf-heap-val s ⟹ *s* (*x*,*SIndexTyp n*) ≠ *Some* (*SValue v*)
⟨*proof*⟩

lemma (**in** *mem-type*) *field-tag-sub*:
field-lookup (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*,*n*) ⟹

$\{\&(p\rightarrow f)..+size\text{-}td\ t\} \subseteq \{ptr\text{-}val\ (p::'a\ ptr)..+size\text{-}of\ TYPE('a)\}$
⟨*proof*⟩

lemma *typ-slice-t-not-empty* [*simp*]:
$typ\text{-}slice\text{-}t\ t\ n \neq []$
⟨*proof*⟩

lemma *list-map-typ-slice-t-not-empty* [*simp*]:
$list\text{-}map\ (typ\text{-}slice\text{-}t\ t\ n) \neq Map.empty$
⟨*proof*⟩

lemma (**in** *c-type*) *s-footprint*:
$s\text{-}footprint\ (p::'a\ ptr) =$
$\{(ptr\text{-}val\ p\ +\ of\text{-}nat\ x,k) \mid x\ k.$
$x < size\text{-}of\ TYPE('a) \wedge$
$(k=SIndexVal \vee (\exists n.\ k=SIndexTyp\ n \wedge n < length\ (typ\text{-}slice\text{-}t\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ x)))\}$
⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-val-SIndexVal-in-s-footprint* [*simp*]:
$(ptr\text{-}val\ p,\ SIndexVal) \in s\text{-}footprint\ (p::'a\ ptr)$
⟨*proof*⟩

lemma (**in** *c-type*) *s-footprintI*:
$[\![\ n < length\ (typ\text{-}slice\text{-}t\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ x);\ x < size\text{-}of\ TYPE('a)\ ]\!] \Longrightarrow$
$(ptr\text{-}val\ p\ +\ of\text{-}nat\ x,\ SIndexTyp\ n) \in s\text{-}footprint\ (p::'a\ ptr)$
⟨*proof*⟩

lemma (**in** *c-type*) *s-footprintI2*:
$x < size\text{-}of\ TYPE('a) \Longrightarrow (ptr\text{-}val\ p\ +\ of\text{-}nat\ x,\ SIndexVal) \in s\text{-}footprint\ (p::'a\ ptr)$
⟨*proof*⟩

lemma (**in** *c-type*) *s-footprintD*:
$(x,k) \in s\text{-}footprint\ p \Longrightarrow x \in \{ptr\text{-}val\ (p::'a\ ptr)..+size\text{-}of\ TYPE('a)\}$
⟨*proof*⟩

lemma (**in** *mem-type*) *s-footprintD2*:
$(x,SIndexTyp\ n) \in s\text{-}footprint\ (p::'a\ ptr) \Longrightarrow$
$n < length\ (typ\text{-}slice\text{-}t\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ (unat\ (x\ -\ ptr\text{-}val\ p)))$
⟨*proof*⟩

lemma (**in** *c-type*) *s-footprint-restrict*:
$x \in s\text{-}footprint\ p \Longrightarrow (s\ |`\ s\text{-}footprint\ p)\ x = s\ x$
⟨*proof*⟩

lemma *restrict-s-fst*:
$fst\ (restrict\text{-}s\ d\ X\ x) \Longrightarrow fst\ (d\ x)$
⟨*proof*⟩

lemma *restrict-s-map-le* [*simp*]:
snd (*restrict-s d X x*) $\subseteq_m$ *snd* (*d x*)
⟨*proof*⟩

lemma *dom-list-map* [*simp*]:
dom (*list-map xs*) = {*0*..<*length xs*}
⟨*proof*⟩

lemma *list-map* [*simp*]:
n < *length xs* ⟹ *list-map xs n* = *Some* (*xs* ! *n*)
⟨*proof*⟩

lemma *list-map-eq*:
list-map xs n = (*if n* < *length xs then Some* (*xs* ! *n*) *else None*)
⟨*proof*⟩

lemma *valid-footprintI*:
⟦ *0* < *size-td t*; ⋀*y*. *y* < *size-td t* ⟹ *list-map* (*typ-slice-t t y*) $\subseteq_m$ *snd* (*d* (*x* + *of-nat y*)) ∧
fst (*d* (*x* + *of-nat y*)) ⟧ ⟹
valid-footprint d x t
⟨*proof*⟩

lemma *valid-footprintD*:
⟦ *valid-footprint d x t*; *y* < *size-td t* ⟧ ⟹
list-map (*typ-slice-t t y*) $\subseteq_m$ *snd* (*d* (*x* + *of-nat y*)) ∧
fst (*d* (*x* + *of-nat y*))
⟨*proof*⟩

lemma (**in** *c-type*) *h-t-valid-taut*:
d,*g* ⊨$_t$ *p* ⟹ *d*,(λ*x*. *True*) ⊨$_t$ *p*
⟨*proof*⟩

lemma (**in** *c-type*) *h-t-valid-restrict*:
restrict-s d (*s-footprint p*),*g* ⊨$_t$ *p* = *d*,*g* ⊨$_t$ *p*
⟨*proof*⟩

lemma (**in** *c-type*) *h-t-valid-restrict2*:
⟦ *d*,*g* ⊨$_t$ *p*; *restrict-s d* (*s-footprint p*) = *restrict-s d'* (*s-footprint p*) ⟧ ⟹ *d'*,*g* ⊨$_t$ (*p*::'*a ptr*)
⟨*proof*⟩

lemma *lift-state-wf-heap-val* [*simp*]:
wf-heap-val (*lift-state* (*h*,*d*))
⟨*proof*⟩

lemma *wf-hs-proj-d*:
fst (*proj-d s x*) ⟹ *s* (*x*,*SIndexVal*) ≠ *None*

⟨*proof*⟩

lemma (**in** *c-type*) *s-valid-g*:
s,*g* ⊨$_s$ *p* ⟹ *g p*
⟨*proof*⟩

lemma (**in** *c-type*) *lift-typ-heap-if*:
lift-typ-heap g s = (λ(*p*::′*a ptr*). *if s*,*g* ⊨$_s$ *p then Some* (*from-bytes*
(*heap-list-s s* (*size-of TYPE*(′*a*)) (*ptr-val p*))) *else None*)
⟨*proof*⟩

lemma (**in** *c-type*) *lift-typ-heap-s-valid*:
lift-typ-heap g s p = *Some x* ⟹ *s*,*g* ⊨$_s$ *p*
⟨*proof*⟩

lemma (**in** *c-type*) *lift-typ-heap-g*:
lift-typ-heap g s p = *Some x* ⟹ *g p*
⟨*proof*⟩

lemma *lift-state-empty* [*simp*]:
lift-state (*h*,*empty-htd*) = *Map.empty*
⟨*proof*⟩

lemma *lift-state-eqI*:
⟦ *h x* = *h*′ *x*; *d x* = *d*′ *x* ⟧ ⟹ *lift-state* (*h*,*d*) (*x*,*k*) = *lift-state* (*h*′,*d*′) (*x*,*k*)
⟨*proof*⟩

lemma *proj-h-lift-state*:
fst (*d x*) ⟹ *proj-h* (*lift-state* (*h*,*d*)) *x* = *h x*
⟨*proof*⟩

lemma *lift-state-proj-simp* [*simp*]:
lift-state (*proj-h* (*lift-state* (*h*, *d*)), *d*) = *lift-state* (*h*, *d*)
⟨*proof*⟩

lemma *f2l-length* [*simp*]:
length (*fun2list f n*) = *n*
⟨*proof*⟩

lemma *GREATEST-lt* [*simp*]:
0 < *n* ⟹ (*GREATEST x*. *x* < *n*) = *n* − (*1*::*nat*)
⟨*proof*⟩

lemma *fun2list-nth* [*simp*]:
x < *n* ⟹ *fun2list f n* ! *x* = *f x*
⟨*proof*⟩

lemma *proj-d-lift-state*:

proj-d (*lift-state* (*h*,*d*)) = *d*
⟨*proof*⟩

lemma *lift-state-proj* [*simp*]:
wf-heap-val s ⟹ *lift-state* (*proj-h s*,*proj-d s*) = *s*
⟨*proof*⟩

lemma *lift-state-Some*:
lift-state (*h*,*d*) (*p*,*SIndexTyp n*) = *Some t* ⟹ *snd* (*d p*) *n* = *Some* (*s-heap-tag t*)
⟨*proof*⟩

lemma *lift-state-Some2*:
snd (*d p*) *n* = *Some t* ⟹
∃ *v*. *lift-state* (*h*,*d*) (*p*,*SIndexTyp n*) = *Some* (*STyp t*)
⟨*proof*⟩

lemma (**in** *c-type*) *h-t-s-valid*:
lift-state (*h*,*d*),*g* ⊨$_s$ *p* = *d*,*g* ⊨$_t$ *p*
⟨*proof*⟩

lemma (**in** *c-type*) *lift-t*:
lift-typ-heap g (*lift-state s*) = *lift-t g s*
⟨*proof*⟩

lemma (**in** *c-type*) *lift-t-h-t-valid*:
lift-t g (*h*,*d*) *p* = *Some x* ⟹ *d*,*g* ⊨$_t$ *p*
⟨*proof*⟩

lemma (**in** *c-type*) *lift-t-g*:
lift-t g s p = *Some x* ⟹ *g p*
⟨*proof*⟩

lemma (**in** *c-type*) *lift-t-proj* [*simp*]:
wf-heap-val s ⟹ *lift-t g* (*proj-h s*, *proj-d s*) = *lift-typ-heap g s*
⟨*proof*⟩

lemma *valid-footprint-Some*:
assumes *valid*: *valid-footprint d p t* **and** *size*: *x* < *size-td t*
shows *fst* (*d* (*p* + *of-nat x*))
⟨*proof*⟩

lemma (**in** *c-type*) *h-t-valid-Some*:
⟦ *d*,*g* ⊨$_t$ (*p*::'*a ptr*); *x* < *size-of TYPE*('*a*) ⟧ ⟹
fst (*d* (*ptr-val p* + *of-nat x*))
⟨*proof*⟩

lemma (**in** *c-type*) *h-t-valid-ptr-safe*:
d,*g* ⊨$_t$ (*p*::'*a ptr*) ⟹ *ptr-safe p d*
⟨*proof*⟩

lemma (**in** *c-type*) *lift-t-ptr-safe*:
lift-t g (h,d) (p::'a ptr) = Some x ⟹ ptr-safe p d
⟨*proof*⟩

lemma (**in** *c-type*) *s-valid-Some*:
⟦ d,g ⊨ₛ (p::'a ptr); x < size-of TYPE('a) ⟧ ⟹
d (ptr-val p + of-nat x,SIndexVal) ≠ None
⟨*proof*⟩

lemma *heap-list-s-heap-list-dom*:
⋀n. (λx. (x,SIndexVal)) ` {n..+k} ⊆ dom-s d ⟹
heap-list-s (lift-state (h,d)) k n = heap-list h k n
⟨*proof*⟩

lemma (**in** *c-type*) *heap-list-s-heap-list*:
d,(λx. True) ⊨ₜ (p::'a ptr) ⟹
heap-list-s (lift-state (h,d)) (size-of TYPE('a)) (ptr-val p)
= heap-list h (size-of TYPE('a)) (ptr-val p)
⟨*proof*⟩

lemma (**in** *c-type*) *lift-t-if*:
lift-t g (h,d) = (λp. if d,g ⊨ₜ p then Some (h-val h (p::'a ptr)) else None)
⟨*proof*⟩

lemma (**in** *c-type*) *lift-lift-t*:
d,g ⊨ₜ (p::'a ptr) ⟹ lift h p = the (lift-t g (h,d) p)
⟨*proof*⟩

lemma (**in** *c-type*) *lift-t-lift*:
lift-t g (h,d) (p::'a ptr) = Some v ⟹ lift h p = v
⟨*proof*⟩

lemma *heap-update-list-same*:
⋀h p k. ⟦ 0 < k; k ≤ addr-card − length v ⟧ ⟹ heap-update-list (p + of-nat k) v h p = h p
⟨*proof*⟩

lemma *heap-list-update*:
⋀h p. length v ≤ addr-card ⟹
heap-list (heap-update-list p v h) (length v) p = v
⟨*proof*⟩

lemma (**in** *mem-type*) *heap-list-update-to-bytes*:
heap-list (heap-update-list p (to-bytes (v::'a) (heap-list h (size-of TYPE('a)) p)) h)
(size-of TYPE('a)) p = to-bytes v (heap-list h (size-of TYPE('a)) p)
⟨*proof*⟩

lemma (**in** *mem-type*) *heap-list-update-to-bytes-padding*:
length bs = *size-of TYPE*('*a*) ⟹ *heap-list* (*heap-update-list p* (*to-bytes* (*v*::'*a*) *bs*) *h*)
(*size-of TYPE*('*a*)) *p* = *to-bytes v bs*
⟨*proof*⟩

lemma (**in** *mem-type*) *h-val-heap-update*[*simp*]:
h-val (*heap-update p v h*) *p* = (*v*::'*a*)
⟨*proof*⟩

lemma (**in** *mem-type*) *h-val-heap-update-padding*:
fixes *p*:: '*a ptr*
shows *length bs* = *size-of TYPE*('*a*) ⟹ *h-val* (*heap-update-padding p v bs h*) *p* = *v*
⟨*proof*⟩

lemma *heap-list-update-disjoint-same*:
shows ⋀*q*. {*p*..+*length v*} ∩ {*q*..+*k*} = {} ⟹
heap-list (*heap-update-list p v h*) *k q* = *heap-list h k q*
⟨*proof*⟩

lemma *heap-update-nmem-same*:
assumes *nmem*: *q* ∉ {*p*..+*length v*}
shows *heap-update-list p v h q* = *h q*
⟨*proof*⟩

lemma *heap-update-mem-same*:
⟦ *q* ∈ {*p*..+*length v*}; *length v* < *addr-card* ⟧ ⟹
heap-update-list p v h q = *heap-update-list p v h*′ *q*
⟨*proof*⟩

lemma *sub-tag-proper-TypScalar* [*simp*]:
¬ *t* < *TypDesc algn*′ (*TypScalar n algn d*) *nm*
⟨*proof*⟩

lemma *tag-disj-com* [*simp*]:
f ⊥*t g* = *g* ⊥*t f*
⟨*proof*⟩

lemma *typ-slice-set*′:
∀ *m n*. *fst* ‘ *set* (*typ-slice-t s n*) ⊆ *fst* ‘ *td-set s m*
∀ *m n*. *fst* ‘ *set* (*typ-slice-struct st n*) ⊆ *fst* ‘ *td-set-struct st m*
∀ *m n*. *fst* ‘ *set* (*typ-slice-list xs n*) ⊆ *fst* ‘ *td-set-list xs m*
∀ *m n*. *fst* ‘ *set* (*typ-slice-tuple x n*) ⊆ *fst* ‘ *td-set-tuple x m*
⟨*proof*⟩

lemma *typ-slice-set*:
fst ‘ *set* (*typ-slice-t s n*) ⊆ *fst* ‘ *td-set s m*
⟨*proof*⟩

lemma *typ-slice-struct-set*:
$(s,t) \in set\ (typ\text{-}slice\text{-}struct\ st\ n) \Longrightarrow \exists k.\ (s,k) \in td\text{-}set\text{-}struct\ st\ m$
⟨*proof*⟩

lemma *typ-slice-set-sub*:
$typ\text{-}slice\text{-}t\ s\ m \leq typ\text{-}slice\text{-}t\ t\ n \Longrightarrow$
$fst\ `\ set\ (typ\text{-}slice\text{-}t\ s\ m) \subseteq fst\ `\ set\ (typ\text{-}slice\text{-}t\ t\ n)$
⟨*proof*⟩

lemma *ladder-set-self*:
$s \in fst\ `\ set\ (typ\text{-}slice\text{-}t\ s\ n)$
⟨*proof*⟩

lemma *typ-slice-sub*:
$typ\text{-}slice\text{-}t\ s\ m \leq typ\text{-}slice\text{-}t\ t\ n \Longrightarrow s \leq t$
⟨*proof*⟩

lemma *typ-slice-self*:
$(s, True) \in set\ (typ\text{-}slice\text{-}t\ s\ 0)$
⟨*proof*⟩

lemma *typ-slice-struct-nmem*:
$(TypDesc\ align\ st\ nm,n) \notin set\ (typ\text{-}slice\text{-}struct\ st\ k)$
⟨*proof*⟩

lemma *typ-slice-0-prefix*:
$0 < n \Longrightarrow \neg\ typ\text{-}slice\text{-}t\ t\ 0 \leq typ\text{-}slice\text{-}t\ t\ n \wedge \neg\ typ\text{-}slice\text{-}t\ t\ n \leq typ\text{-}slice\text{-}t\ t\ 0$
⟨*proof*⟩

lemma *prefix-eq-nth*:
$xs \leq ys = ((\forall i.\ i < length\ xs \longrightarrow xs\ !\ i = ys\ !\ i) \wedge length\ xs \leq length\ ys)$
⟨*proof*⟩

lemma *map-prefix-same-cases*:
$[\![\ list\text{-}map\ xs \subseteq_m f;\ list\text{-}map\ ys \subseteq_m f\ ]\!] \Longrightarrow xs \leq ys \vee ys \leq xs$
⟨*proof*⟩

lemma *list-map-mono*:
$xs \leq ys \Longrightarrow list\text{-}map\ xs \subseteq_m list\text{-}map\ ys$
⟨*proof*⟩

lemma *map-list-map-trans*:
$[\![\ xs \leq ys;\ list\text{-}map\ ys \subseteq_m f\ ]\!] \Longrightarrow list\text{-}map\ xs \subseteq_m f$
⟨*proof*⟩

lemma *valid-footprint-le*:
$valid\text{-}footprint\ d\ x\ t \Longrightarrow size\text{-}td\ t \leq addr\text{-}card$
⟨*proof*⟩

lemma *typ-slice-True-set'*:
$\forall s\ k\ m.\ (s,True) \in set\ (typ\text{-}slice\text{-}t\ t\ k) \longrightarrow (s,k+m) \in td\text{-}set\ t\ m$
$\forall s\ k\ m.\ (s,True) \in set\ (typ\text{-}slice\text{-}struct\ st\ k) \longrightarrow (s,k+m) \in td\text{-}set\text{-}struct\ st\ m$
$\forall s\ k\ m.\ (s,True) \in set\ (typ\text{-}slice\text{-}list\ xs\ k) \longrightarrow (s,k+m) \in td\text{-}set\text{-}list\ xs\ m$
$\forall s\ k\ m.\ (s,True) \in set\ (typ\text{-}slice\text{-}tuple\ x\ k) \longrightarrow (s,k+m) \in td\text{-}set\text{-}tuple\ x\ m$
⟨*proof*⟩

lemma *typ-slice-True-set*:
$(s,True) \in set\ (typ\text{-}slice\text{-}t\ t\ k) \Longrightarrow (s,k+m) \in td\text{-}set\ t\ m$
⟨*proof*⟩

lemma *typ-slice-True-prefix*:
$typ\text{-}slice\text{-}t\ s\ 0 \leq typ\text{-}slice\text{-}t\ t\ k \Longrightarrow (s,k) \in td\text{-}set\ t\ 0$
⟨*proof*⟩

lemma *tag-sub-prefix* [*simp*]:
$t < s \Longrightarrow \neg\ typ\text{-}slice\text{-}t\ s\ m \leq typ\text{-}slice\text{-}t\ t\ n$
⟨*proof*⟩

lemma *tag-disj-prefix* [*simp*]:
$s \perp_t t \Longrightarrow \neg\ typ\text{-}slice\text{-}t\ s\ m \leq typ\text{-}slice\text{-}t\ t\ n$
⟨*proof*⟩

lemma *typ-slice-0-True'*:
$\forall x.\ x \in set\ (typ\text{-}slice\text{-}t\ t\ 0) \longrightarrow snd\ x = True$
$\forall x.\ x \in set\ (typ\text{-}slice\text{-}struct\ st\ 0) \longrightarrow snd\ x = True$
$\forall x.\ x \in set\ (typ\text{-}slice\text{-}list\ xs\ 0) \longrightarrow snd\ x = True$
$\forall x.\ x \in set\ (typ\text{-}slice\text{-}tuple\ y\ 0) \longrightarrow snd\ x = True$
⟨*proof*⟩

lemma *typ-slice-0-True*:
$x \in set\ (typ\text{-}slice\text{-}t\ t\ 0) \Longrightarrow snd\ x = True$
⟨*proof*⟩

lemma *typ-slice-False-self*:
$k \neq 0 \Longrightarrow (t,False) \in set\ (typ\text{-}slice\text{-}t\ t\ k)$
⟨*proof*⟩

lemma *tag-prefix-True*:
$typ\text{-}slice\text{-}t\ s\ k \leq typ\text{-}slice\text{-}t\ t\ 0 \Longrightarrow k = 0$
⟨*proof*⟩

lemma *valid-footprint-neq-nmem*:
assumes *valid-p*: *valid-footprint d p f* **and**
valid-q: *valid-footprint d q g* **and**
neq: $p \neq q$ **and**
disj: $f \perp_t g \vee f{=}g$
shows $p \notin \{q..{+}size\text{-}td\ g\}$ (**is** *?G*)

⟨*proof*⟩

lemma *valid-footprint-sub*:
assumes *valid-p*: *valid-footprint d p s*
assumes *valid-q*: *valid-footprint d q t*
assumes *sub*: ¬ *t* < *s*
shows *p* ∉ {*q*..+*size-td t*} ∨ *field-of* (*p* − *q*) (*s*) (*t*) (**is** *?G*)
⟨*proof*⟩

lemma *valid-footprint-sub2*:
⟦ *valid-footprint d p s*; *valid-footprint d q t*; ¬ *t* < *s* ⟧ ⟹
q ∉ {*p*..+*size-td s*} ∨ *p*=*q*
⟨*proof*⟩

lemma *valid-footprint-neq-disjoint*:
⟦ *valid-footprint d p s*; *valid-footprint d q t*; ¬ *t* < *s*;
¬ *field-of* (*p* − *q*) (*s*) (*t*) ⟧ ⟹ {*p*..+*size-td s*} ∩ {*q*..+*size-td t*} = {}
⟨*proof*⟩

lemma *sub-typ-proper-not-same* [*simp*]:
¬ *t* <$_\tau$ *t*
⟨*proof*⟩

lemma *sub-typ-proper-not-simple* [*simp*]:
¬ *TYPE*('*a*::*c-type*) <$_\tau$ *TYPE*('*b*::*simple-mem-type*)
⟨*proof*⟩

lemma *field-of-sub*:
field-of p s t ⟹ *s* ≤ *t*
⟨*proof*⟩

lemma *h-t-valid-neq-disjoint*:
⟦ *d*,*g* ⊨$_t$ (*p*::'*a*::*c-type ptr*); *d*,*g'* ⊨$_t$ (*q*::'*b*::*c-type ptr*);
¬ *TYPE*('*b*) <$_\tau$ *TYPE*('*a*); ¬ *field-of-t p q* ⟧ ⟹ {*ptr-val p*..+*size-of TYPE*('*a*)}
∩
{*ptr-val q*..+*size-of TYPE*('*b*)} = {}
⟨*proof*⟩

lemma *field-ti-sub-typ*:
⟦ *field-ti* (*TYPE*('*b*::*mem-type*)) *f* = *Some t*; *export-uinfo t* = (*typ-uinfo-t TYPE*('*a*::*c-type*))
⟧ ⟹
TYPE('*a*) ≤$_\tau$ *TYPE*('*b*)
⟨*proof*⟩

lemma *h-t-valid-neq-disjoint-simple*:
⟦ *d*,*g* ⊨$_t$ (*p*::'*a*::*simple-mem-type ptr*); *d*,*g'* ⊨$_t$ (*q*::'*b*::*simple-mem-type ptr*) ⟧
⟹ *ptr-val p* ≠ *ptr-val q* ∨ *typ-uinfo-t TYPE*('*a*) = *typ-uinfo-t TYPE*('*b*)
⟨*proof*⟩

lemma *h-val-heap-same*:
fixes *p*::*'a*::*mem-type ptr* **and** *q*::*'b*::*c-type ptr*
assumes *val-p*: *d*,*g* $\models_t$ *p* **and** *val-q*: *d*,*g'* $\models_t$ *q* **and**
subt: $\neg$ *TYPE*(*'a*) $<_\tau$ *TYPE*(*'b*) **and** *nf*: $\neg$ *field-of-t q p*
shows *h-val* (*heap-update p v h*) *q* = *h-val h q*
⟨*proof*⟩

lemma *h-val-heap-same-padding*:
fixes *p*::*'a*::*mem-type ptr* **and** *q*::*'b*::*c-type ptr*
assumes *val-p*: *d*,*g* $\models_t$ *p* **and** *val-q*: *d*,*g'* $\models_t$ *q* **and**
subt: $\neg$ *TYPE*(*'a*) $<_\tau$ *TYPE*(*'b*) **and** *nf*: $\neg$ *field-of-t q p*
assumes *lbs*: *length bs* = *size-of TYPE*(*'a*)
shows *h-val* (*heap-update-padding p v bs h*) *q* = *h-val h q*
⟨*proof*⟩

lemma *peer-typI*:
typ-uinfo-t TYPE(*'a*) $\perp_t$ *typ-uinfo-t TYPE*(*'b*) $\Longrightarrow$ *peer-typ* (*a*::*'a*::*c-type itself*)
(*b*::*'b*::*c-type itself*)
⟨*proof*⟩

lemma *peer-typD*:
peer-typ TYPE(*'a*::*c-type*) *TYPE*(*'b*::*c-type*) $\Longrightarrow$ $\neg$ *TYPE*(*'a*) $<_\tau$ *TYPE*(*'b*)
⟨*proof*⟩

lemma *peer-typ-refl* [*simp*]:
peer-typ t t
⟨*proof*⟩

lemma *peer-typ-simple* [*simp*]:
peer-typ TYPE(*'a*::*simple-mem-type*) *TYPE*(*'b*::*simple-mem-type*)
⟨*proof*⟩

lemmas *peer-typ-nlt* = *peer-typD*

lemma *peer-typ-not-field-of*:
⟦ *peer-typ TYPE*(*'a*::*c-type*) *TYPE*(*'b*::*c-type*); *ptr-val p* $\neq$ *ptr-val q* ⟧ $\Longrightarrow$
$\neg$ *field-of-t* (*q*::*'b ptr*) (*p*::*'a ptr*)
⟨*proof*⟩

lemma *h-val-heap-same-peer*:
⟦ *d*,*g* $\models_t$ (*p*::*'a*::*mem-type ptr*); *d*,*g'* $\models_t$ (*q*::*'b*::*c-type ptr*);
ptr-val p $\neq$ *ptr-val q*; *peer-typ TYPE*(*'a*) *TYPE*(*'b*) ⟧ $\Longrightarrow$
h-val (*heap-update p v h*) *q* = *h-val h q*
⟨*proof*⟩

lemma *h-val-heap-same-peer-padding*:
⟦ *d*,*g* $\models_t$ (*p*::*'a*::*mem-type ptr*); *d*,*g'* $\models_t$ (*q*::*'b*::*c-type ptr*);

ptr-val p ≠ ptr-val q; *peer-typ TYPE('a) TYPE('b)*; *length bs = size-of TYPE('a)* ⟧ ⟹
h-val (heap-update-padding p v bs h) q = h-val h q
⟨*proof*⟩

lemma *field-offset-footprint-cons-simp* [*simp*]:
field-offset-footprint p (x#xs) = {Ptr &(p→x)} ∪ field-offset-footprint p xs
⟨*proof*⟩

lemma *heap-list-update-list*:
⟦ *n + x ≤ length v*; *length v < addr-card* ⟧ ⟹
heap-list (heap-update-list p v h) n (p + of-nat x) = take n (drop x v)
⟨*proof*⟩

lemma *typ-slice-td-set'*:
∀ *s m n k*. *(s,m + n) ∈ td-set t m ∧ k < size-td s* ⟶
typ-slice-t s k ≤ typ-slice-t t (n + k)
∀ *s m n k*. *(s,m + n) ∈ td-set-struct st m ∧ k < size-td s* ⟶
typ-slice-t s k ≤ typ-slice-struct st (n + k)
∀ *s m n k*. *(s,m + n) ∈ td-set-list ts m ∧ k < size-td s* ⟶
typ-slice-t s k ≤ typ-slice-list ts (n + k)
∀ *s m n k*. *(s,m + n) ∈ td-set-tuple x m ∧ k < size-td s* ⟶
typ-slice-t s k ≤ typ-slice-tuple x (n + k)
⟨*proof*⟩

lemma *typ-slice-td-set*:
⟦ *(s,n) ∈ td-set t 0*; *k < size-td s* ⟧ ⟹
typ-slice-t s k ≤ typ-slice-t t (n + k)
⟨*proof*⟩

lemma *typ-slice-td-set-list*:
⟦ *(s,n) ∈ td-set-list ts 0*; *k < size-td s* ⟧ ⟹
typ-slice-t s k ≤ typ-slice-list ts (n + k)
⟨*proof*⟩

lemma *h-t-valid-sub*:
⟦ *d,g ⊨ₜ (p::'b::mem-type ptr)*;
field-ti TYPE('b) f = Some t; *export-uinfo t = (typ-uinfo-t TYPE('a))* ⟧ ⟹
d,(λx. True) ⊨ₜ ((Ptr &(p→f))::'a::mem-type ptr)
⟨*proof*⟩

lemma *size-of-tag*:
size-td (typ-uinfo-t t) = size-of t
⟨*proof*⟩

lemma *size-of-neq-implies-typ-uinfo-t-neq* [*simp*]:
size-of TYPE('a::c-type) ≠ size-of TYPE('b::c-type) ⟹ typ-uinfo-t TYPE('a) ≠ typ-uinfo-t TYPE('b)
⟨*proof*⟩

lemma *guard-mono-self*[*simp*]:
guard-mono g g
⟨*proof*⟩

lemma (**in** *c-type*) *field-lookup-offset-size*:
field-lookup (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*,*n*) ⟹
size-td t + *n* ≤ *size-td* (*typ-info-t TYPE*('*a*))
⟨*proof*⟩

lemma (**in** *mem-type*) *sub-h-t-valid'*:
⟦ *d*,*g* ⊨$_t$ (*p*::'*a ptr*);
field-lookup (*typ-uinfo-t TYPE*('*a*)) *f 0* = *Some* (*typ-uinfo-t TYPE*('*b*),*n*);
guard-mono g (*g'*::'*b*::*mem-type ptr-guard*) ⟧ ⟹
d,*g'* ⊨$_t$ ((*Ptr* (*ptr-val p* + *of-nat n*))::'*b*::*mem-type ptr*)
⟨*proof*⟩

lemma (**in** *mem-type*) *sub-h-t-valid*:
⟦ *d*,*g* ⊨$_t$ (*p*::'*a ptr*); (*typ-uinfo-t TYPE*('*b*),*n*) ∈ *td-set* (*typ-uinfo-t TYPE*('*a*)) *0*
⟧ ⟹
d,(λ*x*. *True*) ⊨$_t$ ((*Ptr* (*ptr-val p* + *of-nat n*))::'*b*::*mem-type ptr*)
⟨*proof*⟩

lemma (**in** *mem-type*) *h-t-valid-mono*:
⟦ *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*,*n*);
export-uinfo s = *typ-uinfo-t TYPE*('*b*); *guard-mono g g'* ⟧ ⟹
d,*g* ⊨$_t$ (*p*::'*a ptr*) ⟶ *d*,*g'* ⊨$_t$ (*Ptr* (&(*p*→*f*))::'*b*::*mem-type ptr*)
⟨*proof*⟩

lemma (**in** *mem-type*) *s-valid-mono*:
⟦ *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*,*n*);
export-uinfo s = *typ-uinfo-t TYPE*('*b*); *guard-mono g g'* ⟧ ⟹
d,*g* ⊨$_s$ (*p*::'*a ptr*) ⟶ *d*,*g'* ⊨$_s$ (*Ptr* (&(*p*→*f*))::'*b*::*mem-type ptr*)
⟨*proof*⟩

lemma *take-heap-list-le*:
k ≤ *n* ⟹ *take k* (*heap-list h n x*) = *heap-list h k x*
⟨*proof*⟩

lemma *drop-heap-list-le*:
k ≤ *n* ⟹ *drop k* (*heap-list h n x*) = *heap-list h* (*n* − *k*) (*x* + *of-nat k*)
⟨*proof*⟩

lemma (**in** *mem-type*) *h-val-field-from-bytes'*:
⟦ *field-ti TYPE*('*a*) *f* = *Some t*;
export-uinfo t = *export-uinfo* (*typ-info-t TYPE*('*b*::{*mem-type*})) ⟧ ⟹
h-val h (*Ptr* &(*pa*→*f*) :: '*b ptr*) = *from-bytes* (*access-ti*$_0$ *t* (*h-val h pa*))
⟨*proof*⟩

lemma (**in** *mem-type*) *h-val-field-from-bytes*:
⟦ *field-ti TYPE('a) f = Some t*;
export-uinfo t = export-uinfo (typ-info-t TYPE('b::{mem-type})) ⟧ ⟹
h-val (hrs-mem h) (Ptr &(pa→f) :: 'b ptr) = from-bytes (access-ti$_0$ *t (h-val (hrs-mem h) pa))*
⟨*proof*⟩

lemma (**in** *mem-type*) *lift-typ-heap-mono*:
⟦ *field-lookup (typ-info-t TYPE('a)) f 0 = Some (t,n)*;
lift-typ-heap g s (p::'a ptr) = Some v;
export-uinfo t = typ-uinfo-t TYPE('b); *guard-mono g g'* ⟧ ⟹
lift-typ-heap g' s (Ptr (&(p→f))::'b::mem-type ptr) = Some (from-bytes (access-ti$_0$ *t v))*
⟨*proof*⟩

lemma (**in** *mem-type*) *lift-t-mono*:
⟦ *field-lookup (typ-info-t TYPE('a)) f 0 = Some (t,n)*;
lift-t g s (p::'a ptr) = Some v;
export-uinfo t = typ-uinfo-t TYPE('b); *guard-mono g g'* ⟧ ⟹
lift-t g' s (Ptr (&(p→f))::'b::mem-type ptr) = Some (from-bytes (access-ti$_0$ *t v))*
⟨*proof*⟩

lemma *align-td-wo-align-field-lookupD*:
field-lookup (t::('a,'b) typ-desc) f m = Some (s, n) ⟹ *align-td-wo-align s* ≤ *align-td-wo-align t*
⟨*proof*⟩

lemma (**in** *c-type*) *align-td-wo-align-uinfo*:
align-td-wo-align (typ-uinfo-t TYPE('a)) = align-td-wo-align (typ-info-t TYPE('a))
⟨*proof*⟩

lemma (**in** *c-type*) *align-td-uinfo*:
align-td (typ-uinfo-t TYPE('a)) = align-td (typ-info-t TYPE('a))
⟨*proof*⟩

lemma *align-td-export-uinfo*: *align-td (export-uinfo t) = align-td t*
⟨*proof*⟩

lemma (**in** *mem-type*) *align-field-uinfo*:
align-field (typ-uinfo-t TYPE('a)) = align-field (typ-info-t TYPE('a))
⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-aligned-mono'*:
field-lookup (typ-uinfo-t TYPE('a)) f 0 = Some (typ-uinfo-t TYPE('b),n) ⟹
ptr-aligned (p::'a ptr) ⟶ *ptr-aligned (Ptr (&(p→f))::'b::mem-type ptr)*
⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-aligned-mono*:

guard-mono (*ptr-aligned*::'*a ptr-guard*) (*ptr-aligned*::'*b*::*mem-type ptr-guard*)
⟨*proof*⟩

lemma (**in** *wf-type*) *wf-desc-typ-tag* [*simp*]:
wf-desc (*typ-uinfo-t TYPE*('*a*))
⟨*proof*⟩

lemma (**in** *c-type*) *sft1'*:
sub-field-update-t (*f*#*fs*) *p* (*v*::'*a*) *s* =
(*let s'* = *sub-field-update-t fs p* (*v*::'*a*) *s in*
s'(*Ptr* &(*p*→*f*) ↦ *from-bytes* (*access-ti*$_0$ (*field-typ TYPE*('*a*) *f*) *v*))) |`
dom (*s*::'*b*::*c-type typ-heap*)
⟨*proof*⟩

lemma *size-map-td*:
size (*map-td f g t*) = *size t*
size (*map-td-struct f g st*) = *size st*
size-list (*size-dt-tuple size* (λ-. *0*) (λ-. *0*)) (*map-td-list f g ts*) = *size-list* (*size-dt-tuple size* (λ-. *0*) (λ-. *0*)) *ts*
size-dt-tuple size (λ-. *0*) (λ-. *0*) (*map-td-tuple f g x*) = *size-dt-tuple size* (λ-. *0*) (λ-. *0*) *x*
⟨*proof*⟩

lemma *field-names-size'*:
field-names t s ≠ [] ⟶ *size s* ≤ *size* (*t*::('*a*,'*b*) *typ-info*)
field-names-struct st s ≠ [] ⟶ *size s* ≤ *size* (*st*::('*a field-desc*,'*b*) *typ-struct*)
field-names-list ts s ≠ [] ⟶ *size s* ≤ *size-list* (*size-dt-tuple size* (λ-. *0*) (λ-. *0*)) (*ts*::(('*a*,'*b*) *typ-info*,*field-name*,'*b*) *dt-tuple list*)
field-names-tuple x s ≠ [] ⟶ *size s* ≤ *size-dt-tuple size* (λ-. *0*) (λ-. *0*) (*x*::(('*a*,'*b*) *typ-info*,*field-name*,'*b*) *dt-tuple*)
⟨*proof*⟩

lemma *field-names-size*:
f ∈ *set* (*field-names t s*) ⟹ *size s* ≤ *size t*
⟨*proof*⟩

lemma *field-names-size-struct*:
f ∈ *set* (*field-names-struct st s*) ⟹ *size s* ≤ *size st*
⟨*proof*⟩

lemma *field-names-Some3*:
∀ *f m s n*. *field-lookup* (*t*::('*a*,'*b*) *typ-info*) *f m* = *Some* (*s*,*n*) ⟶
f ∈ *set* (*field-names t* (*export-uinfo s*))
∀ *f m s n*. *field-lookup-struct* (*st*::('*a field-desc*,'*b*) *typ-struct*) *f m* = *Some* (*s*,*n*) ⟶
f ∈ *set* (*field-names-struct st* (*export-uinfo s*))
∀ *f m s n*. *field-lookup-list* (*ts*::(('*a*,'*b*) *typ-info*,*field-name*,'*b*) *dt-tuple list*) *f m* =

Some (*s*,*n*) ⟶
    f ∈ *set* (*field-names-list ts* (*export-uinfo s*))
  ∀ *f m s n*. *field-lookup-tuple* (*x*::((′*a*,′*b*) *typ-info*,*field-name*,′*b*) *dt-tuple*) *f m* = *Some* (*s*,*n*) ⟶
    f ∈ *set* (*field-names-tuple x* (*export-uinfo s*))
  ⟨*proof*⟩

lemma *field-names-SomeD3*:
  field-lookup (*t*::(′*a*,′*b*) *typ-info*) *f m* = *Some* (*s*,*n*) ⟹ *f* ∈ *set* (*field-names t* (*export-uinfo s*))
  ⟨*proof*⟩

lemma *empty-not-in-field-names* [*simp*]:
  [] ∉ *set* (*field-names-tuple x s*)
  ⟨*proof*⟩

lemma *empty-not-in-field-names-list* [*simp*]:
  [] ∉ *set* (*field-names-list ts s*)
  ⟨*proof*⟩

lemma *empty-not-in-field-names-struct* [*simp*]:
  [] ∉ *set* (*field-names-struct st s*)
  ⟨*proof*⟩

lemma *field-names-Some*:
  ∀ *m f*. *f* ∈ *set* (*field-names* (*t*::(′*a*,′*b*) *typ-info*) *s*) ⟶ (*field-lookup t f m* ≠ *None*)
  ∀ *m f*. *f* ∈ *set* (*field-names-struct* (*st*::(′*a field-desc*,′*b*) *typ-struct*) *s*) ⟶ (*field-lookup-struct st f m* ≠ *None*)
  ∀ *m f*. *f* ∈ *set* (*field-names-list* (*ts*::((′*a*,′*b*) *typ-info*,*field-name*,′*b*) *dt-tuple list*) *s*) ⟶ (*field-lookup-list ts f m* ≠ *None*)
  ∀ *m f*. *f* ∈ *set* (*field-names-tuple* (*x*::((′*a*,′*b*) *typ-info*,*field-name*,′*b*) *dt-tuple*) *s*) ⟶ (*field-lookup-tuple x f m* ≠ *None*)
⟨*proof*⟩

lemma *dt-snd-field-names-list-simp* [*simp*]:
  f ∉ *dt-snd* ‘ *set xs* ⟹ ¬ *f*#*fs* ∈ *set* (*field-names-list xs s*)
  ⟨*proof*⟩

lemma *field-names-Some2*:
  ∀ *m f*. *wf-desc t* ⟶ *f* ∈ *set* (*field-names* (*t*::(′*a*,′*b*) *typ-info*) *s*) ⟶
    (∃ *n k*. *field-lookup t f m* = *Some* (*k*,*n*) ∧ *export-uinfo k* = *s*)
  ∀ *m f*. *wf-desc-struct st* ⟶ *f* ∈ *set* (*field-names-struct* (*st*::(′*a field-desc*,′*b*) *typ-struct*) *s*) ⟶
    (∃ *n k*. *field-lookup-struct st f m* = *Some* (*k*,*n*) ∧ *export-uinfo k* = *s*)
  ∀ *m f*. *wf-desc-list ts* ⟶ *f* ∈ *set* (*field-names-list* (*ts*::((′*a*,′*b*) *typ-info*,*field-name*,′*b*) *dt-tuple list*) *s*) ⟶
    (∃ *n k*. *field-lookup-list ts f m* = *Some* (*k*,*n*) ∧ *export-uinfo k* = *s*)
  ∀ *m f*. *wf-desc-tuple x* ⟶ *f* ∈ *set* (*field-names-tuple* (*x*::((′*a*,′*b*) *typ-info*,*field-name*,′*b*)

dt-tuple) *s*) ⟶
(∃ *n k*. *field-lookup-tuple x f m* = *Some* (*k*,*n*) ∧ *export-uinfo k* = *s*)
⟨*proof*⟩

lemma *field-names-SomeD2*:
⟦ *f* ∈ *set* (*field-names* (*t*::('*a*,'*b*) *typ-info*) *s*); *wf-desc t* ⟧ ⟹
(∃ *n k*. *field-lookup t f m* = *Some* (*k*,*n*) ∧ *export-uinfo k* = *s*)
⟨*proof*⟩

lemma *field-names-SomeD*:
f ∈ *set* (*field-names* (*t*::('*a*,'*b*) *typ-info*) *s*) ⟹ (*field-lookup t f m* ≠ *None*)
⟨*proof*⟩

lemma *lift-t-sub-field-update'* [*rule-format*]:
⟦ *d*,*g'* ⊨$_t$ *p*; ¬ (*TYPE*('*a*) <$_\tau$ *TYPE*('*b*)) ⟧ ⟹ *fs-consistent fs TYPE*('*a*) *TYPE*('*b*) ⟶
(∀ *K*. *K* = *UNIV* − (((*field-offset-footprint p* (*field-names* (*typ-info-t TYPE*('*a*)) (*typ-uinfo-t TYPE*('*b*))))) − (*field-offset-footprint p fs*)) ⟶
lift-t g (*heap-update p* (*v*::'*a*::*mem-type*) *h*,*d*) |` *K* =
sub-field-update-t fs p v ((*lift-t g* (*h*,*d*))::'*b*::*mem-type typ-heap*) |` *K*)
⟨*proof*⟩

lemma *lift-t-sub-field-update*:
⟦ *d*,*g'* ⊨$_t$ *p*; ¬ (*TYPE*('*a*) <$_\tau$ *TYPE*('*b*))⟧ ⟹
lift-t g (*heap-update p* (*v*::'*a*::*mem-type*) *h*,*d*) =
sub-field-update-t (*field-names* (*typ-info-t TYPE*('*a*)) (*typ-uinfo-t TYPE*('*b*))) *p v*
((*lift-t g* (*h*,*d*))::'*b*::*mem-type typ-heap*)
⟨*proof*⟩

lemma *lift-t-field-ind*:
⟦ *d*,*g'* ⊨$_t$ (*p*::'*b*::*mem-type ptr*); *d*,*ga* ⊨$_t$ (*q*::'*b ptr*);
field-lookup (*typ-info-t TYPE*('*b*::*mem-type*)) *f* 0 = *Some* (*a*,*ba*);
field-lookup (*typ-info-t TYPE*('*b*::*mem-type*)) *z* 0 = *Some* (*c*,*da*) ;
size-td a = *size-of TYPE*('*a*); *size-td c* = *size-of TYPE*('*c*);
¬ *f* ≤ *z*; ¬ *z* ≤ *f* ⟧ ⟹
lift-t g (*heap-update* (*Ptr* (&(*p*→*f*))) (*v*::'*a*::*mem-type*) *h*,*d*) (*Ptr* (&(*q*→*z*))) =
((*lift-t g* (*h*,*d*) (*Ptr* (&(*q*→*z*))))::'*c*::*c-type option*)
⟨*proof*⟩

lemma (**in** *c-type*) *uvt1'*:

update-value-t (*f*#*fs*) *v* (*w*::'*b*) *x* = (*if x*=*field-offset TYPE*('*b*) *f then*
update-ti-t (*field-typ TYPE*('*b*) *f*) (*to-bytes-p* (*v*::'*a*)) (*w*::'*b*::*c-type*) *else update-value-t fs v w x*)
⟨*proof*⟩

lemma (**in** *c-type*) *field-typ-self* [*simp*]:
field-typ TYPE('*a*) [] = *typ-info-t TYPE*('*a*)
⟨*proof*⟩

lemma (**in** *mem-type*) *field-of-t-less-size*:
field-of-t (*p*::'*a ptr*) (*x*::'*b*::*c-type ptr*) ⟹
unat (*ptr-val p* − *ptr-val x*) < *size-of TYPE*('*b*)
⟨*proof*⟩

lemma *unat-minus*:
x ≠ *0* ⟹ *unat* (− (*x*::*addr*)) = *addr-card* − *unat x*
⟨*proof*⟩

lemma (**in** *mem-type*) *field-of-t-nmem*:
⟦ *field-of-t p q*; *ptr-val p* ≠ *ptr-val* (*q*::'*b*::*mem-type ptr*) ⟧ ⟹
ptr-val q ∉ {*ptr-val* (*p*::'*a ptr*)..+*size-of TYPE*('*a*)}
⟨*proof*⟩

lemma (**in** *mem-type*) *field-of-t-init-neq-disjoint*:
field-of-t p (*x*::'*b*::*mem-type ptr*) ⟹
{*ptr-val* (*p*::'*a ptr*)..+*size-of TYPE*('*a*)} ∩
{*ptr-val x*..+*unat* (*ptr-val p* − *ptr-val x*)} = {}
⟨*proof*⟩

lemma (**in** *mem-type*) *field-of-t-final-neq-disjoint*:
field-of-t (*p*::'*a ptr*) (*x*::'*b ptr*) ⟹ {*ptr-val p*..+*size-of TYPE*('*a*)} ∩
{*ptr-val p* + *of-nat* (*size-of TYPE*('*a*))..+*size-of TYPE*('*b*::*mem-type*) −
(*unat* (*ptr-val p* − *ptr-val x*) + *size-of TYPE*('*a*))} = {}
⟨*proof*⟩

lemma (**in** *mem-type*) *h-val-super-update-bs*:
field-of-t p x ⟹ *h-val* (*heap-update p* (*v*::'*a*) *h*) (*x*::'*b*::*mem-type ptr*) =
from-bytes (*super-update-bs* (*to-bytes v* (*heap-list h* (*size-of TYPE*('*a*)) (*ptr-val p*))) (*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val x*)) (*unat* (*ptr-val p* − *ptr-val x*)))
⟨*proof*⟩

lemma (**in** *mem-type*) *h-val-super-update-bs-padding*:
field-of-t p x ⟹ *length bs* = *size-of TYPE*('*a*) ⟹ *h-val* (*heap-update-padding p* (*v*::'*a*) *bs h*) (*x*::'*b*::*mem-type ptr*) =
from-bytes (*super-update-bs* (*to-bytes v bs*) (*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val x*)) (*unat* (*ptr-val p* − *ptr-val x*)))
⟨*proof*⟩

lemma (**in** *c-type*) *update-field-update'*:

$n \in (\lambda f.\ \textit{field-offset}\ \textit{TYPE}('b)\ f)\ `\ \textit{set fs} \Longrightarrow$
$(\exists f.\ \textit{update-value-t fs}\ (v::'a)\ (v'::'b::\textit{c-type})\ n =$
$\textit{field-update}\ (\textit{field-desc}\ (\textit{field-typ}\ \textit{TYPE}('b)\ f))\ (\textit{to-bytes-p}\ v)\ v' \wedge f \in \textit{set}$
$\textit{fs} \wedge n = \textit{field-offset}\ \textit{TYPE}('b)\ f)$
⟨*proof*⟩

lemma (**in** *c-type*) *update-field-update*:
$\textit{field-of-t}\ (p::'a\ \textit{ptr})\ (x::'b\ \textit{ptr}) \Longrightarrow$
$\exists f.\ \textit{update-value-t}\ (\textit{field-names}\ (\textit{typ-info-t}\ \textit{TYPE}('b))\ (\textit{typ-uinfo-t}\ \textit{TYPE}('a)))$
$(v::'a)$
$(v'::'b::\textit{mem-type})\ (\textit{unat}\ (\textit{ptr-val}\ p - \textit{ptr-val}\ x)) =$
$\textit{field-update}\ (\textit{field-desc}\ (\textit{field-typ}\ \textit{TYPE}('b)\ f))\ (\textit{to-bytes-p}\ v)\ v' \wedge$
$f \in \textit{set}\ (\textit{field-names}\ (\textit{typ-info-t}\ \textit{TYPE}('b))\ (\textit{typ-uinfo-t}\ \textit{TYPE}('a))) \wedge$
$\textit{unat}\ (\textit{ptr-val}\ p - \textit{ptr-val}\ x) = \textit{field-offset}\ \textit{TYPE}('b)\ f$
⟨*proof*⟩

lemma *length-fa-ti*:
$[\![\ \textit{wf-fd}\ s;\ \textit{length}\ bs = \textit{size-td}\ s\ ]\!] \Longrightarrow \textit{length}\ (\textit{access-ti}\ s\ w\ bs) = \textit{size-td}\ s$
⟨*proof*⟩

lemma *fa-fu-v*:
$[\![\ \textit{wf-fd}\ s;\ \textit{length}\ bs = \textit{size-td}\ s;\ \textit{length}\ bs' = \textit{size-td}\ s\ ]\!] \Longrightarrow$
$\textit{access-ti}\ s\ (\textit{update-ti-t}\ s\ bs\ v)\ bs' = \textit{access-ti}\ s\ (\textit{update-ti-t}\ s\ bs\ w)\ bs'$
⟨*proof*⟩

lemma *fu-fa*:
$[\![\ \textit{wf-fd}\ s;\ \textit{length}\ bs = \textit{size-td}\ s\ ]\!] \Longrightarrow$
$\textit{update-ti-t}\ s\ (\textit{access-ti}\ s\ v\ bs)\ v = v$
⟨*proof*⟩

lemma *fu-fu*:
$[\![\ \textit{wf-fd}\ s;\ \textit{length}\ bs = \textit{length}\ bs'\ ]\!] \Longrightarrow$
$\textit{update-ti-t}\ s\ bs\ (\textit{update-ti-t}\ s\ bs'\ v) = \textit{update-ti-t}\ s\ bs\ v$
⟨*proof*⟩

lemma *fu-fa'-rpbs*:
$[\![\ \textit{export-uinfo}\ s = \textit{export-uinfo}\ t;\ \textit{length}\ bs = \textit{size-td}\ s;\ \textit{wf-fd}\ s;\ \textit{wf-fd}\ t\ ]\!] \Longrightarrow$
$\textit{update-ti-t}\ s\ (\textit{access-ti}\ t\ v\ bs)\ w = \textit{update-ti-t}\ s\ (\textit{access-ti}_0\ t\ v)\ w$
⟨*proof*⟩

lemma *lift-t-super-field-update*:
$[\![\ d,g' \models_t p;\ \textit{TYPE}('a) \leq_\tau \textit{TYPE}('b)\ ]\!] \Longrightarrow$
$\textit{lift-t}\ g\ (\textit{heap-update}\ p\ (v::'a::\textit{mem-type})\ h,d) =$
$\textit{super-field-update-t}\ p\ v\ ((\textit{lift-t}\ g\ (h,d))::'b::\textit{mem-type typ-heap})$
⟨*proof*⟩

lemma *lift-t-super-field-update-padding*:
$[\![\ d,g' \models_t p;\ \textit{TYPE}('a) \leq_\tau \textit{TYPE}('b);\ \textit{length}\ bs = \textit{size-of}\ \textit{TYPE}('a)\ ]\!] \Longrightarrow$

lift-t g (heap-update-padding p (v::'a::mem-type) bs h,d) =
super-field-update-t p v ((lift-t g (h,d))::'b::mem-type typ-heap)
⟨*proof*⟩

lemma *field-names-same*:
k = export-uinfo ti ⟹ field-names ti k = [[]]
⟨*proof*⟩

lemma *lift-t-heap-update*:
d,g ⊨$_t$ p ⟹ lift-t g (heap-update p v h,d) =
((lift-t g (h,d)) (p ↦ (v::'a::mem-type)))
⟨*proof*⟩

lemma *field-names-disj*:
typ-uinfo-t TYPE('a::c-type) ⊥$_t$ typ-uinfo-t TYPE('b::mem-type) ⟹
field-names (typ-info-t TYPE('b)) (typ-uinfo-t TYPE('a)) = []
⟨*proof*⟩

lemma *lift-t-heap-update-same*:
⟦ *d,g' ⊨$_t$ (p::'b::mem-type ptr); typ-uinfo-t TYPE('a) ⊥$_t$ typ-uinfo-t TYPE('b)* ⟧
⟹
lift-t g (heap-update p v h,d) = (lift-t g (h,d))::'a::mem-type typ-heap)
⟨*proof*⟩

lemma *lift-heap-update*:
⟦ *d,g ⊨$_t$ (p::'a ptr); d,g' ⊨$_t$ q* ⟧ *⟹ lift (heap-update p v h) q =*
((lift h)(p := (v::'a::mem-type))) q
⟨*proof*⟩

lemma (**in** *mem-type*) *lift-heap-update-p* [*simp*]:
lift (heap-update p v h) p = (v::'a)
⟨*proof*⟩

lemma *lift-heap-update-same*:
⟦ *ptr-val p ≠ ptr-val q; d,g ⊨$_t$ (p::'a::mem-type ptr);*
d,g' ⊨$_t$ (q::'b::mem-type ptr); peer-typ TYPE('a) TYPE('b) ⟧ *⟹*
lift (heap-update p v h) q = lift h q
⟨*proof*⟩

lemma *lift-heap-update-same-type*:
fixes *p::'a::mem-type ptr* **and** *q::'b::mem-type ptr*
assumes *valid*: *d,g ⊨$_t$ p d,g' ⊨$_t$ q*
assumes *type*: *typ-uinfo-t TYPE('a) ⊥$_t$ typ-uinfo-t TYPE('b)*
shows *lift (heap-update p v h) q = lift h q*
⟨*proof*⟩

lemma *lift-heap-update-same-ptr-coerce*:
⟦ *ptr-val q ≠ ptr-val p;*
d,g ⊨$_t$ (ptr-coerce (q::'b::mem-type ptr))::'c::mem-type ptr);

$d,g' \models_t (p::'a::mem\text{-}type\ ptr)$;
$size\text{-}of\ TYPE('b) = size\text{-}of\ TYPE('c)$; $peer\text{-}typ\ TYPE('a)\ TYPE('c)$ ⟧ ⟹
$lift\ (heap\text{-}update\ q\ v\ h)\ p = lift\ h\ p$
⟨*proof*⟩

lemma *heap-footprintI*:
⟦ $valid\text{-}footprint\ d\ y\ t$; $x \in \{y..+size\text{-}td\ t\}$ ⟧ ⟹ $x \in heap\text{-}footprint\ d\ t$
⟨*proof*⟩

lemma *valid-heap-footprint*:
$valid\text{-}footprint\ d\ x\ t \Longrightarrow x \in heap\text{-}footprint\ d\ t$
⟨*proof*⟩

lemma *heap-valid-footprint*:
$x \in heap\text{-}footprint\ d\ t \Longrightarrow \exists\, y.\ valid\text{-}footprint\ d\ y\ t \wedge x \in \{y\} \cup \{y..+size\text{-}td\ t\}$
⟨*proof*⟩

lemma *heap-footprint-Some*:
$x \in heap\text{-}footprint\ d\ t \Longrightarrow d\ x \neq (False,Map.empty)$
⟨*proof*⟩

lemma *dom-tll-empty* [*simp*]:
$dom\text{-}tll\ p\ [] = \{\}$
⟨*proof*⟩

lemma *dom-s-upd* [*simp*]:
$dom\text{-}s\ (\lambda b.\ if\ b = p\ then\ (True,a)\ else\ d\ b) =$
$(dom\text{-}s\ d - \{(p,y) \mid y.\ True\}) \cup \{(p,SIndexVal)\} \cup \{(p,SIndexTyp\ n) \mid n.\ a\ n \neq None\}$
⟨*proof*⟩

lemma *dom-tll-cons* [*simp*]:
$dom\text{-}tll\ p\ (x\#xs) = dom\text{-}tll\ (p + 1)\ xs \cup \{(p,SIndexVal)\} \cup \{(p,SIndexTyp\ n) \mid n.\ x\ n \neq None\}$
⟨*proof*⟩

lemma *one-plus-x-zero*:
$(1::addr) + of\text{-}nat\ x = 0 \Longrightarrow x \geq addr\text{-}card - 1$
⟨*proof*⟩

lemma *htd-update-list-dom* [*rule-format*, *simp*]:
$length\ xs < addr\text{-}card \longrightarrow (\forall\, p\ d.\ dom\text{-}s\ (htd\text{-}update\text{-}list\ p\ xs\ d) = dom\text{-}s\ d \cup dom\text{-}tll\ p\ xs)$
⟨*proof*⟩

lemma *htd-update-list-same*:
shows $\bigwedge h\ p\ k.\ [\![\ 0 < k;\ k \leq \mathit{addr\text{-}card} - \mathit{length}\ v\ ]\!] \Longrightarrow$
$(\mathit{htd\text{-}update\text{-}list}\ (p + \mathit{of\text{-}nat}\ k)\ v)\ h\ p = h\ p$
⟨*proof*⟩

lemma *htd-update-list-index* [*rule-format*]:
$\forall\, p\ d.\ \mathit{length}\ xs < \mathit{addr\text{-}card} \longrightarrow x \in \{p..+\mathit{length}\ xs\} \longrightarrow$
$\mathit{htd\text{-}update\text{-}list}\ p\ xs\ d\ x = (\mathit{True},\ \mathit{snd}\ (d\ x)\ ++\ (xs\ !\ (\mathit{unat}\ (x - p))))$
⟨*proof*⟩

lemma (**in** *c-type*) *typ-slices-length* [*simp*]:
$\mathit{length}\ (\mathit{typ\text{-}slices}\ \mathit{TYPE}('a)) = \mathit{size\text{-}of}\ \mathit{TYPE}('a)$
⟨*proof*⟩

lemma (**in** *c-type*) *typ-slices-index* [*simp*]:
$n < \mathit{size\text{-}of}\ \mathit{TYPE}('a) \Longrightarrow \mathit{typ\text{-}slices}\ \mathit{TYPE}('a)\ !\ n =$
$\mathit{list\text{-}map}\ (\mathit{typ\text{-}slice\text{-}t}\ (\mathit{typ\text{-}uinfo\text{-}t}\ \mathit{TYPE}('a))\ n)$
⟨*proof*⟩

lemma (**in** *c-type*) *empty-not-in-typ-slices* [*simp*]:
$\mathit{Map.empty} \notin \mathit{set}\ (\mathit{typ\text{-}slices}\ \mathit{TYPE}('a))$
⟨*proof*⟩

lemma (**in** *c-type*) *dom-tll-s-footprint* [*simp*]:
$\mathit{dom\text{-}tll}\ (\mathit{ptr\text{-}val}\ p)\ (\mathit{typ\text{-}slices}\ \mathit{TYPE}('a)) = \mathit{s\text{-}footprint}\ (p::'a\ \mathit{ptr})$
⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-retyp-dom* [*simp*]:
$\mathit{dom\text{-}s}\ (\mathit{ptr\text{-}retyp}\ (p::'a\ \mathit{ptr})\ d) = \mathit{dom\text{-}s}\ d \cup \mathit{s\text{-}footprint}\ p$
⟨*proof*⟩

lemma *dom-s-empty-htd* [*simp*]:
$\mathit{dom\text{-}s}\ \mathit{empty\text{-}htd} = \{\}$
⟨*proof*⟩

lemma *dom-s-nempty*:
$d\ x \neq (\mathit{False},\ \mathit{Map.empty}) \Longrightarrow \exists\, k.\ (x,k) \in \mathit{dom\text{-}s}\ d$
⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-retyp-None*:
$x \notin \{\mathit{ptr\text{-}val}\ p..+\mathit{size\text{-}of}\ \mathit{TYPE}('a)\} \Longrightarrow$
$\mathit{ptr\text{-}retyp}\ (p::'a\ \mathit{ptr})\ \mathit{empty\text{-}htd}\ x = (\mathit{False},\mathit{Map.empty})$
⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-retyp-footprint*:
$x \in \{\mathit{ptr\text{-}val}\ p..+\mathit{size\text{-}of}\ \mathit{TYPE}('a)\} \Longrightarrow$
$\mathit{ptr\text{-}retyp}\ (p::'a\ \mathit{ptr})\ d\ x =$

(*True*,*snd* (*d x*) ++ *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*('*a*)) (*unat* (*x* − *ptr-val p*))))

⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-retyp-Some*:
ptr-retyp (*p*::'*a ptr*) *d* (*ptr-val p*) =
(*True*,*snd* (*d* (*ptr-val p*)) ++ *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*('*a*)) *0*))

⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-retyp-Some2*:
x ∈ {*ptr-val* (*p*::'*a ptr*)..+*size-of TYPE*('*a*)} ⟹ *ptr-retyp p d x* ≠ (*False*,*Map.empty*)

⟨*proof*⟩

lemma *snd-empty-htd* [*simp*]:
snd (*empty-htd x*) = *Map.empty*

⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-retyp-d-empty*:
x ∈ {*ptr-val* (*p*::'*a ptr*)..+*size-of TYPE*('*a*)} ⟹
(∀ *d*. *fst* (*ptr-retyp p d x*)) ∧
snd (*ptr-retyp p d x*) = *snd* (*d x*) ++ *snd* (*ptr-retyp p empty-htd x*)

⟨*proof*⟩

lemma *unat-minus-abs*:
x ≠ *y* ⟹ *unat* ((*x*::*addr*) − *y*) = *addr-card* − *unat* (*y* − *x*)

⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-retyp-d*:
x ∉ {*ptr-val* (*p*::'*a ptr*)..+*size-of TYPE*('*a*)} ⟹
ptr-retyp p d x = *d x*

⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-retyp-valid-footprint-disjoint*:
⟦ *valid-footprint d p s*; {*p*..+*size-td s*} ∩ {*ptr-val q*..+*size-of TYPE*('*a*)} = {} ⟧
⟹ *valid-footprint* (*ptr-retyp* (*q*::'*a ptr*) *d*) *p s*

⟨*proof*⟩

lemma *ptr-retyp-disjoint*:
⟦ *d*,*g* ⊨$_t$ (*q*::'*b*::*mem-type ptr*); {*ptr-val p*..+*size-of TYPE*('*a*)} ∩
{*ptr-val q*..+*size-of TYPE*('*b*)} = {} ⟧ ⟹
ptr-retyp (*p*::'*a*::*mem-type ptr*) *d*,*g* ⊨$_t$ *q*

⟨*proof*⟩

lemma *ptr-retyp-d-fst*:
(*x*,*SIndexVal*) ∉ *s-footprint* (*p*::'*a*::*mem-type ptr*) ⟹ *fst* (*ptr-retyp p d x*) = *fst* (*d x*)

⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-retyp-d-eq-fst*:

fst (*ptr-retyp p d x*) =
(*if x* ∈ {*ptr-val* (*p*::'*a ptr*)..+*size-of TYPE*('*a*)} *then True else fst* (*d x*))
⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-retyp-d-eq-snd*:
snd (*ptr-retyp p d x*) =
(*if x* ∈ {*ptr-val* (*p*::'*a ptr*)..+*size-of TYPE*('*a*)}
then snd (*d x*) ++ *snd* (*ptr-retyp p empty-htd x*)
else snd (*d x*))
⟨*proof*⟩

lemma *lift-state-ptr-retyp-d-empty*:
x ∈ {*ptr-val* (*p*::'*a*::*mem-type ptr*)..+*size-of TYPE*('*a*)} ⟹
lift-state (*h*,*ptr-retyp p d*) (*x*,*k*) =
(*lift-state* (*h*,*d*) ++ *lift-state* (*h*,*ptr-retyp p empty-htd*)) (*x*,*k*)
⟨*proof*⟩

lemma *lift-state-ptr-retyp-d*:
x ∉ {*ptr-val* (*p*::'*a*::*mem-type ptr*)..+*size-of TYPE*('*a*)} ⟹
lift-state (*h*,*ptr-retyp p d*) (*x*,*k*) = *lift-state* (*h*,*d*) (*x*,*k*)
⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-retyp-valid-footprint*:
valid-footprint (*ptr-retyp p d*) (*ptr-val* (*p*::'*a ptr*)) (*typ-uinfo-t TYPE*('*a*))
⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-retyp-h-t-valid*:
g p ⟹ *ptr-retyp p d*,*g* ⊨$_t$ (*p*::'*a ptr*)
⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-retyp-s-valid*:
g p ⟹ *lift-state* (*h*,*ptr-retyp p d*),*g* ⊨$_s$ (*p*::'*a ptr*)
⟨*proof*⟩

lemma (**in** *mem-type*) *lt-size-of-unat-simps*:
k < *size-of TYPE*('*a*) ⟹ *unat* ((*of-nat k*)::*addr*) < *size-of TYPE*('*a*)
⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-retyp-h-t-valid-same*:
⟦ *d*,*g* ⊨$_t$ (*p*::'*a ptr*); *x* ∈ {*ptr-val p*..+*size-of TYPE*('*a*)} ⟧ ⟹
snd (*ptr-retyp p d x*) ⊆$_m$ *snd* (*d x*)
⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-retyp-ptr-safe* [*simp*]:
ptr-safe p (*ptr-retyp* (*p*::'*a ptr*) *d*)
⟨*proof*⟩

lemma (**in** *mem-type*) *lift-state-ptr-retyp-restrict*:

```
  (lift-state (h, ptr-retyp p d) |` {(x,k). x ∈ {ptr-val p..+size-of TYPE('a)}}) =
     (lift-state (h,d) |` {(x,k). x ∈ {ptr-val p..+size-of TYPE('a)}}) ++
     lift-state (h, ptr-retyp (p::'a ptr) empty-htd) (is ?x = ?y)
⟨proof⟩


lemmas typ-simps = lift-t-heap-update lift-t-heap-update-same lift-heap-update
                   lift-t-h-t-valid h-t-valid-ptr-safe lift-t-ptr-safe lift-lift-t lift-t-lift
                   tag-disj-def typ-tag-le-def  typ-uinfo-t-def

declare field-desc-def [simp del]

lemma field-fd:
  field-fd (t::'a::c-type itself) n =
     (case field-lookup (typ-info-t TYPE('a)) n 0 of
        None ⇒ field-desc (fst (the (None::('a xtyp-info × nat) option)))
      | Some x ⇒ field-desc (fst x))
  ⟨proof⟩

declare field-desc-def [simp add]

lemma (in mem-type) super-field-update-lookup:
  assumes field-lookup (typ-info-t TYPE('b)) f 0 = Some (s,n)
    and typ-uinfo-t TYPE('a) = export-uinfo s
    and lift-t g h p = Some v'
  shows super-field-update-t (Ptr (&(p→f))) (v::'a) (lift-t g h::'b::mem-type typ-heap)
=
          (lift-t g h)(p ↦ field-update (field-desc s) (to-bytes-p v) v')
⟨proof⟩



lemmas typ-rewrs =
  lift-lift-t
  lift-t-heap-update
  lift-t-heap-update-same
  lift-t-heap-update [OF lift-t-h-t-valid]
  lift-t-heap-update-same [OF lift-t-h-t-valid]
  lift-lift-t [OF lift-t-h-t-valid]

end



theory Separation-UMM
imports TypHeap
begin
```

type-synonym $('a,'b)$ *map-assert* $= ('a \rightharpoonup 'b) \Rightarrow$ *bool*
type-synonym *heap-assert* $=$ (*addr* $\times$ *s-heap-index*,*s-heap-value*) *map-assert*

definition *sep-emp* :: $('a,'b)$ *map-assert* ($\langle\Box\rangle$) **where**
sep-emp $\equiv$ $(=)$ *Map.empty*

definition *sep-true* :: $('a,'b)$ *map-assert* **where**
sep-true $\equiv \lambda s.$ *True*

definition *sep-false* :: $('a,'b)$ *map-assert* **where**
sep-false $\equiv \lambda s.$ *False*

definition
sep-conj :: $('a,'b)$ *map-assert* $\Rightarrow$ $('a,'b)$ *map-assert* $\Rightarrow$ $('a,'b)$ *map-assert* (**infixr** $\langle\wedge^*\rangle$ *35*)
where
$P \wedge^* Q \equiv \lambda s.\ \exists\, s_0\ s_1.\ s_0 \perp s_1 \wedge s = s_1 +\!\!+ s_0 \wedge P\ s_0 \wedge Q\ s_1$

definition
sep-impl :: $('a,'b)$ *map-assert* $\Rightarrow$ $('a,'b)$ *map-assert* $\Rightarrow$ $('a,'b)$ *map-assert* (**infixr** $\langle\longrightarrow^*\rangle$ *25*)
where
$x \longrightarrow^* y \equiv \lambda s.\ \forall\, s'.\ s \perp s' \wedge x\ s' \longrightarrow y\ (s +\!\!+ s')$

definition
singleton :: *'a*::*c-type ptr* $\Rightarrow$ *'a* $\Rightarrow$ *heap-mem* $\Rightarrow$ *heap-typ-desc* $\Rightarrow$ *heap-state*
where
singleton p v h d $\equiv$ *lift-state* (*heap-update p v h*,*d*) |` *s-footprint p*

Like in Separation.thy, these arrows are defined using bsub and esub but have an *input* syntax abbreviation with just sub. Why? Because if sub is the only way, people write things like $p \mapsto^i_{(f\ x\ y)}\ v$ instead of $p \mapsto^i_{(f\ x\ y)}\ v$. We preserve the sub syntax though, because esub and bsub are a pain to type.

definition
sep-map :: *'a*::*c-type ptr* $\Rightarrow$ *'a ptr-guard* $\Rightarrow$ *'a* $\Rightarrow$ *heap-assert*
($\langle$($\langle$*open-block notation*=$\langle$*mixfix sep-map*$\rangle$- $\mapsto$_ -)$\rangle$ [*56*,*0*,*51*] *56*)
where
$p \mapsto_g v \equiv \lambda s.$ *lift-typ-heap g s p* $=$ *Some v* $\wedge$ *dom s* $=$ *s-footprint p* $\wedge$ *wf-heap-val s*

notation (*input*)
sep-map ($\langle$($\langle$*open-block notation*=$\langle$*mixfix sep-map*$\rangle$- $\mapsto$_ -)$\rangle$ [*56*,*1000*,*51*] *56*)

definition
sep-map-any :: *'a* ::*c-type ptr* $\Rightarrow$ *'a ptr-guard* $\Rightarrow$ *heap-assert*
($\langle$($\langle$*open-block notation*=$\langle$*mixfix sep-map-any*$\rangle$- $\mapsto$_ −)$\rangle$ [*56*,*0*] *56*)
where

$p \mapsto_g - \equiv \lambda s.\ \exists\, v.\ (p \mapsto_g v)\ s$

notation (*input*)
sep-map-any (‹(‹*open-block notation=‹mixfix sep-map-any››*- ↦_ −)› [*56*,*0*] *56*)

definition
sep-map′ :: *'a*::*c-type ptr* ⇒ *'a ptr-guard* ⇒ *'a* ⇒ *heap-assert*
(‹(‹*open-block notation=‹mixfix sep-map′››*- ↪_ -)› [*56*,*0*,*51*] *56*)
where
$p \hookrightarrow_g v \equiv (p \mapsto_g v) \wedge^* \textit{sep-true}$

notation (*input*)
sep-map′ (‹(‹*open-block notation=‹mixfix sep-map′››*- ↪_ -)› [*56*,*1000*,*51*] *56*)

definition
sep-map′-any :: *'a* ::*c-type ptr* ⇒ *'a ptr-guard* ⇒ *heap-assert*
(‹(‹*open-block notation=‹mixfix sep-map′-any››*- ↪_ −)› [*56*,*0*] *56*)
where
$p \hookrightarrow_g - \equiv \lambda s.\ \exists\, x.\ (p \hookrightarrow_g x)\ s$

notation (*input*)
sep-map′-any (‹(‹*open-block notation=‹mixfix sep-map′-any››*- ↪_ −)› [*56*,*0*] *56*)

syntax
-sep-assert :: *bool* ⇒ *heap-state* ⇒ *bool*
(‹(‹*open-block notation=‹mixfix assertion›› '(-')*^{sep})› [*0*] *100*)

—-

lemma *sep-empD*:
□ $s \Longrightarrow s = \textit{Map.empty}$
⟨*proof*⟩

lemma *sep-emp-empty* [*simp*]:
□ *Map.empty*
⟨*proof*⟩

lemma *sep-true* [*simp*]:
sep-true s
⟨*proof*⟩

lemma *sep-false* [*simp*]:
¬ *sep-false s*
⟨*proof*⟩

declare *sep-false-def* [*symmetric*, *simp add*]

lemma *singleton-dom′*:
dom (*singleton p* (*v*::*'a*::*mem-type*) *h d*) = *dom* (*lift-state* (*h*,*d*)) ∩ *s-footprint p*
⟨*proof*⟩

lemma *lift-state-dom*:
$d,g \models_t p \Longrightarrow$ *s-footprint* $(p::'a::$*mem-type ptr*$) \subseteq$ *dom* (*lift-state* (h,d))
⟨*proof*⟩

lemma *singleton-dom*:
$d,g \models_t p \Longrightarrow$ *dom* (*singleton* p $(v::'a::$*mem-type*$)$ h d) = *s-footprint* p
⟨*proof*⟩

lemma *wf-heap-val-restrict* [*simp*]:
wf-heap-val $s \Longrightarrow$ *wf-heap-val* $(s$ |` $X)$
⟨*proof*⟩

lemma *singleton-wf-heap-val* [*simp*]:
wf-heap-val (*singleton* p v h d)
⟨*proof*⟩

lemma *h-t-valid-restrict-proj-d*:
⟦ *proj-d* $s,g \models_t p$; $\forall x.\ x \in$ *s-footprint* $p \longrightarrow s\ x = s'\ x$ ⟧ $\Longrightarrow$
proj-d $s',g \models_t p$
⟨*proof*⟩

lemma *s-valid-restrict* [*simp*]:
s |` *s-footprint* $p,g \models_s p = s,g \models_s p$
⟨*proof*⟩

lemma *proj-h-restrict*:
$(x,$*SIndexVal*$) \in X \Longrightarrow$ *proj-h* $(s$ |` $X)$ x = *proj-h* s x
⟨*proof*⟩

lemma *heap-list-s-restrict*:
$(\lambda x.\ (x,$*SIndexVal*$))$ ` $\{p..+n\} \subseteq X \Longrightarrow$ *heap-list-s* $(s$ |` $X)$ n p = *heap-list-s* s n p
⟨*proof*⟩

lemma *lift-typ-heap-restrict* [*simp*]:
lift-typ-heap g $(s$ |` *s-footprint* $p)$ p = *lift-typ-heap* g s p
⟨*proof*⟩

lemma *singleton-s-valid*:
$d,g \models_t p \Longrightarrow$ *singleton* p $(v::'a::$*mem-type*$)$ h $d,g \models_s p$
⟨*proof*⟩

lemma *singleton-lift-typ-heap-Some*:
$d,g \models_t p \Longrightarrow$ *lift-typ-heap* g (*singleton* p v h d) p = *Some* $(v::'a::$*mem-type*$)$
⟨*proof*⟩

lemma *sep-map-g*:
$(p \mapsto_g v)$ $s \Longrightarrow$ g p

⟨*proof*⟩

lemma *sep-map-g-sep-false*:
¬ *g p* ⟹ (*p* ↦$_g$ *v*) = *sep-false*
⟨*proof*⟩

lemma *sep-map-singleton*:
d,*g* ⊨$_t$ *p* ⟹ ((*p*::'*a*::*mem-type ptr*) ↦$_g$ *v*) (*singleton p v h d*)
⟨*proof*⟩

lemma *sep-mapD*:
(*p* ↦$_g$ *v*) *s* ⟹ *lift-typ-heap g s p* = *Some v* ∧
dom s = *s-footprint p* ∧ *wf-heap-val s*
⟨*proof*⟩

lemma *sep-map-lift-typ-heapD*:
(*p* ↦$_g$ *v*) *s* ⟹ *lift-typ-heap g s p* = *Some* (*v*::'*a*::*c-type*)
⟨*proof*⟩

lemma *sep-map-dom-exc*:
(*p* ↦$_g$ (*v*::'*a*::*c-type*)) *s* ⟹ *dom s* = *s-footprint p*
⟨*proof*⟩

lemma *sep-map-inj*:
⟦ (*p* ↦$_g$ (*v*::'*a*::*c-type*)) *s*; (*p* ↦$_h$ *v'*) *s* ⟧ ⟹ *v* = *v'*
⟨*proof*⟩

lemma *sep-map-anyI-exc* [*simp*]:
(*p* ↦$_g$ *v*) *s* ⟹ (*p* ↦$_g$ −) *s*
⟨*proof*⟩

lemma *sep-map-anyD-exc*:
(*p* ↦$_g$ −) *s* ⟹ ∃ *v*. (*p* ↦$_g$ *v*) *s*
⟨*proof*⟩

lemma *sep-map-any-singleton*:
d,*g* ⊨$_t$ *i* ⟹ (*i* ↦$_g$ −) (*singleton i* (*v*::'*a*::*mem-type*) *h d*)
⟨*proof*⟩

lemma *proj-h-heap-merge*:
proj-h (*s* ++ *t*) = (λ*x*. *if* (*x*,*SIndexVal*) ∈ *dom t then proj-h t x else proj-h s x*)
⟨*proof*⟩

lemma *s-valid-heap-merge-right*:
s$_1$,*g* ⊨$_s$ *p* ⟹ *s*$_0$ ++ *s*$_1$,*g* ⊨$_s$ *p*
⟨*proof*⟩

lemma *proj-d-map-add-fst*:
fst (*proj-d* (*s* ++ *t*) *x*) = (*if* (*x*,*SIndexVal*) ∈ *dom t then fst* (*proj-d t x*) *else*

fst (*proj-d s x*))
⟨*proof*⟩

lemma *proj-d-map-add-snd*:
snd (*proj-d* (*s* ++ *t*) *x*) *n* = (*if* (*x*,*SIndexTyp n*) ∈ *dom t then snd* (*proj-d t x*) *n else*
snd (*proj-d s x*) *n*)
⟨*proof*⟩

lemma *proj-d-restrict-map-fst*:
(*x*,*SIndexVal*) ∈ *X* ⟹ *fst* (*proj-d* (*s* |‘ *X*) *x*) = *fst* (*proj-d s x*)
⟨*proof*⟩

lemma *proj-d-restrict-map-snd*:
(*x*,*SIndexTyp n*) ∈ *X* ⟹ *snd* (*proj-d* (*s* |‘ *X*) *x*) *n* = *snd* (*proj-d s x*) *n*
⟨*proof*⟩

lemma *s-valid-heap-merge-right2*:
⟦ *s*$_0$ ++ *s*$_1$,*g* ⊨$_s$ *p*; *s-footprint p* ⊆ *dom s*$_1$ ⟧ ⟹ *s*$_1$,*g* ⊨$_s$ *p*
⟨*proof*⟩

lemma *heap-list-s-heap-merge-right'*:
⟦ *s*$_1$,*g* ⊨$_s$ (*p*::*'a*::*c-type ptr*); *n* ≤ *size-of TYPE*(*'a*) ⟧ ⟹
heap-list-s (*s*$_0$ ++ *s*$_1$) *n* (*ptr-val p* + *of-nat* (*size-of TYPE*(*'a*) − *n*))
= *heap-list-s s*$_1$ *n* (*ptr-val p* + *of-nat* (*size-of TYPE*(*'a*) − *n*))
⟨*proof*⟩

lemma *heap-list-s-heap-merge-right*:
s$_1$,*g* ⊨$_s$ ((*Ptr p*)::*'a*::*c-type ptr*) ⟹
heap-list-s (*s*$_0$ ++ *s*$_1$) (*size-of TYPE*(*'a*)) *p* = *heap-list-s s*$_1$ (*size-of TYPE*(*'a*)) *p*
⟨*proof*⟩

lemma *lift-typ-heap-heap-merge-right*:
lift-typ-heap g s$_1$ *p* = *Some v* ⟹ *lift-typ-heap g* (*s*$_0$ ++ *s*$_1$) (*p*::*'a*::*c-type ptr*) = *Some v*
⟨*proof*⟩

lemma *lift-typ-heap-heap-merge-sep-map*:
(*p* ↦$_g$ *v*) *s*$_1$ ⟹ *lift-typ-heap g* (*s*$_0$ ++ *s*$_1$) *p* = *Some* (*v*::*'a*::*c-type*)
⟨*proof*⟩

lemma *sep-conjI*:
⟦ *P s*$_0$; *Q s*$_1$; *s*$_0$ ⊥ *s*$_1$; *s* = *s*$_1$ ++ *s*$_0$ ⟧ ⟹ (*P* ∧* *Q*) *s*
⟨*proof*⟩

lemma *sep-conjD*:
(*P* ∧* *Q*) *s* ⟹ ∃ *s*$_0$ *s*$_1$. *s*$_0$ ⊥ *s*$_1$ ∧ *s* = *s*$_1$ ++ *s*$_0$ ∧ *P s*$_0$ ∧ *Q s*$_1$
⟨*proof*⟩

lemma *sep-map′I*:
$((p \mapsto_g v) \wedge^* \textit{sep-true})\ s \Longrightarrow (p \hookrightarrow_g v)\ s$
⟨*proof*⟩

lemma *sep-map′D*:
$(p \hookrightarrow_g v)\ s \Longrightarrow ((p \mapsto_g v) \wedge^* \textit{sep-true})\ s$
⟨*proof*⟩

lemma *sep-map′-g-exc*:
$(p \hookrightarrow_g v)\ s \Longrightarrow g\ p$
⟨*proof*⟩

lemma *sep-conj-sep-true*:
$P\ s \Longrightarrow (P \wedge^* \textit{sep-true})\ s$
⟨*proof*⟩

lemma *sep-map-sep-map′-exc* [*simp*]:
$(p \mapsto_g v)\ s \Longrightarrow (p \hookrightarrow_g v)\ s$
⟨*proof*⟩

lemma *sep-conj-true* [*simp*]:
$(\textit{sep-true} \wedge^* \textit{sep-true}) = \textit{sep-true}$
⟨*proof*⟩

lemma *sep-conj-assocD*:
assumes *l*: $((P \wedge^* Q) \wedge^* R)\ s$
shows $(P \wedge^* (Q \wedge^* R))\ s$
⟨*proof*⟩

lemma *sep-conj-com*:
$(P \wedge^* Q) = (Q \wedge^* P)$
⟨*proof*⟩

lemma *sep-conj-false-right* [*simp*]:
$(P \wedge^* \textit{sep-false}) = \textit{sep-false}$
⟨*proof*⟩

lemma *sep-conj-false-left* [*simp*]:
$(\textit{sep-false} \wedge^* P) = \textit{sep-false}$
⟨*proof*⟩

lemma *sep-conj-comD*:
$(P \wedge^* Q)\ s \Longrightarrow (Q \wedge^* P)\ s$
⟨*proof*⟩

lemma *exists-left*:
$(Q \wedge^* (\lambda s.\ \exists x.\ P\ x\ s)) = ((\lambda s.\ \exists x.\ P\ x\ s) \wedge^* Q)$ ⟨*proof*⟩

lemma *sep-conj-assoc*:
$((P \wedge^* Q) \wedge^* R) = (P \wedge^* (Q \wedge^* R))$ (**is** *?x* = *?y*)
⟨*proof*⟩

lemma *sep-conj-left-com*:
$(P \wedge^* (Q \wedge^* R)) = (Q \wedge^* (P \wedge^* R))$ (**is** *?x* = *?y*)
⟨*proof*⟩

lemmas *sep-conj-ac* = *sep-conj-com sep-conj-assoc sep-conj-left-com*

lemma *sep-conj-empty*:
$(P \wedge^* \square) = P$
⟨*proof*⟩

lemma *sep-conj-empty'* [*simp*]:
$(\square \wedge^* P) = P$
⟨*proof*⟩

lemma *sep-conj-true-P* [*simp*]:
(*sep-true* $\wedge^*$ (*sep-true* $\wedge^*$ *P*)) = (*sep-true* $\wedge^*$ *P*)
⟨*proof*⟩

lemma *sep-map'-unfold-exc*:
$(p \hookrightarrow_g v) = ((p \hookrightarrow_g v) \wedge^*$ *sep-true*)
⟨*proof*⟩

lemma *sep-conj-disj*:
$((\lambda s.\ P\ s \vee Q\ s) \wedge^* R)\ s = ((P \wedge^* R)\ s \vee (Q \wedge^* R)\ s)$ (**is** *?x* = (*?y* ∨ *?z*))
⟨*proof*⟩

lemma *sep-conj-conj*:
$((\lambda s.\ P\ s \wedge Q\ s) \wedge^* R)\ s \Longrightarrow (P \wedge^* R)\ s \wedge (Q \wedge^* R)\ s$
⟨*proof*⟩

lemma *sep-conj-exists1*:
$((\lambda s.\ \exists x.\ P\ x\ s) \wedge^* Q) = (\lambda s.\ (\exists x.\ (P\ x \wedge^* Q)\ s))$
⟨*proof*⟩

lemma *sep-conj-exists2*:
$(P \wedge^* (\lambda s.\ \exists x.\ Q\ x\ s)) = (\lambda s.\ (\exists x.\ (P \wedge^* Q\ x)\ s))$
⟨*proof*⟩

lemmas *sep-conj-exists* = *sep-conj-exists1 sep-conj-exists2*

lemma *sep-conj-forall*:

$((\lambda s.\ \forall x.\ P\ x\ s) \wedge^*\ Q)\ s \Longrightarrow (P\ x \wedge^*\ Q)\ s$
⟨*proof*⟩

lemma *sep-conj-impl*:
$[\![\ (P \wedge^*\ Q)\ s;\ \bigwedge s.\ P\ s \Longrightarrow P'\ s;\ \bigwedge s.\ Q\ s \Longrightarrow Q'\ s\ ]\!] \Longrightarrow (P' \wedge^*\ Q')\ s$
⟨*proof*⟩

lemma *sep-conj-sep-true-left*:
$(P \wedge^*\ Q)\ s \Longrightarrow (\mathit{sep\text{-}true} \wedge^*\ Q)\ s$
⟨*proof*⟩

lemma *sep-conj-sep-true-right*:
$(P \wedge^*\ Q)\ s \Longrightarrow (P \wedge^*\ \mathit{sep\text{-}true})\ s$
⟨*proof*⟩

lemma *sep-globalise*:
$[\![\ (P \wedge^*\ R)\ s;\ (\bigwedge s.\ P\ s \Longrightarrow Q\ s)\ ]\!] \Longrightarrow (Q \wedge^*\ R)\ s$
⟨*proof*⟩

lemma *sep-implI*:
$\forall s'.\ s \perp s' \wedge x\ s' \longrightarrow y\ (s +\!\!+ s') \Longrightarrow (x \longrightarrow^*\ y)\ s$
⟨*proof*⟩

lemma *sep-implI'*:
assumes x: $\bigwedge h'.\ [\![\ h \perp h';\ x\ h'\ ]\!] \Longrightarrow y\ (h +\!\!+ h')$
shows $(x \longrightarrow^*\ y)\ h$
⟨*proof*⟩

lemma *sep-implD*:
$(x \longrightarrow^*\ y)\ s \Longrightarrow \forall s'.\ s \perp s' \wedge x\ s' \longrightarrow y\ (s +\!\!+ s')$
⟨*proof*⟩

lemma *sep-emp-sep-impl* [*simp*]:
$(\square \longrightarrow^*\ P) = P$
⟨*proof*⟩

lemma *sep-impl-sep-true* [*simp*]:
$(P \longrightarrow^*\ \mathit{sep\text{-}true}) = \mathit{sep\text{-}true}$
⟨*proof*⟩

lemma *sep-impl-sep-false* [*simp*]:
$(\mathit{sep\text{-}false} \longrightarrow^*\ P) = \mathit{sep\text{-}true}$
⟨*proof*⟩

lemma *sep-impl-sep-true-P*:
$(\mathit{sep\text{-}true} \longrightarrow^*\ P)\ s \Longrightarrow P\ s$
⟨*proof*⟩

lemma *sep-impl-sep-true-false* [*simp*]:

$(sep\text{-}true \longrightarrow^* sep\text{-}false) = sep\text{-}false$
⟨*proof*⟩

lemma *sep-impl-impl*:
$(P \longrightarrow^* Q \longrightarrow^* R) = (P \wedge^* Q \longrightarrow^* R)$
⟨*proof*⟩

lemma *sep-conj-sep-impl*:
$[\![P\ s; \bigwedge s.\ (P \wedge^* Q)\ s \Longrightarrow R\ s]\!] \Longrightarrow (Q \longrightarrow^* R)\ s$
⟨*proof*⟩

lemma *sep-conj-sep-impl2*:
$[\![(P \wedge^* Q)\ s; \bigwedge s.\ P\ s \Longrightarrow (Q \longrightarrow^* R)\ s]\!] \Longrightarrow R\ s$
⟨*proof*⟩

lemma *sep-map'-anyI-exc* [*simp*]:
$(p \hookrightarrow_g v)\ s \Longrightarrow (p \hookrightarrow_g -)\ s$
⟨*proof*⟩

lemma *sep-map'-anyD-exc*:
$(p \hookrightarrow_g -)\ s \Longrightarrow \exists v.\ (p \hookrightarrow_g v)\ s$
⟨*proof*⟩

lemma *sep-map'-any-unfold-exc*:
$(i \hookrightarrow_g -) = ((i \hookrightarrow_g -) \wedge^* sep\text{-}true)$
⟨*proof*⟩

lemma *sep-map'-inj-exc*:
assumes *pv*: $(p \hookrightarrow_g (v::'a::c\text{-}type))\ s$ **and** *pv'*: $(p \hookrightarrow_h v')\ s$
shows $v = v'$
⟨*proof*⟩

lemma *sep-map'-any-dom-exc*:
$((p::'a::mem\text{-}type\ ptr) \hookrightarrow_g -)\ s \Longrightarrow (ptr\text{-}val\ p, SIndexVal) \in dom\ s$
⟨*proof*⟩

lemma *sep-map'-dom-exc*:
$(p \hookrightarrow_g (v::'a::mem\text{-}type))\ s \Longrightarrow (ptr\text{-}val\ p, SIndexVal) \in dom\ s$
⟨*proof*⟩

lemma *sep-map'-lift-typ-heapD*:
$(p \hookrightarrow_g v)\ s \Longrightarrow$
$\quad lift\text{-}typ\text{-}heap\ g\ s\ p = Some\ (v::'a::c\text{-}type)$
⟨*proof*⟩

lemma *sep-map'-merge*:
assumes *map'-v*: $(p \hookrightarrow_g v)\ s_0 \vee (p \hookrightarrow_g v)\ s_1$ **and** *disj*: $s_0 \perp s_1$
shows $(p \hookrightarrow_g v)\ (s_0 +\!\!+ s_1)$ (**is** *?x*)
⟨*proof*⟩

lemma *sep-conj-overlapD*:
⟦ (*P* ∧* *Q*) *s*; ⋀*s*. *P s* ⟹ ((*p*::'*a*::*mem-type ptr*) ↪*g* −) *s*;
⋀*s*. *Q s* ⟹ (*p* ↪*h* −) *s* ⟧ ⟹ *False*
⟨*proof*⟩

lemma *sep-no-skew*:
(λ*s*. (*p* ↪*g* *v*) *s* ∧ (*q* ↪*h* *w*) *s*) *s* ⟹
p=*q* ∨ {*ptr-val* (*p*::'*a*::*c-type ptr*)..+*size-of TYPE*('*a*)} ∩
{*ptr-val q*..+*size-of TYPE*('*a*)} = {}
⟨*proof*⟩

lemma *sep-no-skew2*:
⟦ (λ*s*. (*p* ↪*g* *v*) *s* ∧ (*q* ↪*h* *w*) *s*) *s*; *typ-uinfo-t TYPE*('*a*) ⊥*t* *typ-uinfo-t TYPE*('*b*)
⟧
⟹ {*ptr-val* (*p*::'*a*::*c-type ptr*)..+*size-of TYPE*('*a*)} ∩
{*ptr-val* (*q*::'*b*::*c-type ptr*)..+*size-of TYPE*('*b*)} = {}
⟨*proof*⟩

lemma *sep-conj-impl-same*:
(*P* ∧* (*P* −→* *Q*)) *s* ⟹ *Q s*
⟨*proof*⟩

definition *pure* :: ('*a*,'*b*) *map-assert* ⇒ *bool* **where**
pure P ≡ ∀ *s s*'. *P s* = *P s*'

lemma *pure-sep-true*:
pure sep-true
⟨*proof*⟩

lemma *pure-sep-fasle*:
pure sep-true
⟨*proof*⟩

lemma *pure-split*:
pure P = (*P* = *sep-true* ∨ *P* = *sep-false*)
⟨*proof*⟩

lemma *pure-sep-conj*:
⟦ *pure P*; *pure Q* ⟧ ⟹ *pure* (*P* ∧* *Q*)
⟨*proof*⟩

lemma *pure-sep-impl*:
⟦ *pure P*; *pure Q* ⟧ ⟹ *pure* (*P* −→* *Q*)
⟨*proof*⟩

lemma *pure-conj-sep-conj*:
⟦ (λs. P s ∧ Q s) s; pure P ∨ pure Q ⟧ ⟹ (P ∧* Q) s
⟨*proof*⟩

lemma *pure-sep-conj-conj*:
⟦ (P ∧* Q) s; pure P; pure Q ⟧ ⟹ (λs. P s ∧ Q s) s
⟨*proof*⟩

lemma *pure-conj-sep-conj-assoc*:
pure P ⟹ ((λs. P s ∧ Q s) ∧* R) = (λs. P s ∧ (Q ∧* R) s)
⟨*proof*⟩

lemma *pure-sep-impl-impl*:
⟦ (P −→* Q) s; pure P ⟧ ⟹ P s −→ Q s
⟨*proof*⟩

lemma *pure-impl-sep-impl*:
⟦ P s −→ Q s; pure P; pure Q ⟧ ⟹ (P −→* Q) s
⟨*proof*⟩

definition
intuitionistic :: ('a,'b) *map-assert* ⇒ *bool*
where
intuitionistic P ≡ ∀ s s'. P s ∧ s ⊆$_m$ s' −→ P s'

lemma *intuitionisticI*:
(⋀s s'. ⟦ P s; s ⊆$_m$ s' ⟧ ⟹ P s') ⟹ *intuitionistic P*
⟨*proof*⟩

lemma *intuitionisticD*:
⟦ *intuitionistic P*; P s; s ⊆$_m$ s' ⟧ ⟹ P s'
⟨*proof*⟩

lemma *pure-intuitionistic*:
pure P ⟹ *intuitionistic P*
⟨*proof*⟩

lemma *intuitionistic-sep-map'*:
intuitionistic (p ↪$_g$ v)
⟨*proof*⟩

lemma *intuitionistic-sep-conj-sep-true*:
intuitionistic (P ∧* *sep-true*)
⟨*proof*⟩

lemma *intuitionistic-sep-impl-sep-true*:
intuitionistic (*sep-true* −→* P)

⟨*proof*⟩

lemma *intuitionistic-conj*:
⟦ *intuitionistic P*; *intuitionistic Q* ⟧ ⟹ *intuitionistic* (λ*s*. *P s* ∧ *Q s*)
⟨*proof*⟩

lemma *intuitionistic-disj*:
⟦ *intuitionistic P*; *intuitionistic Q* ⟧ ⟹ *intuitionistic* (λ*s*. *P s* ∨ *Q s*)
⟨*proof*⟩

lemma *intuitionistic-forall*:
(⋀*x*. *intuitionistic* (*P x*)) ⟹ *intuitionistic* (λ*s*. ∀ *x*. *P x s*)
⟨*proof*⟩

lemma *intuitionistic-exists*:
(⋀*x*. *intuitionistic* (*P x*)) ⟹ *intuitionistic* (λ*s*. ∃ *x*. *P x s*)
⟨*proof*⟩

lemma *intuitionistic-sep-conj*:
intuitionistic (*P*::('*a*,'*b*) *map-assert*) ⟹ *intuitionistic* (*P* ∧* *Q*)
⟨*proof*⟩

lemma *intuitionistic-sep-impl*:
intuitionistic (*Q*::('*a*,'*b*) *map-assert*) ⟹ *intuitionistic* (*P* −→* *Q*)
⟨*proof*⟩

lemma *strongest-intuitionistic*:
¬ (∃ *Q*. (∀ *s*. (*Q s* ⟶ (*P* ∧* *sep-true*) *s*)) ∧ *intuitionistic Q* ∧
Q ≠ (*P* ∧* *sep-true*) ∧ (∀ *s*. *P s* ⟶ *Q s*))
⟨*proof*⟩

lemma *weakest-intuitionistic*:
¬ (∃ *Q*. (∀ *s*. ((*sep-true* −→* *P*) *s* ⟶ *Q s*)) ∧ *intuitionistic Q* ∧
Q ≠ (*sep-true* −→* *P*) ∧ (∀ *s*. *Q s* ⟶ *P s*))
⟨*proof*⟩

lemma *intuitionistic-sep-conj-sep-true-P*:
⟦ (*P* ∧* *sep-true*) *s*; *intuitionistic P* ⟧ ⟹ *P s*
⟨*proof*⟩

lemma *intuitionistic-sep-conj-sep-true-simp*:
intuitionistic P ⟹ (*P* ∧* *sep-true*) = *P*
⟨*proof*⟩

lemma *intuitionistic-sep-impl-sep-true-P*:
⟦ *P s*; *intuitionistic P* ⟧ ⟹ (*sep-true* −→* *P*) *s*
⟨*proof*⟩

lemma *intuitionistic-sep-impl-sep-true-simp*:

intuitionistic $P \Longrightarrow (\mathit{sep\text{-}true} \longrightarrow^* P) = P$
⟨*proof*⟩

definition
dom-exact :: ('*a*,'*b*) *map-assert* ⇒ *bool*
where
$\mathit{dom\text{-}exact}\ P \equiv \forall\, s\ s'.\ P\ s \wedge P\ s' \longrightarrow \mathit{dom}\ s = \mathit{dom}\ s'$

lemma *dom-exactI*:
$(\bigwedge s\ s'.\ [\![\ P\ s;\ P\ s'\ ]\!] \Longrightarrow \mathit{dom}\ s = \mathit{dom}\ s') \Longrightarrow \mathit{dom\text{-}exact}\ P$
⟨*proof*⟩

lemma *dom-exactD*:
$[\![\ \mathit{dom\text{-}exact}\ P;\ P\ s_0;\ P\ s_1\ ]\!] \Longrightarrow \mathit{dom}\ s_0 = \mathit{dom}\ s_1$
⟨*proof*⟩

lemma *dom-exact-sep-conj*:
$[\![\ \mathit{dom\text{-}exact}\ P;\ \mathit{dom\text{-}exact}\ Q\ ]\!] \Longrightarrow \mathit{dom\text{-}exact}\ (P \wedge^*\ Q)$
⟨*proof*⟩

lemma *dom-exact-sep-conj-conj*:
$[\![\ (P \wedge^*\ R)\ s;\ (Q \wedge^*\ R)\ s;\ \mathit{dom\text{-}exact}\ R\ ]\!] \Longrightarrow ((\lambda s.\ P\ s \wedge Q\ s) \wedge^*\ R)\ s$
⟨*proof*⟩

lemma *sep-conj-conj-simp*:
$\mathit{dom\text{-}exact}\ R \Longrightarrow ((\lambda s.\ P\ s \wedge Q\ s) \wedge^*\ R) = (\lambda s.\ (P \wedge^*\ R)\ s \wedge (Q \wedge^*\ R)\ s)$
⟨*proof*⟩

definition *dom-eps* :: ('*a*,'*b*) *map-assert* ⇒ '*a set* **where**
$\mathit{dom\text{-}eps}\ P \equiv \mathit{THE}\ x.\ \forall\, s.\ P\ s \longrightarrow x = \mathit{dom}\ s$

lemma *dom-epsI*:
$[\![\ \mathit{dom\text{-}exact}\ P;\ P\ s;\ x \in \mathit{dom}\ s\ ]\!] \Longrightarrow x \in \mathit{dom\text{-}eps}\ P$
⟨*proof*⟩

lemma *dom-epsD* [*rule-format*]:
$[\![\ \mathit{dom\text{-}exact}\ P;\ P\ s\ ]\!] \Longrightarrow x \in \mathit{dom\text{-}eps}\ P \longrightarrow x \in \mathit{dom}\ s$
⟨*proof*⟩

lemma *dom-eps*:
$[\![\ \mathit{dom\text{-}exact}\ P;\ P\ s\ ]\!] \Longrightarrow \mathit{dom}\ s = \mathit{dom\text{-}eps}\ P$
⟨*proof*⟩

lemma *map-restrict-dom-exact*:
$[\![\ \mathit{dom\text{-}exact}\ P;\ P\ s\ ]\!] \Longrightarrow s\ |`\ \mathit{dom\text{-}eps}\ P = s$
⟨*proof*⟩

lemma *map-restrict-dom-exact2*:
⟦ *dom-exact P*; *P* s_0; $s_0 \perp s_1$ ⟧ ⟹ (s_1 |‘ *dom-eps P*) = *Map.empty*
⟨*proof*⟩

lemma *map-restrict-dom-exact3*:
⟦ *dom-exact P*; *P s* ⟧ ⟹ *s* |‘ (*UNIV* − *dom-eps P*) = *Map.empty*
⟨*proof*⟩

lemma *map-add-restrict-dom-exact*:
⟦ *dom-exact P*; $s_0 \perp s_1$; *P* s_1 ⟧ ⟹ (s_1 ++ s_0) |‘ (*dom-eps P*) = s_1
⟨*proof*⟩

lemma *map-add-restrict-dom-exact2*:
⟦ *dom-exact P*; $s_0 \perp s_1$; *P* s_0 ⟧ ⟹ (s_1 ++ s_0) |‘ (*UNIV* − *dom-eps P*) = s_1
⟨*proof*⟩

lemma *dom-exact-sep-conj-forall*:
assumes *sc*: ∀ *x*. (*P x* ∧* *Q*) *s* **and** *de*: *dom-exact Q*
shows ((λ*s*. ∀ *x*. *P x s*) ∧* *Q*) *s*
⟨*proof*⟩

lemma *sep-conj-forall-simp*:
dom-exact Q ⟹ ((λ*s*. ∀ *x*. *P x s*) ∧* *Q*) = (λ*s*. ∀ *x*. (*P x* ∧* *Q*) *s*)
⟨*proof*⟩

lemma *dom-exact-sep-map*:
dom-exact (*i* ↦$_g$ *x*)
⟨*proof*⟩

lemma *dom-exact-P-emp*:
⟦ *dom-exact P*; *P Map.empty*; *P s* ⟧ ⟹ *s* = *Map.empty*
⟨*proof*⟩

definition
strictly-exact :: ('*a*,'*b*) *map-assert* ⇒ *bool*
where
strictly-exact P ≡ ∀ *s s'*. *P s* ∧ *P s'* ⟶ *s* = *s'*

lemma *strictly-exactD*:
⟦ *strictly-exact P*; *P* s_0; *P* s_1 ⟧ ⟹ s_0 = s_1
⟨*proof*⟩

lemma *strictly-exactI*:
(⋀*s s'*. ⟦ *P s*; *P s'* ⟧ ⟹ *s* = *s'*) ⟹ *strictly-exact P*
⟨*proof*⟩

lemma *strictly-exact-dom-exact*:
strictly-exact P $\Longrightarrow$ *dom-exact P*
⟨*proof*⟩

lemma *strictly-exact-sep-emp*:
strictly-exact □
⟨*proof*⟩

lemma *strictly-exact-sep-conj*:
⟦ *strictly-exact P*; *strictly-exact Q* ⟧ $\Longrightarrow$ *strictly-exact* (*P* $\wedge^*$ *Q*)
⟨*proof*⟩

lemma *strictly-exact-conj-impl*:
⟦ (*Q* $\wedge^*$ *sep-true*) *s*; *P s*; *strictly-exact Q* ⟧ $\Longrightarrow$ (*Q* $\wedge^*$ (*Q* $\longrightarrow^*$ *P*)) *s*
⟨*proof*⟩

lemma *dom-eps-sep-emp* [*simp*]:
dom-eps □ = {}
⟨*proof*⟩

lemma *dom-eps-sep-map*:
g p $\Longrightarrow$ *dom-eps* (*p* $\mapsto_g$ (*v*::'*a*::*mem-type*)) = *s-footprint p*
⟨*proof*⟩

definition *non-empty* :: ('*a*,'*b*) *map-assert* ⇒ *bool* **where**
non-empty P ≡ ∃ *s*. *P s*

lemma *non-emptyI*:
P s $\Longrightarrow$ *non-empty P*
⟨*proof*⟩

lemma *non-emptyD*:
non-empty P $\Longrightarrow$ ∃ *s*. *P s*
⟨*proof*⟩

lemma *non-empty-sep-true*:
non-empty sep-true
⟨*proof*⟩

lemma *non-empty-sep-false*:
¬ *non-empty sep-false*
⟨*proof*⟩

lemma *non-empty-sep-emp*:
non-empty □
⟨*proof*⟩

lemma *non-empty-sep-map*:
$g\ p \Longrightarrow$ *non-empty* $(p \mapsto_g (v::'a::mem\text{-}type))$
⟨*proof*⟩

lemma *non-empty-sep-conj*:
$[\![$ *non-empty* P; *non-empty* Q; *dom-exact* P; *dom-exact* Q;
dom-eps $P \cap$ *dom-eps* $Q = \{\}\ ]\!] \Longrightarrow$ *non-empty* $(P \wedge^* Q)$
⟨*proof*⟩

lemma *non-empty-sep-map'*:
$g\ p \Longrightarrow$ *non-empty* $(p \hookrightarrow_g (v::'a::mem\text{-}type))$
⟨*proof*⟩

lemma *non-empty-sep-impl*:
$\neg$ *P Map.empty* $\Longrightarrow$ *non-empty* $(P \longrightarrow^* Q)$
⟨*proof*⟩

lemma *pure-conj-right*: $(Q \wedge^* (\lambda s.\ P' \wedge Q'\ s)) = (\lambda s.\ P' \wedge (Q \wedge^* Q')\ s)$
⟨*proof*⟩

lemma *pure-conj-right'*: $(Q \wedge^* (\lambda s.\ P'\ s \wedge Q')) = (\lambda s.\ Q' \wedge (Q \wedge^* P')\ s)$
⟨*proof*⟩

lemma *pure-conj-left*: $((\lambda s.\ P' \wedge Q'\ s) \wedge^* Q) = (\lambda s.\ P' \wedge (Q' \wedge^* Q)\ s)$
⟨*proof*⟩

lemma *pure-conj-left'*: $((\lambda s.\ P'\ s \wedge Q') \wedge^* Q) = (\lambda s.\ Q' \wedge (P' \wedge^* Q)\ s)$
⟨*proof*⟩

lemmas *pure-conj = pure-conj-right pure-conj-right' pure-conj-left pure-conj-left'*

declare *pure-conj* [*simp add*]

lemma *sep-conj-sep-conj-sep-impl-sep-conj*:
$(P \wedge^* R)\ s \Longrightarrow (P \wedge^* (Q \longrightarrow^* (Q \wedge^* R)))\ s$
⟨*proof*⟩

lemma *sep-map'-conjE1-exc*:
$[\![\ (P \wedge^* Q)\ s;\ \bigwedge s.\ P\ s \Longrightarrow (i \hookrightarrow_g v)\ s\ ]\!] \Longrightarrow (i \hookrightarrow_g v)\ s$
⟨*proof*⟩

lemma *sep-map'-conjE2-exc*:
$[\![\ (P \wedge^* Q)\ s;\ \bigwedge s.\ Q\ s \Longrightarrow (i \hookrightarrow_g v)\ s\ ]\!] \Longrightarrow (i \hookrightarrow_g v)\ s$
⟨*proof*⟩

lemma *sep-map'-any-conjE1-exc*:
$[\![\ (P \wedge^* Q)\ s;\ \bigwedge s.\ P\ s \Longrightarrow (i \hookrightarrow_g -)\ s\ ]\!] \Longrightarrow (i \hookrightarrow_g -)\ s$

⟨*proof*⟩

lemma *sep-map'-any-conjE2-exc*:
⟦ (*P* ∧* *Q*) *s*; ⋀*s*. *Q s* ⟹ (*i* ↪*g* −) *s* ⟧ ⟹ (*i* ↪*g* −) *s*
⟨*proof*⟩

lemma *sep-conj-mapD-exc*:
((*i* ↦*g* *v*) ∧* *P*) *s* ⟹ (*i* ↪*g* *v*) *s* ∧ ((*i* ↦*g* −) ∧* *P*) *s*
⟨*proof*⟩

lemma *sep-impl-conj-sameD*:
⟦ (*P* −→* *P* ∧* *Q*) *s*; *dom-exact P*; *non-empty P*; *dom s* ⊆ *UNIV* − *dom-eps P*
⟧
⟹ *Q s*
⟨*proof*⟩

lemma *sep-impl-conj-sameI*:
Q s ⟹ (*P* −→* *P* ∧* *Q*) *s*
⟨*proof*⟩

end

theory *SepCode*
imports
Separation-UMM
Simpl.Vcg
begin

definition
singleton-t :: *'a*::*c-type ptr* ⇒ *'a* ⇒ *heap-state*
where
singleton-t p v ≡ *lift-state* (*heap-update p v* (λ*x*. *0*), (*ptr-retyp p empty-htd*))

definition
tagd :: *'a ptr-guard* ⇒ *'a*::*c-type ptr* ⇒ *heap-assert* (**infix** ‹⊢*s*› *100*)
where
g ⊢*s* *p* ≡ λ*s*. *s*,*g* ⊨*s* *p* ∧ *dom s* = *s-footprint p*

definition
field-footprint :: *'a*::*c-type ptr* ⇒ *qualified-field-name* ⇒ (*addr* × *s-heap-index*)
set
where
field-footprint p f ≡
s-footprint-untyped (*ptr-val p* + *of-nat* (*field-offset TYPE*(*'a*) *f*))
(*export-uinfo* (*field-typ TYPE*(*'a*) *f*))

definition
fs-footprint :: *'a::c-type ptr* ⇒ *qualified-field-name set* ⇒ (*addr* × *s-heap-index*) *set*
where
fs-footprint p F ≡ ⋃{*field-footprint p f* | *f*. *f* ∈ *F*}

definition *fields* :: *'a::c-type itself* ⇒ *qualified-field-name set* **where**
fields t ≡ {*f*. *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* ≠ *None*}

definition
mfs-sep-map :: *'a::c-type ptr* ⇒ *'a ptr-guard* ⇒ *qualified-field-name set* ⇒ *'a* ⇒ *heap-assert*
(‹(‹*open-block notation*=‹*mixfix mfs-sep-map*››- ↦_ -)› [*56*,*0*,*0*,*51*] *56*)
where
p ↦$_g{}^F$ *v* ≡ λ*s*. *lift-typ-heap g* (*singleton-t p v* ++ *s*) *p* = *Some v* ∧
F ⊆ *fields TYPE*('*a*) ∧
dom s = *s-footprint p* − *fs-footprint p F* ∧ *wf-heap-val s*

notation (*input*)
mfs-sep-map (‹(‹*open-block notation*=‹*mixfix mfs-sep-map*››- ↦_ -)› [*56*,*0*,*1000*,*51*] *56*)

definition
disjoint-fn :: *qualified-field-name* ⇒ *qualified-field-name set* ⇒ *bool*
where
disjoint-fn f F ≡ ∀*f'*∈*F*. ¬ *f* ≤ *f'* ∧ ¬ *f'* ≤ *f*

definition
sep-cut' :: *addr* ⇒ *nat* ⇒ (*s-addr*,*'b*) *map-assert*
where
sep-cut' p n ≡ λ*s*. *dom s* = {(*x*,*y*). *x* ∈ {*p*..+*n*}}

definition
sep-cut :: *addr* ⇒ *addr-bitsize word* ⇒ (*s-addr*,*'b*) *map-assert*
where
sep-cut x y ≡ *sep-cut' x* (*unat y*)

—-

lemma *heap-list-h-eq*:
⟦ *x* ∈ {*p*..+*q*}; *q* < *addr-card*; *heap-list h q p* = *heap-list h' q p* ⟧ ⟹ *h x* = *h' x*
⟨*proof*⟩

lemma *s-footprint-intvl*:
(*a*, *SIndexVal*) ∈ *s-footprint p* = (*a* ∈ {*ptr-val* (*p*::*'a*::*c-type ptr*)..+*size-of TYPE*('*a*)})
⟨*proof*⟩

lemma *singleton-t-dom* [*simp*]:
dom (*singleton-t p* (*v*::*'a*::*mem-type*)) = *s-footprint p*
⟨*proof*⟩

lemma *heap-update-merge*:
assumes *val*: $d,g \models_t p$
shows *lift-state* ((*heap-update* p (v::'a::*mem-type*) h),d)
= *lift-state* (h,d) ++ *singleton* p v h d (**is** *?x* = *?y*)
⟨*proof*⟩

lemma *tagd-dom-exc*:
($g \vdash_s p$) $s \Longrightarrow$ *dom* s = *s-footprint* p
⟨*proof*⟩

lemma *tagd-dom-p-exc*:
($g \vdash_s p$) $s \Longrightarrow$ (*ptr-val* (p::'a::*mem-type ptr*),*SIndexVal*) ∈ *dom* s
⟨*proof*⟩

lemma *tagd-g-exc*:
($g \vdash_s p \wedge^* P$) $s \Longrightarrow g\ p$
⟨*proof*⟩

lemma *sep-map-tagd-exc*:
($p \mapsto_g$ (v::'a::*mem-type*)) $s \Longrightarrow$ ($g \vdash_s p$) s
⟨*proof*⟩

lemma *sep-map-any-tagd-exc*:
($p \mapsto_g -$) $s \Longrightarrow$ ($g \vdash_s$ (p::'a::*mem-type ptr*)) s
⟨*proof*⟩

lemma *ptr-retyp-tagd-exc*:
g (p::'a::*mem-type ptr*) ⟹
($g \vdash_s p$) (*lift-state* (h, *ptr-retyp* p *empty-htd*))
⟨*proof*⟩

lemma *singleton-dom-proj-d* [*simp*]:
($g \vdash_s p$) $s \Longrightarrow$ *dom* (*singleton* p (v::'a::*mem-type*) h (*proj-d* s)) = *dom* s
⟨*proof*⟩

lemma *singleton-d-restrict-eq*:
restrict-s d (*s-footprint* p) = *restrict-s* d' (*s-footprint* p)
⟹ *singleton* p v h d = *singleton* p (v::'a::*mem-type*) h d'
⟨*proof*⟩

lemma *sep-heap-update'-exc*:
assumes *sep*: ($g \vdash_s p \wedge^* (p \mapsto_g v \longrightarrow^* P)$) (*lift-state* ($h$,$d$))
shows P (*lift-state* (*heap-update* p (v::'a::*mem-type*) h,d))
⟨*proof*⟩

lemma *sep-heap-update-exc*:
⟦ ($p \mapsto_g - \wedge^* (p \mapsto_g v \longrightarrow^* P)$) (*lift-state* ($h$,$d$)) ⟧ ⟹
P (*lift-state* (*heap-update* p (v::'a::*mem-type*) h,d))

⟨*proof*⟩

lemma *sep-heap-update-global'-exc*:
(*g* ⊢$_s$ *p* ∧* *R*) (*lift-state* (*h*,*d*)) ⟹
((*p* ↦$_g$ *v*) ∧* *R*) (*lift-state* (*heap-update p* (*v*::'*a*::*mem-type*) *h*,*d*))
⟨*proof*⟩

lemma *sep-heap-update-global-exc*:
(*p* ↦$_g$ − ∧* *R*) (*lift-state* (*h*,*d*)) ⟹
((*p* ↦$_g$ *v*) ∧* *R*) (*lift-state* (*heap-update p* (*v*::'*a*::*mem-type*) *h*,*d*))
⟨*proof*⟩

lemma *sep-heap-update-global-exc2*:
(*p* ↦$_g$ *u* ∧* *R*) (*lift-state* (*h*,*d*)) ⟹
((*p* ↦$_g$ *v*) ∧* *R*) (*lift-state* (*heap-update p* (*v*::'*a*::*mem-type*) *h*,*d*))
⟨*proof*⟩

lemma *heap-update-mem-same-point*:
⟦ *q* ∈ {*p*..+*length v*}; *length v* < *addr-card* ⟧ ⟹
heap-update-list p v h q = *v* ! *unat* (*q* − *p*)
⟨*proof*⟩

lemma *heap-update-list-value*:
length v < *addr-card* ⟹
heap-update-list p v h q = (*if q* ∈ {*p*..+*length v*} *then v*!*unat* (*q*−*p*) *else h q*)
⟨*proof*⟩

lemma *heap-update-list-value'*:
length xs < *addr-card* ⟹
heap-update-list ptr xs hp x = (*if unat* (*x* − *ptr*) < *length xs then xs* ! *unat* (*x* − *ptr*) *else hp x*)
⟨*proof*⟩

lemma *heap-list-h-eq2*:
(⋀*x*. *x* ∈ {*p*..+*n*} ⟹ *h x* = *h'* *x*) ⟹ *heap-list h n p* = *heap-list h' n p*
⟨*proof*⟩

lemma *map-td-f-eq'*:
(*f*=*g*) ⟶ (*map-td f h t* = *map-td g h t*)
(*f*=*g*) ⟶ (*map-td-struct f h st* = *map-td-struct g h st*)
(*f*=*g*) ⟶ (*map-td-list f h ts* = *map-td-list g h ts*)
(*f*=*g*) ⟶ (*map-td-tuple f h x* = *map-td-tuple g h x*)
⟨*proof*⟩

lemma *map-td-f-eq*:
f=*g* ⟹ *map-td f t* = *map-td g t*
⟨*proof*⟩

lemma *sep-map'-lift-exc*:

$(p \hookrightarrow_g (v::'a::mem\text{-}type))\ (lift\text{-}state\ (h,d)) \Longrightarrow lift\ h\ p = v$
⟨*proof*⟩

lemma *sep-map-lift-wp-exc*:
$\exists v.\ (p \mapsto_g v \wedge^* (p \mapsto_g v \longrightarrow^* P\ v))\ (lift\text{-}state\ (h,d))$
$\Longrightarrow P\ (lift\ h\ (p::'a::mem\text{-}type\ ptr))\ (lift\text{-}state\ (h,d))$
⟨*proof*⟩

lemma *sep-map-lift-exc*:
$((p::'a::mem\text{-}type\ ptr) \mapsto_g -)\ (lift\text{-}state\ (h,d)) \Longrightarrow$
$(p \mapsto_g lift\ h\ p)\ (lift\text{-}state\ (h,d))$
⟨*proof*⟩

lemma *sep-map'-lift-rev-exc*:
$[\![\ lift\ h\ p = (v::'a::mem\text{-}type);\ (p \hookrightarrow_g -)\ (lift\text{-}state\ (h,d))\ ]\!] \Longrightarrow$
$(p \hookrightarrow_g v)\ (lift\text{-}state\ (h,d))$
⟨*proof*⟩

lemma *sep-lift-exists-exc*:
fixes $p :: 'a::mem\text{-}type\ ptr$
assumes *ex*: $((\lambda s.\ \exists v.\ (p \hookrightarrow_g v)\ s \wedge P\ v\ s) \wedge^* Q)\ (lift\text{-}state\ (h,d))$
shows $(P\ (lift\ h\ p) \wedge^* Q)\ (lift\text{-}state\ (h,d))$
⟨*proof*⟩

lemma *merge-dom*:
$x \in dom\ s \Longrightarrow (t +\!\!+ s)\ x = s\ x$
⟨*proof*⟩

lemma *merge-dom2*:
$x \notin dom\ s \Longrightarrow (t +\!\!+ s)\ x = t\ x$
⟨*proof*⟩

lemma *fs-footprint-empty* [*simp*]:
$fs\text{-}footprint\ p\ \{\} = \{\}$
⟨*proof*⟩

lemma *fs-footprint-un*:
$fs\text{-}footprint\ p\ (insert\ f\ F) = fs\text{-}footprint\ p\ \{f\} \cup fs\text{-}footprint\ p\ F$
⟨*proof*⟩

lemma *proj-d-restrict-map-le*:
$snd\ (proj\text{-}d\ (s\ |`\ X)\ x) \subseteq_m snd\ (proj\text{-}d\ s\ x)$
⟨*proof*⟩

lemma *SIndexVal-conj-setcomp-simp* [*simp*]:
$\{x.\ snd\ x = SIndexVal \wedge x \notin s\text{-}footprint\text{-}untyped\ p\ t\}$
$= \{(x,SIndexVal) \mid x.\ x \notin \{p..+size\text{-}td\ t\}\}$
⟨*proof*⟩

lemma *heap-list-s-restrict-same*:
$\{(x,SIndexVal) \mid x.\ x \in \{p..+n\}\} \subseteq X \Longrightarrow heap\text{-}list\text{-}s\ (s\ |`\ X)\ n\ p = heap\text{-}list\text{-}s\ s\ n\ p$
⟨*proof*⟩

lemma *heap-list-s-restrict-fs-footprint*:
$field\text{-}lookup\ (typ\text{-}info\text{-}t\ TYPE('a::c\text{-}type))\ f\ 0 = Some\ (t,n) \Longrightarrow$
$heap\text{-}list\text{-}s\ (s\ |`\ fs\text{-}footprint\ p\ \{f\})\ (size\text{-}td\ t)\ \&(p\rightarrow f) = heap\text{-}list\text{-}s\ s\ (size\text{-}td\ t)\ \&((p::'a\ ptr)\rightarrow f)$
⟨*proof*⟩

lemma *heap-list-proj-h-disj*:
$\{(x,SIndexVal) \mid x.\ x \in \{p..+n\}\} \cap dom\ s_1 = \{\} \Longrightarrow$
$heap\text{-}list\ (proj\text{-}h\ (s_0\ +\!\!+\ s_1))\ n\ p = heap\text{-}list\ (proj\text{-}h\ s_0)\ n\ p$
⟨*proof*⟩

lemma *heap-list-proj-h-sub*:
$\{(x,SIndexVal) \mid x.\ x \in \{p..+n\}\} \subseteq dom\ s_1 \Longrightarrow$
$heap\text{-}list\ (proj\text{-}h\ (s_0\ +\!\!+\ s_1))\ n\ p = heap\text{-}list\ (proj\text{-}h\ s_1)\ n\ p$
⟨*proof*⟩

lemma *heap-list-s-map-add-super-update-bs*:
$[\![\ \{x.\ (x,SIndexVal) \in dom\ s_1\} = \{p+of\text{-}nat\ k..+z\};\ k + z \leq n;\ n < addr\text{-}card\ ]\!]$
$\Longrightarrow heap\text{-}list\text{-}s\ (s_0\ +\!\!+\ s_1)\ n\ p =$
$super\text{-}update\text{-}bs\ (heap\text{-}list\text{-}s\ s_1\ z\ (p+of\text{-}nat\ k))\ (heap\text{-}list\text{-}s\ s_0\ n\ p)\ k$
⟨*proof*⟩

lemma *s-footprint-untyped-dom-SIndexVal*:
$dom\ s = s\text{-}footprint\text{-}untyped\ p\ t \Longrightarrow$
$\{x.\ (x,SIndexVal) \in dom\ s\} = \{p..+size\text{-}td\ t\}$
⟨*proof*⟩

lemma *field-ti-s-sub*:
$field\text{-}lookup\ (export\text{-}uinfo\ (typ\text{-}info\text{-}t\ TYPE('b::mem\text{-}type)))\ f\ 0 = Some\ (a,b) \Longrightarrow$
$s\text{-}footprint\text{-}untyped\ \&(p\rightarrow f)\ a \subseteq s\text{-}footprint\ (p::'b\ ptr)$
⟨*proof*⟩

lemma *wf-heap-val-map-add* [*simp*]:
$[\![\ wf\text{-}heap\text{-}val\ s_0;\ wf\text{-}heap\text{-}val\ s_1\ ]\!] \Longrightarrow wf\text{-}heap\text{-}val\ (s_0\ +\!\!+\ s_1)$
⟨*proof*⟩

lemma *of-nat-lt-size-of*:
$[\![\ (of\text{-}nat\ x::addr) = of\text{-}nat\ y + of\text{-}nat\ z;\ x < size\text{-}of\ TYPE('a::mem\text{-}type);$
$y + z < size\text{-}of\ TYPE('a)\ ]\!] \Longrightarrow x = y+z$
⟨*proof*⟩

lemma *proj-d-map-add*:
snd (*proj-d* s_1 *p*) *n* = *Some k* ⟹ *snd* (*proj-d* (s_0 ++ s_1) *p*) *n* = *Some k*
⟨*proof*⟩

lemma *proj-d-map-add2*:
fst (*proj-d* s_1 *p*) ⟹ *fst* (*proj-d* (s_0 ++ s_1) *p*)
⟨*proof*⟩

lemma *heap-list-s-restrict-disj-same*:
dom s ∩ (*UNIV* − *X*) = {} ⟹ *heap-list-s* (*s* |‘ *X*) *n p* = *heap-list-s s n p*
⟨*proof*⟩

lemma *UNIV-minus-inter*:
(*X* − *Y*) ∩ (*X* ∩ (*X* − *Y*) − *Z*) = *X* − (*Y* ∪ *Z*)
⟨*proof*⟩

lemma *sep-map-mfs-sep-map-empty*:
(*p* ↦$_g$ (*v*::′*a*::*mem-type*)) = (*p* ↦$_g$‘{}) *v*)
⟨*proof*⟩

lemma *fd-cons-double-update*:
⟦ *fd-cons t*; *length bs* = *length bs*′ ⟧ ⟹
update-ti-t t bs (*update-ti-t t bs*′ *v*) = *update-ti-t t bs v*
⟨*proof*⟩

lemma *fd-cons-update-access*:
⟦ *fd-cons t*; *length bs* = *size-td t* ⟧ ⟹
update-ti-t t (*access-ti t v bs*) *v* = *v*
⟨*proof*⟩

lemma *fd-cons-length*:
⟦ *fd-cons t*; *length bs* = *size-td t* ⟧ ⟹
length (*access-ti t v bs*) = *size-td t*
⟨*proof*⟩

lemma *fd-cons-length-p*:
fd-cons t ⟹ *length* (*access-ti*$_0$ *t v*) = *size-td t*
⟨*proof*⟩

lemma *fd-cons-update-normalise*:
⟦ *fd-cons t*; *length bs* = *size-td t* ⟧ ⟹
update-ti-t t ((*norm-desc* (*field-desc t*) (*size-td t*)) *bs*) *v* = *update-ti-t t bs v*
⟨*proof*⟩

lemma *field-footprint-SIndexVal*:
field-lookup (*typ-info-t TYPE*(′*a*::*c-type*)) *f* 0 = *Some* (*t*, *n*) ⟹
{*x*. (*x*, *SIndexVal*) ∈ *field-footprint* (*p*::′*a ptr*) *f*} =
{*ptr-val p* + *of-nat n*..+*size-td t*}
⟨*proof*⟩

lemma *fs-footprint-subset*:
$F \subseteq$ *fields* $TYPE('a::mem\text{-}type) \Longrightarrow$ *fs-footprint* $(p::'a\ ptr)\ F \subseteq$ *s-footprint* p
⟨*proof*⟩

lemma *length-heap-list-s* [*simp*]:
length (*heap-list-s* $s\ n\ p$) $= n$
⟨*proof*⟩

lemma *heap-list-proj-h-restrict*:
$\{p..+n\} \subseteq \{x.\ (x,SIndexVal) \in X\} \Longrightarrow$ *heap-list* (*proj-h* (s |` X)) $n\ p =$ *heap-list* (*proj-h* s) $n\ p$
⟨*proof*⟩

lemma *heap-list-proj-h-lift-state*:
$\{p..+n\} \subseteq \{x.\ fst\ (d\ x)\} \Longrightarrow$ *heap-list* (*proj-h* (*lift-state* (h,d))) $n\ p =$ *heap-list* $h\ n\ p$
⟨*proof*⟩

lemma *heap-list-rpbs*:
heap-list $(\lambda x.\ 0)\ n\ p =$ *replicate* $n\ 0$
⟨*proof*⟩

lemma *field-access-take-drop*:
fixes
$t::('a,'b)$ *typ-info* **and**
$st::('a,'b)$ *typ-info-struct* **and**
$ts::('a,'b)$ *typ-info-tuple list* **and**
$x::('a,'b)$ *typ-info-tuple*
shows
$\forall s\ m\ n\ f.$ *field-lookup* $t\ f\ m =$ *Some* $(s,n) \longrightarrow$ *wf-fd* $t \longrightarrow$
take (*size-td* s) (*drop* $(n - m)$ (*access-ti*$_0$ $t\ v$)) =
access-ti$_0$ $s\ v$
$\forall s\ m\ n\ f.$ *field-lookup-struct* $st\ f\ m =$ *Some* $(s,n) \longrightarrow$ *wf-fd-struct* $st \longrightarrow$
take (*size-td* s) (*drop* $(n - m)$ (*access-ti-struct* $st\ v$ (*replicate* (*size-td-struct* st) 0))) =
access-ti$_0$ $s\ v$
$\forall s\ m\ n\ f.$ *field-lookup-list* $ts\ f\ m =$ *Some* $(s,n) \longrightarrow$ *wf-fd-list* $ts \longrightarrow$
take (*size-td* s) (*drop* $(n - m)$ (*access-ti-list* $ts\ v$ (*replicate* (*size-td-list* ts) 0))) =
access-ti$_0$ $s\ v$
$\forall s\ m\ n\ f.$ *field-lookup-tuple* $x\ f\ m =$ *Some* $(s,n) \longrightarrow$ *wf-fd-tuple* $x \longrightarrow$
take (*size-td* s) (*drop* $(n - m)$ (*access-ti-tuple* $x\ v$ (*replicate* (*size-td-tuple* x) 0))) =
access-ti$_0$ $s\ v$
⟨*proof*⟩

lemma *field-access-take-dropD*:
⟦ *field-lookup* $t\ f\ 0 =$ *Some* (s,n); *wf-lf* (*lf-set* t []); *wf-desc* t ⟧ $\Longrightarrow$

take (*size-td s*) (*drop n* (*access-ti*$_0$ *t v*)) = *access-ti*$_0$ *s v*
⟨*proof*⟩

lemma *singleton-t-field*:
field-lookup (*typ-info-t TYPE*('*a*::*mem-type*)) *f 0* = *Some* (*t*, *n*) ⟹
heap-list-s (*singleton-t p v* |‘ *fs-footprint p* {*f*}) (*size-td t*) (*ptr-val p* + *of-nat n*) =
access-ti$_0$ *t v*
⟨*proof*⟩

lemma *field-lookup-fd-consD*:
field-lookup (*typ-info-t TYPE*('*a*::*mem-type*)) *f 0* = *Some* (*t*,*n*) ⟹ *fd-cons t*
⟨*proof*⟩

lemma *s-valid-map-add*:
⟦ *s*,*g* ⊨$_s$ *p*; *t*,*g*′ ⊨$_s$ *p* ⟧ ⟹ (*s* ++ *t* |‘ *X*),*g* ⊨$_s$ *p*
⟨*proof*⟩

lemma *singleton-t-s-valid*:
g p ⟹ *singleton-t p* (*v*::'*a*::*mem-type*),*g* ⊨$_s$ *p*
⟨*proof*⟩

lemma *sep-map-mfs-sep-map*:
⟦ (*p* ↦$_g{}^F$ *v*) *s*; *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*,*n*) ⟧ ⟹
(*p* ↦$_g{}^{(\{f\}\cup F)}$ (*v*::'*a*::*mem-type*)) (*s* |‘ (*dom s* − *fs-footprint p* {*f*}))
⟨*proof*⟩

lemma *disjoint-fn-disjoint*:
⟦ *disjoint-fn f F*; *F* ⊆ *fields TYPE*('*a*::*mem-type*);
field-lookup (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*,*n*) ⟧ ⟹
fs-footprint (*p*::'*a ptr*) *F* ∩ *field-footprint p f* = {}
⟨*proof*⟩

lemma *sep-map-mfs-sep-map2*:
⟦*field-lookup* (*typ-info-t TYPE*('*a*::*mem-type*)) *f 0* = *Some* (*s*, *n*);
disjoint-fn f F; *guard-mono g g*′;
export-uinfo s = *typ-uinfo-t TYPE*('*b*); ((*p*::'*a ptr*) ↦$_g{}^F$ *v*) *x*⟧
⟹ (*Ptr* &(*p*→*f*) ↦$_g$′ ((*from-bytes* (*access-ti*$_0$ *s v*))::'*b*::*mem-type*))
(*x* |‘ *field-footprint p f*)
⟨*proof*⟩

lemma *export-size-of*:
export-uinfo t = *typ-uinfo-t TYPE*('*a*) ⟹ *size-of TYPE*('*a*::*c-type*) = *size-td t*
⟨*proof*⟩

lemma *sep-map-field-unfold*:
⟦ *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*,*n*);
disjoint-fn f F; *guard-mono g g*′;
export-uinfo t = *typ-uinfo-t TYPE*('*b*) ⟧ ⟹

$(p \mapsto_g{}^F\ v) = (p \mapsto_g{}^{(\{f\} \cup F)}\ (v::'a::mem\text{-}type) \wedge^*$
$Ptr\ (\&(p \rightarrow f)) \mapsto_g{}'\ ((from\text{-}bytes\ (access\text{-}ti_0\ t\ v))::'b::mem\text{-}type))$
⟨*proof*⟩

lemma *disjoint-fn-empty* [*simp*]:
disjoint-fn f {}
⟨*proof*⟩

lemma *sep-map-field-map'*:
⟦ $((p::'a::mem\text{-}type\ ptr) \mapsto_g v)\ s$;
field-lookup (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*d*,*n*); *export-uinfo d* = *typ-uinfo-t TYPE*('*b*);
guard-mono g g' ⟧ ⟹
$((Ptr\ (\&(p \rightarrow f))::'b::mem\text{-}type\ ptr) \hookrightarrow_g{}'\ from\text{-}bytes\ (access\text{-}ti_0\ d\ v))\ s$
⟨*proof*⟩

lemma *fd-cons-access-update-p*:
⟦ *fd-cons t*; *length bs* = *size-td t* ⟧ ⟹
$access\text{-}ti_0\ t\ (update\text{-}ti\text{-}t\ t\ bs\ v) = access\text{-}ti_0\ t\ (update\text{-}ti\text{-}t\ t\ bs\ w)$
⟨*proof*⟩

lemma *length-to-bytes-p* [*simp*]:
length (*to-bytes-p* ($v::'a$)) = *size-of TYPE*('*a*::*mem-type*)
⟨*proof*⟩

lemma *inv-p* [*simp*]:
from-bytes (*to-bytes-p v*) = ($v::'a::mem\text{-}type$)
⟨*proof*⟩

lemma *singleton-SIndexVal*:
$x \in \{ptr\text{-}val\ p..+size\text{-}of\ TYPE('a)\} \Longrightarrow$
singleton-t p ($v::'a::mem\text{-}type$) (x,*SIndexVal*) = *Some* (*SValue* (*to-bytes-p v* ! *unat* (x − *ptr-val p*)))
⟨*proof*⟩

lemma *access-ti*$_0$:
access-ti s v (*replicate* (*size-td s*) *0*) = $access\text{-}ti_0\ s\ v$
⟨*proof*⟩

lemma *fd-cons-mem-type* [*simp*]:
fd-cons (*typ-info-t TYPE*('*a*::*mem-type*))
⟨*proof*⟩

lemma *norm-tu-rpbs*:
wf-fd t ⟹ *norm-tu* (*export-uinfo t*) ($access\text{-}ti_0\ t\ v$) = $access\text{-}ti_0\ t\ v$
⟨*proof*⟩

lemma *heap-list-s-singleton-t-field-update*:
⟦ *field-lookup* (*typ-info-t TYPE*('*a*::*mem-type*)) *f 0* = *Some* (*s*, *n*);

export-uinfo s = typ-uinfo-t TYPE('b) ⟧ ⟹
heap-list-s (singleton-t p (update-ti-t s (to-bytes-p w) v)) (size-td s)
(ptr-val (p::'a::mem-type ptr) + of-nat n) =
to-bytes-p (w::'b::mem-type)
⟨*proof*⟩

lemma *field-access-update-nth-disj*:
fixes
t::('*a*,'*b*) *typ-info* **and**
st::('*a*,'*b*) *typ-info-struct* **and**
ts::('*a*,'*b*) *typ-info-tuple list* **and**
y::('*a*,'*b*) *typ-info-tuple*
shows
∀ *m f s n x bs bs'*. *field-lookup t f m = Some (s,n)* ⟶ *x < size-td t* ⟶
(x < n − m ∨ x ≥ (n − m) + size-td s) ⟶ *wf-fd t* ⟶ *length bs = size-td s* ⟶ *length bs' = size-td t* ⟶
access-ti t (update-ti-t s bs v) bs' ! x
= access-ti t v bs' ! x
∀ *m f s n x bs bs'*. *field-lookup-struct st f m = Some (s,n)* ⟶ *x < size-td-struct st* ⟶
(x < n − m ∨ x ≥ (n − m) + size-td s) ⟶ *wf-fd-struct st* ⟶*length bs = size-td s* ⟶ *length bs' = size-td-struct st* ⟶
access-ti-struct st (update-ti-t s bs v) bs' ! x
= access-ti-struct st v bs' ! x
∀ *m f s n x bs bs'*. *field-lookup-list ts f m = Some (s,n)* ⟶ *x < size-td-list ts* ⟶
(x < n − m ∨ x ≥ (n − m) + size-td s) ⟶ *wf-fd-list ts* ⟶*length bs = size-td s* ⟶ *length bs' = size-td-list ts* ⟶
access-ti-list ts (update-ti-t s bs v) bs' ! x
= access-ti-list ts v bs' ! x
∀ *m f s n x bs bs'*. *field-lookup-tuple y f m = Some (s,n)* ⟶ *x < size-td-tuple y* ⟶
(x < n − m ∨ x ≥ (n − m) + size-td s) ⟶ *wf-fd-tuple y* ⟶*length bs = size-td s* ⟶ *length bs' = size-td-tuple y* ⟶
access-ti-tuple y (update-ti-t s bs v) bs' ! x
= access-ti-tuple y v bs' ! x
⟨*proof*⟩

lemma *field-access-update-nth-disjD*:
⟦ *field-lookup t f m = Some (s,n)*; *x < size-td t*;
(x < n − m ∨ x ≥ (n − m) + size-td s); *wf-fd t*;
length bs = size-td s; *length bs' = size-td t* ⟧ ⟹
access-ti t (update-ti-t s bs v) bs' ! x
= access-ti t v bs' ! x
⟨*proof*⟩

lemma *intvl-cut*:
⟦ *(x::addr) ∈ {p..+m}*; *x ∉ {p+of-nat k..+n}*; *m < addr-card* ⟧ ⟹
unat (x − p) < k ∨ k + n ≤ unat (x − p)

⟨*proof*⟩

lemma *singleton-t-mask-out*:
⟦ *field-lookup* (*typ-info-t TYPE*('*a*::*mem-type*)) *f 0* = *Some* (*s*,*n*);
export-uinfo s = *typ-uinfo-t TYPE*('*b*);
K = (*UNIV* − *s-footprint-untyped* &(*p*→*f*) (*export-uinfo s*)) ⟧ ⟹
singleton-t p (*update-ti-t s* (*to-bytes-p* (*w*::'*b*::*mem-type*)) (*v*::'*a*)) |‘ *K* =
singleton-t p v |‘ *K*
⟨*proof*⟩

lemma *singleton-t-SIndexTyp*:
singleton-t p v (*x*,*SIndexTyp n*) = *singleton-t p undefined* (*x*,*SIndexTyp n*)
⟨*proof*⟩

lemma *proj-d-singleton-t*:
proj-d (*singleton-t p* (*v*::'*a*::*mem-type*) ++ *x*) = *proj-d* (*singleton-t p undefined* ++ *x*)
⟨*proof*⟩

lemma *from-bytes-heap-list-s-update*:
⟦ *field-lookup* (*typ-info-t TYPE*('*a*::*mem-type*)) *f 0* = *Some* (*s*, *n*);
export-uinfo s = *typ-uinfo-t TYPE*('*b*::*mem-type*);
dom x = *s-footprint p* − *fs-footprint p F*; *f* ∈ *F* ⟧ ⟹
from-bytes (*heap-list-s* (*singleton-t p* (*update-ti-t s* (*to-bytes-p* (*w*::'*b*)) (*v*::'*a*)) ++ *x*)
(*size-of TYPE*('*a*)) (*ptr-val p*)) =
update-ti-t s (*to-bytes-p w*)
(*from-bytes* (*heap-list-s* (*singleton-t p v* ++ *x*) (*size-of TYPE*('*a*)) (*ptr-val p*)))
⟨*proof*⟩

lemma *mfs-sep-map-field-update*:
⟦ *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*); *f* ∈ *F*;
export-uinfo s = *typ-uinfo-t TYPE*('*b*) ⟧ ⟹
(*p* ↦$_g{}^F$ *update-ti-t s* (*to-bytes-p* (*w*::'*b*::*mem-type*)) *v*) =
(*p* ↦$_g{}^F$ *update-ti-t s* (*to-bytes-p* (*u*::'*b*::*mem-type*)) (*v*::'*a*::*mem-type*))
⟨*proof*⟩

lemma *mfs-sep-map-field-update-v*:
⟦*field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*, *n*); *f* ∈ *F*;
disjoint-fn f (*F* − {*f*}); *guard-mono g g*′;
export-uinfo t = *typ-uinfo-t TYPE*('*b*) ⟧ ⟹
p ↦$_g{}^F$ *update-ti-t t* (*to-bytes-p* (*w*::'*b*::*mem-type*)) (*v*::'*a*::*mem-type*) = *p* ↦$_g{}^F$ *v*
⟨*proof*⟩

lemma *sep-map-field-fold*:
⟦ *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*,*n*);
f ∈ *F*; *disjoint-fn f* (*F* − {*f*}); *guard-mono g g*′;

export-uinfo t = typ-uinfo-t TYPE('b) ⟧ ⟹
(p ↦$_g{}^F$ *(v::'a::mem-type)* ∧*
Ptr &(p→f) ↦$_{g'}$ *(w::'b::mem-type))*
= p ↦$_g{}^{(F - \{f\})}$ *(update-ti-t t (to-bytes-p w) v)*
⟨*proof*⟩

lemma *norm-bytes*:
length bs = size-of TYPE('a) ⟹
to-bytes-p ((from-bytes bs)::'a) = norm-bytes TYPE('a::mem-type) bs
⟨*proof*⟩

lemma *sep-heap-update-global-super-fl*:
⟦ *(p* ↦$_g$ *u* ∧* *R) (lift-state (h,d))*;
field-lookup (typ-info-t TYPE('b::mem-type)) f 0 = Some (t,n);
export-uinfo t = (typ-uinfo-t TYPE('a)) ⟧ ⟹
((p ↦$_g$ *update-ti-t t (to-bytes-p v) u)* ∧* *R)*
(lift-state (heap-update (Ptr &(p→f)) (v::'a::mem-type) h,d))
⟨*proof*⟩

lemma *sep-cut'-dom*:
sep-cut' x y s ⟹ *dom s = {(a,b). a* ∈ *{x..+y}}*
⟨*proof*⟩

lemma *dom-exact-sep-cut'*:
dom-exact (sep-cut' x y)
⟨*proof*⟩

lemma *dom-lift-state-dom-s* [*simp*]:
dom (lift-state (h,d)) = dom-s d
⟨*proof*⟩

lemma *dom-ptr-retyp-empty-htd* [*simp*]:
dom (lift-state (h,ptr-retyp (p::'a::mem-type ptr) empty-htd)) = s-footprint p
⟨*proof*⟩

lemma *ptr-retyp-sep-cut'-exc*:
fixes *p::'a::mem-type ptr*
assumes *sc*: *(sep-cut' (ptr-val p) (size-of TYPE('a)) ∧* P) (lift-state (h,d))* **and**
g p
shows *(g* ⊢$_s$ *p* ∧* *sep-true* ∧* *P) (lift-state (h,(ptr-retyp p d)))*
⟨*proof*⟩

lemma *sep-cut-dom*:
sep-cut x y s ⟹ *dom s = {(a,b). a* ∈ *{x..+unat y}}*
⟨*proof*⟩

lemma *sep-cut-0* [*simp*]:
sep-cut p 0 = □
⟨*proof*⟩

lemma *heap-merge-restrict-dom-un*:
dom s = *P* ∪ *Q* ⟹ (*s*|‘*P*) ++ (*s*|‘*Q*) = *s*
⟨*proof*⟩

lemma *sep-cut-split*:
assumes *sc*: *sep-cut p y s* **and** *le*: *x* ≤ *y*
shows (*sep-cut p x* ∧* *sep-cut* (*p* + *x*) (*y* − *x*)) *s*
⟨*proof*⟩

lemma *tagd-ptr-safe-exc*:
(*g* ⊢*s p* ∧* *sep-true*) (*lift-state* (*h*,*d*)) ⟹ *ptr-safe p d*
⟨*proof*⟩

lemma *sep-map′-ptr-safe-exc*:
(*p* ↪*g* (*v*::′*a*::*mem-type*)) (*lift-state* (*h*,*d*)) ⟹ *ptr-safe p d*
⟨*proof*⟩

end

theory *SepInv*
imports *SepCode*
begin

definition *inv-footprint* :: ′*a*::*c-type ptr* ⇒ *heap-assert* **where**
inv-footprint p ≡
λ*s*. *dom s* = {(*x*,*y*). *x* ∈ {*ptr-val p*..+*size-of TYPE*(′*a*)}} − *s-footprint p*

Like in Separation.thy, these arrows are defined using bsub and esub but have an *input* syntax abbreviation with just sub. See original comment there for justification.

definition
sep-map-inv :: ′*a*::*c-type ptr* ⇒ ′*a ptr-guard* ⇒ ′*a* ⇒ *heap-assert*
(‹(‹*open-block notation*=‹*mixfix sep-map-inv*››- ↦*i*_ -)› [*56*,*0*,*51*] *56*)
where
p ↦*i g v* ≡ *p* ↦*g v* ∧* *inv-footprint p*

notation (*input*)
sep-map-inv (‹(‹*open-block notation*=‹*mixfix sep-map-inv*››- ↦*i*_ -)› [*56*,*1000*,*51*] *56*)

definition
sep-map-any-inv :: ′*a* ::*c-type ptr* ⇒ ′*a ptr-guard* ⇒ *heap-assert*
(‹(‹*open-block notation*=‹*mixfix sep-map-any-inv*››- ↦*i*_ −)› [*56*,*0*] *56*)

where
$p \mapsto^i g - \equiv p \mapsto_g - \wedge^*$ *inv-footprint p*

notation (*input*)
sep-map-any-inv (‹(‹*open-block notation*=‹*mixfix sep-map-any-inv*››- $\mapsto^i$_ −)› [56,0] 56)

definition
sep-map′-inv :: *'a*::*c-type ptr* ⇒ *'a ptr-guard* ⇒ *'a* ⇒ *heap-assert*
(‹(‹*open-block notation*=‹*mixfix sep-map′-inv*››- $\hookrightarrow^i$_ -)› [56,0,51] 56)
where
$p \hookrightarrow^i g\ v \equiv p \hookrightarrow_g v \wedge^*$ *inv-footprint p*

notation (*input*)
sep-map′-inv (‹(‹*open-block notation*=‹*mixfix sep-map′-inv*››- $\hookrightarrow^i$_ -)› [56,1000,51] 56)

definition
sep-map′-any-inv :: *'a*::*c-type ptr* ⇒ *'a ptr-guard* ⇒ *heap-assert*
(‹(‹*open-block notation*=‹*mixfix sep-map′-any-inv*››- $\hookrightarrow^i$_ −)› [56,0] 56)
where
$p \hookrightarrow^i g - \equiv p \hookrightarrow_g - \wedge^*$ *inv-footprint p*

notation (*input*)
sep-map′-any-inv (‹(‹*open-block notation*=‹*mixfix sep-map′-any-inv*››- $\hookrightarrow^i$_ −)› [56,0] 56)

definition
tagd-inv :: *'a ptr-guard* ⇒ *'a*::*c-type ptr* ⇒ *heap-assert* (**infix** ‹$\vdash_s{}^i$› 100)
where
$g \vdash_s{}^i p \equiv g \vdash_s p \wedge^*$ *inv-footprint p*

—-

lemma *sep-map′-g*:
$(p \hookrightarrow^i{}_g\ v)\ s \Longrightarrow g\ p$
⟨*proof*⟩

lemma *sep-map′-unfold*:
$(p \hookrightarrow^i{}_g\ v) = ((p \hookrightarrow^i{}_g\ v) \wedge^*$ *sep-true*$)$
⟨*proof*⟩

lemma *sep-map′-any-unfold*:
$(i \hookrightarrow^i{}_g -) = ((i \hookrightarrow^i{}_g -) \wedge^*$ *sep-true*$)$
⟨*proof*⟩

lemma *sep-map′-conjE1*:
$[\![(P \wedge^* Q)\ s; \bigwedge s.\ P\ s \Longrightarrow (i \hookrightarrow^i{}_g\ v)\ s]\!] \Longrightarrow (i \hookrightarrow^i{}_g\ v)\ s$
⟨*proof*⟩

lemma *sep-map′-conjE2*:
⟦ (*P* ∧* *Q*) *s*; ⋀*s*. *Q s* ⟹ (*i* ↪$^{i}_{g}$ *v*) *s* ⟧ ⟹ (*i* ↪$^{i}_{g}$ *v*) *s*
⟨*proof*⟩

lemma *sep-map′-any-conjE1*:
⟦ (*P* ∧* *Q*) *s*; ⋀*s*. *P s* ⟹ (*i* ↪$^{i}_{g}$ −) *s* ⟧ ⟹ (*i* ↪$^{i}_{g}$ −) *s*
⟨*proof*⟩

lemma *sep-map′-any-conjE2*:
⟦ (*P* ∧* *Q*) *s*; ⋀*s*. *Q s* ⟹ (*i* ↪$^{i}_{g}$ −) *s* ⟧ ⟹ (*i* ↪$^{i}_{g}$ −) *s*
⟨*proof*⟩

lemma *sep-map-any-old*:
(*p* ↦$^{i}_{g}$ −) = (λ*s*. ∃ *v*. (*p* ↦$^{i}_{g}$ *v*) *s*)
⟨*proof*⟩

lemma *sep-map′-old*:
(*p* ↪$^{i}_{g}$ *v*) = ((*p* ↦$^{i}_{g}$ *v*) ∧* *sep-true*)
⟨*proof*⟩

lemma *sep-map′-any-old*:
(*p* ↪$^{i}_{g}$ −) = (λ*s*. ∃ *v*. (*p* ↪$^{i}_{g}$ *v*) *s*)
⟨*proof*⟩

lemma *sep-map-sep-map′* [*simp*]:
(*p* ↦$^{i}_{g}$ *v*) *s* ⟹ (*p* ↪$^{i}_{g}$ *v*) *s*
⟨*proof*⟩

lemmas *guardI* = *sep-map′-g*[*OF sep-map-sep-map′*]

lemma *sep-map-anyI* [*simp*]:
(*p* ↦$^{i}_{g}$ *v*) *s* ⟹ (*p* ↦$^{i}_{g}$ −) *s*
⟨*proof*⟩

lemma *sep-map-anyD*:
(*p* ↦$^{i}_{g}$ −) *s* ⟹ ∃ *v*. (*p* ↦$^{i}_{g}$ *v*) *s*
⟨*proof*⟩

lemma *sep-conj-mapD*:
((*i* ↦$^{i}_{g}$ *v*) ∧* *P*) *s* ⟹ (*i* ↪$^{i}_{g}$ *v*) *s* ∧ ((*i* ↦$^{i}_{g}$ −) ∧* *P*) *s*
⟨*proof*⟩

lemma *sep-map′-ptr-safe*:
(*p* ↪$^{i}_{g}$ (*v*::′*a*::*mem-type*)) (*lift-state* (*h*,*d*)) ⟹ *ptr-safe p d*
⟨*proof*⟩

lemmas *sep-map-ptr-safe* = *sep-map′-ptr-safe*[*OF sep-map-sep-map′*]

lemma *sep-map-any-ptr-safe*:

fixes $p::'a::mem\text{-}type\ ptr$
shows $(p \mapsto^i_g -)\ (lift\text{-}state\ (h,\ d)) \Longrightarrow ptr\text{-}safe\ p\ d$
⟨*proof*⟩

lemma *sep-heap-update'*:
$(g \vdash_s{}^i\ p \wedge^*\ (p \mapsto^i_g\ v \longrightarrow^*\ P))\ (lift\text{-}state\ (h,d)) \Longrightarrow$
$P\ (lift\text{-}state\ (heap\text{-}update\ p\ (v::'a::mem\text{-}type)\ h,d))$
⟨*proof*⟩

lemma *tagd-g*:
$(g \vdash_s{}^i\ p \wedge^*\ P)\ s \Longrightarrow g\ p$
⟨*proof*⟩

lemma *tagd-ptr-safe*:
$(g \vdash_s{}^i\ p \wedge^*\ sep\text{-}true)\ (lift\text{-}state\ (h,d)) \Longrightarrow ptr\text{-}safe\ p\ d$
⟨*proof*⟩

lemma *sep-map-tagd*:
$(p \mapsto^i_g\ (v::'a::mem\text{-}type))\ s \Longrightarrow (g \vdash_s{}^i\ p)\ s$
⟨*proof*⟩

lemma *sep-map-any-tagd*:
$(p \mapsto^i_g\ -)\ s \Longrightarrow (g \vdash_s{}^i\ (p::'a::mem\text{-}type\ ptr))\ s$
⟨*proof*⟩

lemma *sep-heap-update*:
$[\![\ (p \mapsto^i_g\ - \wedge^*\ (p \mapsto^i_g\ v \longrightarrow^*\ P))\ (lift\text{-}state\ (h,d))\ ]\!] \Longrightarrow$
$P\ (lift\text{-}state\ (heap\text{-}update\ p\ (v::'a::mem\text{-}type)\ h,d))$
⟨*proof*⟩

lemma *sep-heap-update-global'*:
$(g \vdash_s{}^i\ p \wedge^*\ R)\ (lift\text{-}state\ (h,d)) \Longrightarrow$
$((p \mapsto^i_g\ v) \wedge^*\ R)\ (lift\text{-}state\ (heap\text{-}update\ p\ (v::'a::mem\text{-}type)\ h,d))$
⟨*proof*⟩

lemma *sep-heap-update-global*:
$(p \mapsto^i_g\ - \wedge^*\ R)\ (lift\text{-}state\ (h,d)) \Longrightarrow$
$((p \mapsto^i_g\ v) \wedge^*\ R)\ (lift\text{-}state\ (heap\text{-}update\ p\ (v::'a::mem\text{-}type)\ h,d))$
⟨*proof*⟩

lemma *sep-heap-update-global-super-fl-inv*:
$[\![\ (p \mapsto^i_g\ u \wedge^*\ R)\ (lift\text{-}state\ (h,d));$
$field\text{-}lookup\ (typ\text{-}info\text{-}t\ TYPE('b::mem\text{-}type))\ f\ 0 = Some\ (t,n);$
$export\text{-}uinfo\ t = (typ\text{-}uinfo\text{-}t\ TYPE('a))\ ]\!] \Longrightarrow$
$((p \mapsto^i_g\ update\text{-}ti\text{-}t\ t\ (to\text{-}bytes\text{-}p\ v)\ u) \wedge^*\ R)$
$(lift\text{-}state\ (heap\text{-}update\ (Ptr\ \&(p\rightarrow f))\ (v::'a::mem\text{-}type)\ h,d))$
⟨*proof*⟩

lemma *sep-map'-inv*:

$(p \hookrightarrow^i_g\ v)\ s \Longrightarrow (p \hookrightarrow_g\ v)\ s$
⟨*proof*⟩

lemma *sep-map'-lift*:
$(p \hookrightarrow^i_g\ (v::'a::mem\text{-}type))\ (lift\text{-}state\ (h,d)) \Longrightarrow lift\ h\ p = v$
⟨*proof*⟩

lemma *sep-map-lift*:
$((p::'a::mem\text{-}type\ ptr) \mapsto^i_g\ -)\ (lift\text{-}state\ (h,d)) \Longrightarrow$
$(p \mapsto^i_g\ lift\ h\ p)\ (lift\text{-}state\ (h,d))$
⟨*proof*⟩

lemma *sep-map-lift-wp*:
$[\![\ \exists v.\ (p \mapsto^i_g\ v \wedge^*\ (p \mapsto^i_g\ v \longrightarrow^*\ P\ v))\ (lift\text{-}state\ (h,d))\ ]\!]$
$\Longrightarrow P\ (lift\ h\ (p::'a::mem\text{-}type\ ptr))\ (lift\text{-}state\ (h,d))$
⟨*proof*⟩

lemma *sep-map'-anyI* [*simp*]:
$(p \hookrightarrow^i_g\ v)\ s \Longrightarrow (p \hookrightarrow^i_g\ -)\ s$
⟨*proof*⟩

lemma *sep-map'-anyD*:
$(p \hookrightarrow^i_g\ -)\ s \Longrightarrow \exists v.\ (p \hookrightarrow^i_g\ v)\ s$
⟨*proof*⟩

lemma *sep-map'-lift-rev*:
$[\![\ lift\ h\ p = (v::'a::mem\text{-}type);\ (p \hookrightarrow^i_g\ -)\ (lift\text{-}state\ (h,d))\ ]\!] \Longrightarrow$
$(p \hookrightarrow^i_g\ v)\ (lift\text{-}state\ (h,d))$
⟨*proof*⟩

lemma *sep-map'-any-g*:
$(p \hookrightarrow^i_g\ -)\ s \Longrightarrow g\ p$
⟨*proof*⟩

lemma *any-guardI*:
$(p \mapsto^i_g\ -)\ s \Longrightarrow g\ p$
⟨*proof*⟩

lemma *sep-map-sep-map-any*:
$(p \mapsto^i_g\ v)\ s \Longrightarrow (p \mapsto^i_g\ -)\ s$
⟨*proof*⟩

lemma *sep-lift-exists*:
fixes $p :: 'a::mem\text{-}type\ ptr$
assumes *ex*: $((\lambda s.\ \exists v.\ (p \hookrightarrow^i_g\ v)\ s \wedge P\ v\ s) \wedge^*\ Q)\ (lift\text{-}state\ (h,d))$
shows $(P\ (lift\ h\ p) \wedge^*\ Q)\ (lift\text{-}state\ (h,d))$
⟨*proof*⟩

lemma *sep-map-dom*:
(*p* $\mapsto^i_g$ (*v*::'*a*::*c-type*)) *s* $\Longrightarrow$ *dom s* = {(*a*,*b*). *a* ∈ {*ptr-val p*..+*size-of TYPE*('*a*)}}
⟨*proof*⟩

lemma *sep-map'-dom*:
(*p* $\hookrightarrow^i_g$ (*v*::'*a*::*mem-type*)) *s* $\Longrightarrow$ (*ptr-val p*,*SIndexVal*) ∈ *dom s*
⟨*proof*⟩

lemma *sep-map'-inj*:
⟦ (*p* $\hookrightarrow^i_g$ (*v*::'*a*::*c-type*)) *s*; (*p* $\hookrightarrow^i_h$ *v'*) *s* ⟧ $\Longrightarrow$ *v*=*v'*
⟨*proof*⟩

lemma *ptr-retyp-sep-cut'*:
fixes *p*::'*a*::*mem-type ptr*
assumes *sc*: (*sep-cut'* (*ptr-val p*) (*size-of TYPE*('*a*)) ∧* *P*)
(*lift-state* (*h*,*d*)) **and** *g p*
shows (*g* $\vdash_s{}^i$ *p* ∧* *P*) (*lift-state* (*h*,(*ptr-retyp p d*)))
⟨*proof*⟩

lemma *ptr-retyp-sep-cut'-wp*:
⟦ (*sep-cut'* (*ptr-val p*) (*size-of TYPE*('*a*)) ∧* (*g* $\vdash_s{}^i$ *p* $\longrightarrow^*$ *P*))
(*lift-state* (*h*,*d*)); *g* (*p*::'*a*::*mem-type ptr*) ⟧
$\Longrightarrow$ *P* (*lift-state* (*h*,(*ptr-retyp p d*)))
⟨*proof*⟩

lemma *tagd-dom*:
(*g* $\vdash_s{}^i$ *p*) *s* $\Longrightarrow$ *dom s* = {(*a*,*b*). *a* ∈ {*ptr-val* (*p*::'*a*::*c-type ptr*)..+*size-of TYPE*('*a*)}}
⟨*proof*⟩

lemma *tagd-dom-p*:
(*g* $\vdash_s{}^i$ *p*) *s* $\Longrightarrow$ (*ptr-val* (*p*::'*a*::*mem-type ptr*),*SIndexVal*) ∈ *dom s*
⟨*proof*⟩

end

theory *SepTactic*
imports *SepInv*
begin

11.24 *sep-point-tac*

definition *sep-conj-extract* :: *heap-assert* ⇒ *heap-assert* **where**
sep-conj-extract ≡ *id*

definition *sep-points* :: *heap-assert* ⇒ *heap-assert* **where**

sep-points $P \equiv P \wedge^* \textit{sep-true}$

lemma *sep-conj-extract1D*:
$(P \wedge^* Q)\ s \Longrightarrow \textit{sep-conj-extract}\ (P \wedge^* Q)\ s$
⟨*proof*⟩

lemma *sep-conj-extract2D*:
$\textit{sep-conj-extract}\ P\ s \Longrightarrow P\ s \wedge (\textit{sep-conj-extract}\ P \wedge^* \textit{sep-true})\ s$
⟨*proof*⟩

lemma *sep-conj-extract-assoc*:
$\textit{sep-conj-extract}\ ((P \wedge^* Q) \wedge^* R) = \textit{sep-conj-extract}\ (P \wedge^* (Q \wedge^* R))$
⟨*proof*⟩

lemma *sep-conj-extract-decomposeD*:
$(\textit{sep-conj-extract}\ (P \wedge^* Q) \wedge^* \textit{sep-true})\ s \Longrightarrow \textit{sep-points}\ P\ s \wedge$
$(\textit{sep-conj-extract}\ Q \wedge^* \textit{sep-true})\ s$
⟨*proof*⟩

lemma *sep-conj-extract-decomposeD2*:
$(\textit{sep-conj-extract}\ P \wedge^* \textit{sep-true})\ s \Longrightarrow \textit{sep-points}\ P\ s$
⟨*proof*⟩

lemma *sep-point-mapD*:
$\textit{sep-points}\ (p \mapsto^i_g v)\ s \Longrightarrow (p \hookrightarrow^i_g v)\ s$
⟨*proof*⟩

lemma *sep-point-map-excD*:
$\textit{sep-points}\ (p \mapsto_g v)\ s \Longrightarrow (p \hookrightarrow_g v)\ s$
⟨*proof*⟩

lemma *sep-point-otherD*:
$\textit{sep-points}\ P\ s \Longrightarrow \textit{True}$
⟨*proof*⟩

⟨*ML*⟩

lemma $(P \wedge^* p \mapsto_g v \wedge^* Q)\ s \Longrightarrow (p \hookrightarrow_g v)\ s$
⟨*proof*⟩

11.25 *sep-exists-tac*

⟨*ML*⟩

lemma $(P \wedge^* (\lambda s.\ (\exists x.\ Q\ x\ s)))\ s \Longrightarrow \exists x.\ (P \wedge^* Q\ x)\ s$
⟨*proof*⟩

11.26 *sep-select-tac*

definition *sep-mark* :: *heap-assert* ⇒ *heap-assert* **where**
sep-mark ≡ *id*

definition *sep-mark2* :: *heap-assert* ⇒ *heap-assert* **where**
sep-mark2 ≡ *id*

lemma *sep-mark2-id*:
sep-mark2 P = *P*
⟨*proof*⟩

lemma *sep-markI*:
(□ ∧* *sep-mark* (*P* ∧* *Q*)) *s* ⟹ (*P* ∧* *Q*) *s*
⟨*proof*⟩

lemma *sep-mark-match*:
(*R* ∧* *sep-mark2 P* ∧* *Q*) *s* ⟹ (*R* ∧* *sep-mark* (*P* ∧* *Q*)) *s*
⟨*proof*⟩

lemma *sep-mark-match2*:
(*R* ∧* *sep-mark2 P*) *s* ⟹ (*R* ∧* *sep-mark P*) *s*
⟨*proof*⟩

lemma *sep-mark-mismatch*:
((*R* ∧* *P*) ∧* *sep-mark Q*) *s* ⟹ (*R* ∧* *sep-mark* (*P* ∧* *Q*)) *s*
⟨*proof*⟩

lemma *sep-mark-mismatch2*:
(*R* ∧* *P*) *s* ⟹ (*R* ∧* *sep-mark P*) *s*
⟨*proof*⟩

lemma *sep-emp-rem*:
P s ⟹ (□ ∧* *P*) *s*
⟨*proof*⟩

lemma *sep-mark2-left*:
(*P* ∧* (*sep-mark2 Q* ∧* *R*)) = (*sep-mark2 Q* ∧* (*P* ∧* *R*))
⟨*proof*⟩

lemma *sep-mark2-left2*:
(*P* ∧* *sep-mark2 Q*) = (*sep-mark2 Q* ∧* *P*)
⟨*proof*⟩

⟨*ML*⟩

lemma
⋀*R x f n*. ((*P*::*heap-assert*) ∧* *fac x n* ∧* *R* (*f x*)) *s*
⟨*proof*⟩

lemma
((*P*::*heap-assert*) $\wedge^*$ *fac x n*) *s*
⟨*proof*⟩

lemma
((*P*::*heap-assert*) $\wedge^*$ *long-name*) *s*
⟨*proof*⟩

lemma *sep-heap-update'-hrs*:
(*c-guard* $\vdash_s{}^i$ *p* $\wedge^*$ (*p* $\mapsto^i{}_c$-*guard v* $\longrightarrow^*$ *P*)) (*lift-state s*) $\Longrightarrow$
P (*lift-state* (*hrs-mem-update* (*heap-update p* (*v*::'*a*::*mem-type*)) *s*))
⟨*proof*⟩

lemma *sep-map-lift-wp-hrs*:
⟦ $\exists$ *v*. (*p* $\mapsto^i{}_c$-*guard v* $\wedge^*$ (*p* $\mapsto^i{}_c$-*guard v* $\longrightarrow^*$ *P v*)) (*lift-state s*) ⟧
$\Longrightarrow$ *P* (*lift* (*hrs-mem s*) (*p*::'*a*::*mem-type ptr*)) (*lift-state s*)
⟨*proof*⟩

lemma *ptr-retyp-sep-cut'-wp-hrs*:
⟦ (*sep-cut'* (*ptr-val p*) (*size-of TYPE*('*a*)) $\wedge^*$ (*c-guard* $\vdash_s{}^i$ *p* $\longrightarrow^*$ *P*))
(*lift-state s*); *c-guard* (*p*::'*a*::*mem-type ptr*) ⟧
$\Longrightarrow$ *P* (*lift-state* (*hrs-htd-update* (*ptr-retyp p*) *s*))
⟨*proof*⟩

⟨*ML*⟩

lemma
((λ*z*. *lift* (*hrs-mem s*) (*p*::(*32 word*) *ptr*) + *1* = *2*) $\wedge^*$ *P*) (*lift-state s*)
⟨*proof*⟩

end

theory *SepFrame*
imports *SepTactic*
begin

class *heap-state-type'*

instance *heap-state-type'* $\subseteq$ *type* ⟨*proof*⟩

consts
hst-mem :: *'a*::*heap-state-type'* ⇒ *heap-mem*
hst-mem-update :: (*heap-mem* ⇒ *heap-mem*) ⇒ *'a*::*heap-state-type'* ⇒ *'a*
hst-htd :: *'a*::*heap-state-type'* ⇒ *heap-typ-desc*
hst-htd-update :: (*heap-typ-desc* ⇒ *heap-typ-desc*) ⇒ *'a*::*heap-state-type'* ⇒ *'a*

class *heap-state-type* = *heap-state-type'* +
assumes *hst-htd-htd-update* [*simp*]: *hst-htd* (*hst-htd-update d s*) = *d* (*hst-htd s*)
assumes *hst-mem-mem-update* [*simp*]: *hst-mem* (*hst-mem-update h s*) = *h* (*hst-mem s*)
assumes *hst-htd-mem-update* [*simp*]: *hst-htd* (*hst-mem-update h s*) = *hst-htd s*
assumes *hst-mem-htd-update* [*simp*]: *hst-mem* (*hst-htd-update d s*) = *hst-mem s*

translations
s(| *hst-mem* := *x*|) <= *CONST hst-mem-update* (*K-record x*) *s*
s(| *hst-htd* := *x*|) <= *CONST hst-htd-update* (*K-record x*) *s*

definition *lift-hst* :: *'a*::*heap-state-type'* ⇒ *heap-state* **where**
lift-hst s ≡ *lift-state* (*hst-mem s*,*hst-htd s*)

definition *point-eq-mod* :: (*'a* ⇒ *'b*) ⇒ (*'a* ⇒ *'b*) ⇒ *'a set* ⇒ *bool* **where**
point-eq-mod f g X ≡ ∀ *x*. *x* ∉ *X* ⟶ *f x* = *g x*

definition *exec-fatal* :: (*'s*,*'b*,*'c*) *com* ⇒ (*'s*,*'b*,*'c*) *body* ⇒ *'s* ⇒ *bool* **where**
exec-fatal C Γ *s* ≡ (∃ *f*. Γ ⊢ ⟨*C*,*Normal s*⟩ ⇒ *Fault f*) ∨ (Γ ⊢ ⟨*C*,*Normal s*⟩ ⇒ *Stuck*)

definition *restrict-htd* :: *'s*::*heap-state-type'* ⇒ *s-addr set* ⇒ *'s* **where**
restrict-htd s X ≡ *s*(| *hst-htd* := *restrict-s* (*hst-htd s*) *X* |)

definition *restrict-safe-OK* ::
's ⇒ *'s* ⇒ (*'s* ⇒ (*'s*,*'c*) *xstate*) ⇒ *s-addr set* ⇒ (*'s*::*heap-state-type'*,*'b*,*'c*) *com* ⇒
(*'s*,*'b*,*'c*) *body* ⇒ *bool* **where**
restrict-safe-OK s t f X C Γ ≡
Γ ⊢ ⟨*C*,(*Normal* (*restrict-htd s X*))⟩ ⇒ *f* (*restrict-htd t X*) ∧
point-eq-mod (*lift-state* (*hst-mem t*,*hst-htd t*)) (*lift-state* (*hst-mem s*,*hst-htd s*)) *X*

definition *restrict-safe* ::
's ⇒ (*'s*,*'c*) *xstate* ⇒ (*'s*::*heap-state-type'*,*'b*,*'c*) *com* ⇒ (*'s*,*'b*,*'c*) *body* ⇒ *bool* **where**
restrict-safe s t C Γ ≡
∀ *X*. (*case t of*
Normal t' ⇒ *restrict-safe-OK s t' Normal X C* Γ
| *Abrupt t'* ⇒ *restrict-safe-OK s t' Abrupt X C* Γ
| - ⇒ *False*) ∨
exec-fatal C Γ (*restrict-htd s X*)

definition *mem-safe* :: (*'s*::{*heap-state-type'*,*type*},*'b*,*'c*) *com* ⇒ (*'s*,*'b*,*'c*) *body* ⇒

bool **where**
mem-safe C Γ ≡ ∀ s t. Γ ⊢ ⟨C,Normal s⟩ ⇒ t ⟶ restrict-safe s t C Γ

definition *point-eq-mod-safe* ::
's::{heap-state-type',type} set ⇒ ('s ⇒ 's) ⇒ ('s ⇒ s-addr ⇒ 'c) ⇒ bool **where**
point-eq-mod-safe P f g ≡ ∀ s X. restrict-htd s X ∈ P ⟶ point-eq-mod (g (f s)) (g s) X

definition *comm-restrict* :: *('s::{heap-state-type',type} ⇒ 's) ⇒ 's ⇒ s-addr set ⇒ bool* **where**
comm-restrict f s X ≡ f (restrict-htd s X) = restrict-htd (f s) X

definition *comm-restrict-safe* :: *'s set ⇒ ('s::{heap-state-type',type} ⇒ 's) ⇒ bool* **where**
comm-restrict-safe P f ≡ ∀ s X. restrict-htd s X ∈ P ⟶ comm-restrict f s X

definition *htd-ind* :: *('a::{heap-state-type',type} ⇒ 'b) ⇒ bool* **where**
htd-ind f ≡ ∀ x s. f s = f (hst-htd-update x s)

definition *mono-guard* :: *'s::{heap-state-type',type} set ⇒ bool* **where**
mono-guard P ≡ ∀ s X. restrict-htd s X ∈ P ⟶ s ∈ P

definition *expr-htd-ind* :: *'s::{heap-state-type',type} set ⇒ bool* **where**
expr-htd-ind P ≡ ∀ d s. s⦇ hst-htd := d ⦈ ∈ P = (s ∈ P)

primrec *intra-safe* :: *('s::heap-state-type','b,'c) com ⇒ bool*
where
intra-safe Skip = True
| *intra-safe (Basic f) = (comm-restrict-safe UNIV f ∧*
point-eq-mod-safe UNIV f (λs. lift-state (hst-mem s,hst-htd s)))
| *intra-safe (Spec r) = (∀ Γ. mem-safe (Spec r) (Γ :: ('s,'b,'c) body))*
| *intra-safe (Seq C D) = (intra-safe C ∧ intra-safe D)*
| *intra-safe (Cond P C D) = (expr-htd-ind P ∧ intra-safe C ∧ intra-safe D)*
| *intra-safe (While P C) = (expr-htd-ind P ∧ intra-safe C)*
| *intra-safe (Call p) = True*
| *intra-safe (DynCom f) = (htd-ind f ∧ (∀ s. intra-safe (f s)))*
| *intra-safe (Guard f P C) = (mono-guard P ∧ (case C of*
Basic g ⇒ comm-restrict-safe P g ∧
point-eq-mod-safe P g (λs. lift-state (hst-mem s,hst-htd s))
| - ⇒ intra-safe C))
| *intra-safe Throw = True*
| *intra-safe (Catch C D) = (intra-safe C ∧ intra-safe D)*

instance *state-ext* :: *(heap-state-type',type) heap-state-type' ⟨proof⟩*

overloading
hs-mem-state ≡ hst-mem
hs-mem-update-state ≡ hst-mem-update
hs-htd-state ≡ hst-htd

hs-htd-update-state ≡ *hst-htd-update*
begin
definition *hs-mem-state* [*simp*]: *hs-mem-state* ≡ *hst-mem* ∘ *globals*
definition *hs-mem-update-state* [*simp*]: *hs-mem-update-state* ≡ *globals-update* ∘ *hst-mem-update*
definition *hs-htd-state*[*simp*]: *hs-htd-state* ≡ *hst-htd* ∘ *globals*
definition *hs-htd-update-state* [*simp*]: *hs-htd-update-state* ≡ *globals-update* ∘ *hst-htd-update*
end

instance *state-ext* :: (*heap-state-type*,*type*) *heap-state-type*
⟨*proof*⟩

primrec
intra-deps :: ('*s*,'*b*,'*c*) *com* ⇒ '*b set*
where
intra-deps Skip = {}
| *intra-deps* (*Basic f*) = {}
| *intra-deps* (*Spec r*) = {}
| *intra-deps* (*Seq C D*) = (*intra-deps C* ∪ *intra-deps D*)
| *intra-deps* (*Cond P C D*) = (*intra-deps C* ∪ *intra-deps D*)
| *intra-deps* (*While P C*) = *intra-deps C*
| *intra-deps* (*Call p*) = {*p*}
| *intra-deps* (*DynCom f*) = ⋃{*intra-deps* (*f s*) | *s*. *True*}
| *intra-deps* (*Guard f P C*) = *intra-deps C*
| *intra-deps Throw* = {}
| *intra-deps* (*Catch C D*) = (*intra-deps C* ∪ *intra-deps D*)

inductive-set
proc-deps :: ('*s*,'*b*,'*c*) *com* ⇒ ('*s*,'*b*,'*c*) *body* ⇒ '*b set*
for *C* :: ('*s*,'*b*,'*c*) *com*
and Γ :: ('*s*,'*b*,'*c*) *body*
where
x ∈ *intra-deps C* ⟹ *x* ∈ *proc-deps C* Γ
| ⟦ *x* ∈ *proc-deps C* Γ; Γ *x* = *Some D*; *y* ∈ *intra-deps D* ⟧ ⟹ *y* ∈ *proc-deps C* Γ

—

lemma *point-eq-mod-refl* [*simp*]:
point-eq-mod f f X
⟨*proof*⟩

lemma *point-eq-mod-subs*:
⟦ *point-eq-mod f g Y*; *Y* ⊆ *X* ⟧ ⟹ *point-eq-mod f g X*
⟨*proof*⟩

lemma *point-eq-mod-trans*:
⟦ *point-eq-mod x y X*; *point-eq-mod y z X* ⟧ ⟹ *point-eq-mod x z X*
⟨*proof*⟩

lemma *mem-safe-NormalD*:
⟦ Γ ⊢ ⟨*C*,*Normal s*⟩ ⇒ *Normal t*; *mem-safe C* Γ;
¬ *exec-fatal C* Γ (*restrict-htd s X*) ⟧ ⟹
(Γ ⊢ ⟨*C*,(*Normal* (*restrict-htd s X*))⟩ ⇒ *Normal* (*restrict-htd t X*) ∧
point-eq-mod (*lift-state* (*hst-mem t*,*hst-htd t*))
(*lift-state* (*hst-mem s*,*hst-htd s*)) *X*)
⟨*proof*⟩

lemma *mem-safe-AbruptD*:
⟦ Γ ⊢ ⟨*C*,*Normal s*⟩ ⇒ *Abrupt t*; *mem-safe C* Γ;
¬ *exec-fatal C* Γ (*restrict-htd s X*) ⟧ ⟹
(Γ ⊢ ⟨*C*,(*Normal* (*restrict-htd s X*))⟩ ⇒ *Abrupt* (*restrict-htd t X*) ∧
point-eq-mod (*lift-state* (*hst-mem t*,*hst-htd t*))
(*lift-state* (*hst-mem s*,*hst-htd s*)) *X*)
⟨*proof*⟩

lemma *mem-safe-FaultD*:
⟦ Γ ⊢ ⟨*C*,*Normal s*⟩ ⇒ *Fault f*; *mem-safe C* Γ ⟧ ⟹
exec-fatal C Γ (*restrict-htd s X*)
⟨*proof*⟩

lemma *mem-safe-StuckD*:
⟦ Γ ⊢ ⟨*C*,*Normal s*⟩ ⇒ *Stuck*; *mem-safe C* Γ ⟧ ⟹
exec-fatal C Γ (*restrict-htd s X*)
⟨*proof*⟩

lemma *lift-state-d-restrict* [*simp*]:
lift-state (*h*,(*restrict-s d X*)) = *lift-state* (*h*,*d*) |‘ *X*
⟨*proof*⟩

lemma *dom-merge-restrict* [*simp*]:
(*x* ++ *y*) |‘ *dom y* = *y*
⟨*proof*⟩

lemma *dom-compl-restrict* [*simp*]:
x |‘ (*UNIV* − *dom x*) = *Map.empty*
⟨*proof*⟩

lemma *lift-state-point-eq-mod*:
⟦ *point-eq-mod* (*lift-state* (*h*,*d*)) (*lift-state* (*h′*,*d′*)) *X* ⟧ ⟹
lift-state (*h*,*d*) |‘ (*UNIV* − *X*) =
lift-state (*h′*,*d′*) |‘ (*UNIV* − *X*)
⟨*proof*⟩

lemma *htd-′-update-ind* [*simp*]:
htd-ind f ⟹ *f* (*hst-htd-update x s*) = *f s*
⟨*proof*⟩

lemma *sep-frame′*:

assumes *orig-spec*: $\forall s.\ \Gamma \vdash \{\!|s.\ P\ (f\ `(\lambda x.\ x))\ (\mathit{lift\text{-}hst}\ `(\lambda x.\ x))|\!\}$
C
$\{\!|Q\ (g\ s\ `(\lambda x.\ x))\ (\mathit{lift\text{-}hst}\ `(\lambda x.\ x))|\!\}$
and *hi-f*: *htd-ind f* **and** *hi-g*: *htd-ind g*
and *hi-g'*: $\forall s.$ *htd-ind* $(g\ s)$
and *safe*: *mem-safe* $(C::('s::\mathit{heap\text{-}state\text{-}type},'b,'c)\ \mathit{com})\ \Gamma$
shows $\forall s.\ \Gamma \vdash \{\!|s.\ (P\ (f\ `(\lambda x.\ x)) \wedge^*\ R\ (h\ `(\lambda x.\ x)))\ (\mathit{lift\text{-}hst}\ `(\lambda x.\ x))|\!\}$
C
$\{\!|(Q\ (g\ s\ `(\lambda x.\ x)) \wedge^*\ R\ (h\ s))\ (\mathit{lift\text{-}hst}\ `(\lambda x.\ x))|\!\}$
⟨*proof*⟩

lemma *sep-frame*:
$[\![\ k = (\lambda s.\ (\mathit{hst\text{-}mem}\ s, \mathit{hst\text{-}htd}\ s));$
$\forall s.\ \Gamma \vdash \{\!|s.\ P\ (f\ `(\lambda x.\ x))\ (\mathit{lift\text{-}state}\ (k\ `(\lambda x.\ x)))|\!\}$
C
$\{\!|Q\ (g\ s\ `(\lambda x.\ x))\ (\mathit{lift\text{-}state}\ (k\ `(\lambda x.\ x)))|\!\};$
htd-ind f; *htd-ind g*; $\forall s.$ *htd-ind* $(g\ s)$;
mem-safe $(C::('s::\mathit{heap\text{-}state\text{-}type},'b,'c)\ \mathit{com})\ \Gamma\ ]\!] \Longrightarrow$
$\forall s.\ \Gamma \vdash \{\!|s.\ (P\ (f\ `(\lambda x.\ x)) \wedge^*\ R\ (h\ `(\lambda x.\ x)))\ (\mathit{lift\text{-}state}\ (k\ `(\lambda x.\ x)))|\!\}$
C
$\{\!|(Q\ (g\ s\ `(\lambda x.\ x)) \wedge^*\ R\ (h\ s))\ (\mathit{lift\text{-}state}\ (k\ `(\lambda x.\ x)))|\!\}$
⟨*proof*⟩

lemma *point-eq-mod-safe* [*simp*]:
$[\![$ *point-eq-mod-safe P f g*; *restrict-htd s X* $\in P$; $x \notin X\ ]\!] \Longrightarrow$
$g\ (f\ s)\ x = (g\ s)\ x$
⟨*proof*⟩

lemma *comm-restrict-safe* [*simp*]:
$[\![$ *comm-restrict-safe P f*; *restrict-htd s X* $\in P\ ]\!] \Longrightarrow$
restrict-htd $(f\ s\ )\ X = f\ (\mathit{restrict\text{-}htd}\ s\ X)$
⟨*proof*⟩

lemma *mono-guardD*:
$[\![$ *mono-guard P*; *restrict-htd s X* $\in P\ ]\!] \Longrightarrow s \in P$
⟨*proof*⟩

lemma *expr-htd-ind*:
expr-htd-ind P $\Longrightarrow$ *restrict-htd s X* $\in P = (s \in P)$
⟨*proof*⟩

lemmas *exec-other-intros* = *exec.intros*(*1*−*3*) *exec.intros*(*5*−*14*) *exec.intros*(*16*−*17*) *exec.intros*(*19*−)

lemma *exec-fatal-Seq*:
exec-fatal C Γ *s* $\Longrightarrow$ *exec-fatal* $(C;;D)\ \Gamma\ s$
⟨*proof*⟩

lemma *exec-fatal-Seq2*:
⟦ Γ ⊢ ⟨*C*,*Normal s*⟩ ⇒ *Normal t*; *exec-fatal D* Γ *t* ⟧ ⟹ *exec-fatal* (*C*;;*D*) Γ *s*
⟨*proof*⟩

lemma *exec-fatal-Cond*:
exec-fatal (*Cond P C D*) Γ *s* = (*if s* ∈ *P then exec-fatal C* Γ *s else*
exec-fatal D Γ *s*)
⟨*proof*⟩

lemma *exec-fatal-While*:
⟦ *exec-fatal C* Γ *s*; *s* ∈ *P* ⟧ ⟹ *exec-fatal* (*While P C*) Γ *s*
⟨*proof*⟩

lemma *exec-fatal-While2*:
⟦ *exec-fatal* (*While P C*) Γ *t*; Γ ⊢ ⟨*C*,*Normal s*⟩ ⇒ *Normal t*; *s* ∈ *P* ⟧ ⟹
exec-fatal (*While P C*) Γ *s*
⟨*proof*⟩

lemma *exec-fatal-Call*:
⟦ Γ *p* = *Some C*; *exec-fatal C* Γ *s* ⟧ ⟹ *exec-fatal* (*Call p*) Γ *s*
⟨*proof*⟩

lemma *exec-fatal-DynCom*:
exec-fatal (*f s*) Γ *s* ⟹ *exec-fatal* (*DynCom f*) Γ *s*
⟨*proof*⟩

lemma *exec-fatal-Guard*:
exec-fatal (*Guard f P C*) Γ *s* = (*s* ∈ *P* ⟶ *exec-fatal C* Γ *s*)
⟨*proof*⟩

lemma *restrict-safe-Guard*:
assumes *restrict*: *restrict-safe s t C* Γ
shows *restrict-safe s t* (*Guard f P C*) Γ
⟨*proof*⟩

lemma *restrict-safe-Guard2*:
⟦ *s* ∉ *P*; *mono-guard P* ⟧ ⟹ *restrict-safe s* (*Fault f*) (*Guard f P C*) Γ
⟨*proof*⟩

lemma *exec-fatal-Catch*:
exec-fatal C Γ *s* ⟹ *exec-fatal* (*TRY C CATCH D END*) Γ *s*
⟨*proof*⟩

lemma *exec-fatal-Catch2*:
⟦ Γ ⊢ ⟨*C*,*Normal s*⟩ ⇒ *Abrupt t*; *exec-fatal D* Γ *t* ⟧ ⟹
exec-fatal (*TRY C CATCH D END*) Γ *s*
⟨*proof*⟩

lemma *intra-safe-restrict* [*rule-format*]:
assumes *safe-env*: $\bigwedge n\ C.\ \Gamma\ n = Some\ C \Longrightarrow intra\text{-}safe\ C$ **and**
exec: $\Gamma \vdash \langle C,s\rangle \Rightarrow t$
shows $\forall s'.\ s = Normal\ s' \longrightarrow intra\text{-}safe\ C \longrightarrow restrict\text{-}safe\ s'\ t\ C\ \Gamma$
⟨*proof*⟩

lemma *intra-mem-safe*:
$[\![\bigwedge n\ C.\ \Gamma\ n = Some\ C \Longrightarrow intra\text{-}safe\ C;\ intra\text{-}safe\ C\ ]\!] \Longrightarrow mem\text{-}safe\ C\ \Gamma$
⟨*proof*⟩

lemma *point-eq-mod-safe-triv*:
$(\bigwedge s.\ g\ (f\ s) = g\ s) \Longrightarrow point\text{-}eq\text{-}mod\text{-}safe\ P\ f\ g$
⟨*proof*⟩

lemma *comm-restrict-safe-triv*:
$(\bigwedge s\ X.\ f\ (s(\!|\ hst\text{-}htd := restrict\text{-}s\ (hst\text{-}htd\ s)\ X\ |\!)) =$
$(f\ s)(\!|\ hst\text{-}htd := restrict\text{-}s\ (hst\text{-}htd\ (f\ s))\ X\ |\!)) \Longrightarrow comm\text{-}restrict\text{-}safe\ P\ f$
⟨*proof*⟩

lemma *mono-guard-UNIV* [*simp*]:
mono-guard UNIV
⟨*proof*⟩

lemma *mono-guard-triv*:
$(\bigwedge s\ X.\ s(\!|\ hst\text{-}htd := X\ |\!) \in g \Longrightarrow s \in g) \Longrightarrow mono\text{-}guard\ g$
⟨*proof*⟩

lemma *mono-guard-triv2*:
$(\bigwedge s\ X.\ s(\!|\ hst\text{-}htd := X\ |\!) \in g = ((s::'a::heap\text{-}state\text{-}type') \in g)) \Longrightarrow$
mono-guard g
⟨*proof*⟩

lemma *dom-restrict-s*:
$x \in dom\text{-}s\ (restrict\text{-}s\ d\ X) \Longrightarrow x \in dom\text{-}s\ d \wedge x \in X$
⟨*proof*⟩

lemma *mono-guard-ptr-safe*:
$[\![\bigwedge s.\ d\ s = hst\text{-}htd\ (s::'a::heap\text{-}state\text{-}type);\ htd\text{-}ind\ p\ ]\!] \Longrightarrow$
mono-guard $\{s.\ ptr\text{-}safe\ (p\ s)\ (d\ s)\}$
⟨*proof*⟩

lemma *point-eq-mod-safe-ptr-safe-update*:
$[\![d = (hst\text{-}htd::'a::heap\text{-}state\text{-}type \Rightarrow heap\text{-}typ\text{-}desc);$
$m = (\lambda s.\ hst\text{-}mem\text{-}update\ (heap\text{-}update\ (p\ s)\ ((v\ s)::'b::mem\text{-}type))\ s);$
$h = hst\text{-}mem;\ k = (\lambda s.\ lift\text{-}state\ (h\ s,d\ s));\ htd\text{-}ind\ p\ ]\!] \Longrightarrow$
point-eq-mod-safe $\{s.\ ptr\text{-}safe\ (p\ s)\ (d\ s)\}\ m\ k$
⟨*proof*⟩

lemma *field-ti-s-sub-typ*:

field-lookup (*export-uinfo* (*typ-info-t TYPE*('*b*::*mem-type*))) *f 0* = *Some* (*typ-uinfo-t TYPE*('*a*),*b*) ⟹
s-footprint ((*Ptr* &(*p*→*f*))::'*a*::*mem-type ptr*) ⊆ *s-footprint* (*p*::'*b ptr*)
⟨*proof*⟩

lemma *ptr-safe-mono*:
⟦ *ptr-safe* (*p*::'*a*::*mem-type ptr*) *d*; *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0*
= *Some* (*t*,*n*); *export-uinfo t* = *typ-uinfo-t TYPE*('*b*) ⟧ ⟹
ptr-safe ((*Ptr* &(*p*→*f*))::'*b*::*mem-type ptr*) *d*
⟨*proof*⟩

lemma *point-eq-mod-safe-ptr-safe-update-fl*:
⟦ *d* = (*hst-htd*::'*a*::*heap-state-type* ⇒ *heap-typ-desc*);
m = (λ*s*. *hst-mem-update* (*heap-update* (*Ptr* &((*p s*)→*f*)) ((*v s*)::'*b*::*mem-type*)) *s*);
h = *hst-mem*; *k* = (λ*s*. *lift-state* (*h s*,*d s*)); *htd-ind p*;
field-lookup (*typ-info-t TYPE*('*c*)) *f 0* = *Some* (*t*,*n*);
export-uinfo t = *typ-uinfo-t TYPE*('*b*) ⟧ ⟹
point-eq-mod-safe {*s*. *ptr-safe* ((*p*::'*a* ⇒ '*c*::*mem-type ptr*) *s*) (*d s*)} *m k*
⟨*proof*⟩

context
begin

private method *m* =
(*clarsimp simp*: *ptr-retyp-d-eq-snd ptr-retyp-footprint list-map-eq*,
erule notE,
drule intvlD, *clarsimp*,
(*rule s-footprintI*; *assumption?*),
subst (*asm*) *unat-of-nat*,
(*subst* (*asm*) *mod-less*; *assumption?*),
subst len-of-addr-card,
erule less-trans,
simp)

lemma *point-eq-mod-safe-ptr-safe-tag*:
⟦ *d* = (*hst-htd*::'*a*::*heap-state-type* ⇒ *heap-typ-desc*); *h* = *hst-mem*;
m = (λ*s*. *hst-htd-update* (*ptr-retyp* (*p s*)) *s*);
k = (λ*s*. *lift-state* (*h s*,*d s*));
htd-ind p ⟧ ⟹
point-eq-mod-safe {*s*. *ptr-safe* ((*p s*)::'*b*::*mem-type ptr*) (*d s*)} *m k*
⟨*proof*⟩

end

lemma *comm-restrict-safe-ptr-safe-tag*:
fixes *d*::'*a*::*heap-state-type* ⇒ *heap-typ-desc*
assumes
fun-d: *d* = *hst-htd* **and**

fun-upd: $m = (\lambda s.\ \textit{hst-htd-update}\ (\textit{ptr-retyp}\ (p\ s))\ s)$ **and**
ind: *htd-ind p* **and**
upd: $\bigwedge d\ d'\ (s::'a).$
hst-htd-update $(d\ s)$ $(\textit{hst-htd-update}\ (d'\ s)\ s) = \textit{hst-htd-update}\ ((d\ s) \circ (d'\ s))\ s$
shows *comm-restrict-safe* $\{s.\ \textit{ptr-safe}\ ((p\ s)::'b::\textit{mem-type ptr})\ (d\ s)\}\ m$
⟨*proof*⟩

lemmas *intra-sc = hrs-comm comp-def hrs-htd-update-htd-update*
point-eq-mod-safe-triv comm-restrict-safe-triv mono-guard-triv2
mono-guard-ptr-safe point-eq-mod-safe-ptr-safe-update
point-eq-mod-safe-ptr-safe-tag comm-restrict-safe-ptr-safe-tag
point-eq-mod-safe-ptr-safe-update-ft

declare *expr-htd-ind-def* [*iff*]
declare *htd-ind-def* [*iff*]

lemma *proc-deps-Skip* [*simp*]:
$\textit{proc-deps}\ \textit{Skip}\ \Gamma = \{\}$
⟨*proof*⟩

lemma *proc-deps-Basic* [*simp*]:
$\textit{proc-deps}\ (\textit{Basic}\ f)\ \Gamma = \{\}$
⟨*proof*⟩

lemma *proc-deps-Spec* [*simp*]:
$\textit{proc-deps}\ (\textit{Spec}\ r)\ \Gamma = \{\}$
⟨*proof*⟩

lemma *proc-deps-Seq* [*simp*]:
$\textit{proc-deps}\ (\textit{Seq}\ C\ D)\ \Gamma = \textit{proc-deps}\ C\ \Gamma \cup \textit{proc-deps}\ D\ \Gamma$
⟨*proof*⟩

lemma *proc-deps-Cond* [*simp*]:
$\textit{proc-deps}\ (\textit{Cond}\ P\ C\ D)\ \Gamma = \textit{proc-deps}\ C\ \Gamma \cup \textit{proc-deps}\ D\ \Gamma$
⟨*proof*⟩

lemma *proc-deps-While* [*simp*]:
$\textit{proc-deps}\ (\textit{While}\ P\ C)\ \Gamma = \textit{proc-deps}\ C\ \Gamma$
⟨*proof*⟩

lemma *proc-deps-Guard* [*simp*]:
$\textit{proc-deps}\ (\textit{Guard}\ f\ P\ C)\ \Gamma = \textit{proc-deps}\ C\ \Gamma$
⟨*proof*⟩

lemma *proc-deps-Throw* [*simp*]:
$\textit{proc-deps}\ \textit{Throw}\ \Gamma = \{\}$
⟨*proof*⟩

lemma *proc-deps-Catch* [*simp*]:
proc-deps (*Catch C D*) Γ = *proc-deps C* Γ ∪ *proc-deps D* Γ
⟨*proof*⟩

lemma *proc-deps-Call* [*simp*]:
proc-deps (*Call p*) Γ = {*p*} ∪ (*case* Γ *p of Some C* ⇒
proc-deps C (Γ(*p* := *None*)) | - ⇒ {}) (**is** *?X* = *?Y* ∪ *?Z*)
⟨*proof*⟩

lemma *proc-deps-DynCom* [*simp*]:
proc-deps (*DynCom f*) Γ = ⋃{*proc-deps* (*f s*) Γ | *s*. *True*}
⟨*proof*⟩

lemma *proc-deps-restrict*:
proc-deps C Γ ⊆ *proc-deps C* (Γ(*p* := *None*)) ∪ *proc-deps* (*Call p*) Γ
⟨*proof*⟩

lemma *exec-restrict*:
assumes *exec*: Γ′ ⊢ ⟨*C*,*s*⟩ ⇒ *t*
shows ⋀Γ *X*. ⟦ Γ′ = Γ |‘ *X*; *proc-deps C* Γ ⊆ *X* ⟧ ⟹ Γ ⊢ ⟨*C*,*s*⟩ ⇒ *t*
⟨*proof*⟩

lemma *exec-restrict2*:
assumes *exec*: Γ ⊢ ⟨*C*,*s*⟩ ⇒ *t*
shows ⋀*X*. *proc-deps C* Γ ⊆ *X* ⟹ Γ |‘ *X* ⊢ ⟨*C*,*s*⟩ ⇒ *t*
⟨*proof*⟩

lemma *exec-restrict-eq*:
Γ |‘ *proc-deps C* Γ ⊢ ⟨*C*,*s*⟩ ⇒ *t* = Γ ⊢ ⟨*C*,*s*⟩ ⇒ *t*
⟨*proof*⟩

lemma *mem-safe-restrict*:
mem-safe C Γ = *mem-safe C* (Γ |‘ *proc-deps C* Γ)
⟨*proof*⟩

end

theory *StructSupport*
imports *SepCode SepInv*
begin

lemma *field-lookup-list-Some2*:
fn ∉ *dt-snd* ‘ *set ts* ⟹
field-lookup-list (*ts*@[*DTuple t fn d*]) (*fn* # *fs*) *m* = *field-lookup t fs* (*m* + *size-td-list ts*)
⟨*proof*⟩

lemma *field-lookup-list-mismatch*:

$f \neq fn \Longrightarrow$ *field-lookup-list* (*ts*@[*DTuple t f d*]) (*fn* # *fs*) *m* =
field-lookup-list ts (*fn* # *fs*) *m*
⟨*proof*⟩

lemma *fnl-set*:
set (*CompoundCTypes.field-names-list* (*TypDesc algn* (*TypAggregate xs*) *tn*)) = *dt-snd* ‘ *set xs*
⟨*proof*⟩

lemma *fnl-extend-ti*:
⟦ *fn* ∉ *set* (*CompoundCTypes.field-names-list tag*); *aggregate tag* ⟧ ⟹
field-lookup (*extend-ti tag t algn fn d*) (*f* # *fs*) *m* =
(*if f*=*fn then field-lookup t fs* (*size-td tag*+*m*) *else field-lookup tag* (*f* # *fs*) *m*)
⟨*proof*⟩

lemma *fnl-extend-ti-mismatch*:
f ≠ *fn* ⟹ *field-lookup* (*extend-ti tag t algn fn d*) (*f* # *fs*) *m* = *field-lookup tag* (*f* # *fs*) *m*
⟨*proof*⟩

lemma *fl-ti-pad-combine*:
⟦ *hd f* ≠ *CHR* ''!''; *aggregate tag* ⟧ ⟹
field-lookup (*ti-pad-combine n tag*) (*f*#*fs*) *m* = *field-lookup tag* (*f*#*fs*) *m*
⟨*proof*⟩

lemma *fl-ti-typ-combine*:
⟦ *fn* ∉ *set* (*CompoundCTypes.field-names-list tag*); *aggregate tag* ⟧ ⟹
field-lookup (*ti-typ-combine* (*t-b*::'*b*::*c-type itself*) *f-ab f-upd-ab algn fn tag*) (*f*#*fs*) *m* =
(*if f*=*fn*
then field-lookup (*adjust-ti* (*typ-info-t TYPE*('*b*)) *f-ab f-upd-ab*) *fs* (*size-td tag*+*m*)
else field-lookup tag (*f* # *fs*) *m*)
⟨*proof*⟩

lemma *fl-ti-typ-combine-match*:
⟦ *fn* ∉ *set* (*CompoundCTypes.field-names-list tag*); *aggregate tag* ⟧ ⟹
field-lookup (*ti-typ-combine* (*t-b*::'*b*::*c-type itself*) *f-ab f-upd-ab algn fn tag*) (*fn*#*fs*) *m* =
field-lookup (*adjust-ti* (*typ-info-t TYPE*('*b*)) *f-ab f-upd-ab*) *fs* (*size-td tag*+*m*)
⟨*proof*⟩

lemma *fl-ti-typ-combine-mismatch*:
⟦ *f* ≠ *fn* ⟧ ⟹
field-lookup (*ti-typ-combine* (*t-b*::'*b*::*c-type itself*) *f-ab f-upd-ab algn fn tag*) (*f*#*fs*) *m* =
field-lookup tag (*f* # *fs*) *m*
⟨*proof*⟩

lemma *fl-ti-typ-pad-combine*:
⟦ *fn* ∉ *set* (*CompoundCTypes.field-names-list tag*); *hd f* ≠ *CHR* ''!''; *hd fn* ≠ *CHR* ''!''; *aggregate tag* ⟧ ⟹ *field-lookup* (*ti-typ-pad-combine* (*t-b*::'*b*::*c-type itself*) *f-ab f-upd-ab algn fn tag*) (*f*#*fs*) *m* = (*if f*=*fn* *then field-lookup* (*adjust-ti* (*typ-info-t TYPE*('*b*)) *f-ab f-upd-ab*) *fs* (*padup* (*max* (2 ^ *algn*) (*align-of TYPE*('*b*))) (*size-td tag*) + *size-td tag*+*m*) *else field-lookup tag* (*f* # *fs*) *m*)
⟨*proof*⟩

lemma *fl-ti-typ-pad-combine-match*:
⟦ *fn* ∉ *set* (*CompoundCTypes.field-names-list tag*); *hd fn* ≠ *CHR* ''!''; *aggregate tag* ⟧ ⟹ *field-lookup* (*ti-typ-pad-combine* (*t-b*::'*b*::*c-type itself*) *f-ab f-upd-ab algn fn tag*) (*fn*#*fs*) *m* = *field-lookup* (*adjust-ti* (*typ-info-t TYPE*('*b*)) *f-ab f-upd-ab*) *fs* (*padup* (*max* (2 ^ *algn*) (*align-of TYPE*('*b*))) (*size-td tag*) + *size-td tag*+*m*)
⟨*proof*⟩

lemma *fl-ti-typ-pad-combine-mismatch*:
⟦ *hd f* ≠ *CHR* ''!''; *aggregate tag*; *f* ≠ *fn* ⟧ ⟹ *field-lookup* (*ti-typ-pad-combine* (*t-b*::'*b*::*c-type itself*) *f-ab f-upd-ab algn fn tag*) (*f*#*fs*) *m* = *field-lookup tag* (*f* # *fs*) *m*
⟨*proof*⟩

lemma *field-lookup-map-align-cons*: *field-lookup* (*map-align g t*) (*f*#*fs*) *m* = *field-lookup t* (*f*#*fs*) *m*
⟨*proof*⟩

lemma *fl-final-pad*:
⟦ *hd f* ≠ *CHR* ''!''; *aggregate tag* ⟧ ⟹ *field-lookup* (*final-pad algn tag*) (*f*#*fs*) *m* = *field-lookup tag* (*f*#*fs*) *m*
⟨*proof*⟩

lemma *field-lookup-adjust-ti2'* [*rule-format*]:
∀ *fn m s n*. *field-lookup ti fn m* = *Some* (*s*,*n*) ⟶ (*field-lookup* (*adjust-ti ti f g*) *fn m* = *Some* (*adjust-ti s f g*,*n*))
∀ *fn m s n*. *field-lookup-struct st fn m* = *Some* (*s*,*n*) ⟶ *field-lookup-struct* (*map-td-struct* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*) *st*) *fn m* = *Some* (*adjust-ti s f g*,*n*)
∀ *fn m s n*. *field-lookup-list ts fn m* = *Some* (*s*,*n*) ⟶ *field-lookup-list* (*map-td-list* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*)

ts) *fn m* = *Some* (*adjust-ti s f g*,*n*)
∀ *fn m s n*. *field-lookup-tuple x fn m* = *Some* (*s*,*n*) ⟶
field-lookup-tuple (*map-td-tuple* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*) *x*) *fn m* = *Some* (*adjust-ti s f g*,*n*)
⟨*proof*⟩

lemma *field-lookup-adjust-ti2*:
field-lookup t fn m = *Some* (*s*,*n*) ⟹
field-lookup (*adjust-ti t f g*) *fn m* = *Some* (*adjust-ti s f g*,*n*)
⟨*proof*⟩

lemma *fl-update*:
field-lookup (*adjust-ti ti f g*) *fs m* =
(*case-option None* (λ(*t*,*n*). *Some* (*adjust-ti t f g*,*n*)) (*field-lookup ti fs m*))
⟨*proof*⟩

lemmas *fl-simps* = *fl-final-pad fl-ti-pad-combine*
fl-ti-typ-combine-match fl-ti-typ-combine-mismatch
fl-ti-typ-pad-combine-match fl-ti-typ-pad-combine-mismatch

lemma *access-ti-props-simps* [*simp*]:
∀ *g x*. *access-ti* (*adjust-ti* (*tag*::′*a xtyp-info*) (*f*::′*b* ⇒ ′*a*) *g*) *x* = *access-ti tag* (*f x*)
∀ *g x*. *access-ti-struct* (*map-td-struct* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*) (*st*::′*a xtyp-info-struct*)) *x* = *access-ti-struct st* (*f x*)
∀ *g x*. *access-ti-list* (*map-td-list* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*) (*ts*::′*a xtyp-info-tuple list*)) *x* = *access-ti-list ts* (*f x*)
∀ *g x*. *access-ti-tuple* (*map-td-tuple* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*) (*k*::′*a xtyp-info-tuple*)) *x* = *access-ti-tuple k* (*f x*)
⟨*proof*⟩

lemma *field-norm-blah*:
⟦ ∀ *u v*. *f* (*g u v*) = *u*; *fd-cons-access-update d n* ⟧ ⟹
field-norm n algn (*update-desc f g d*) = *field-norm n algn d*
⟨*proof*⟩

lemma *map-td-ext′*:
wf-fd t ∧ (∀ *n algn d*. *fd-cons-access-update d n* ⟶ (*f n algn d* = *g n algn d*)) ⟶ *map-td f h t* = *map-td g h t*
wf-fd-struct st ∧ (∀ *n algn d*. *fd-cons-access-update d n* ⟶ (*f n algn d* = *g n algn d*)) ⟶ *map-td-struct f h st* = *map-td-struct g h st*
wf-fd-list ts ∧ (∀ *n algn d*. *fd-cons-access-update d n* ⟶ (*f n algn d* = *g n algn d*)) ⟶ *map-td-list f h ts* = *map-td-list g h ts*
wf-fd-tuple x ∧ (∀ *n algn d*. *fd-cons-access-update d n* ⟶ (*f n algn d* = *g n algn d*)) ⟶ *map-td-tuple f h x* = *map-td-tuple g h x*
⟨*proof*⟩

lemma *map-td-extI*:

⟦ *wf-fd t*; (∀ *n algn d*. *fd-cons-access-update d n* ⟶ (*f n algn d* = *g n algn d*)) ⟧
⟹ *map-td f h t* = *map-td g h t*
⟨*proof*⟩

lemma *comp-unit*: (λ-. ()) ∘ *f* = (λ-. ())
⟨*proof*⟩

lemma *export-tag-adjust-ti2*:
⟦ ∀ *u v*. *f* (*g u v*) = *u*; *wf-lf* (*lf-set t* []); *wf-desc t* ⟧ ⟹
export-uinfo (*adjust-ti t f g*) = (*export-uinfo t*)
⟨*proof*⟩

lemma *field-names-list*:
field-names-list (*xs*@*ys*) *t* = *field-names-list xs t* @ *field-names-list ys t*
⟨*proof*⟩

lemma *field-names-extend-ti*:
typ-name t ≠ *typ-name ti* ⟹
field-names (*extend-ti ti xi algn fn d*) *t* = *field-names ti t* @ (*map* (λ*fs*. *fn*#*fs*)
(*field-names xi t*))
⟨*proof*⟩

lemma *field-names-ti-pad-combine*:
⟦ *typ-name t* ≠ *typ-name ti*; *hd* (*typ-name t*) ≠ *CHR* ''!'' ⟧ ⟹
field-names (*ti-pad-combine n ti*) *t* = *field-names ti t*
⟨*proof*⟩

lemma *export-uinfo-map-align-commute*: *export-uinfo* (*map-align f t*) = *map-align*
f (*export-uinfo t*)
⟨*proof*⟩

lemma *field-names-map-align*: *typ-name t* ≠ *typ-name ti* ⟹ *field-names* (*map-align*
f ti) *t* = *field-names ti t*
⟨*proof*⟩

lemma *typ-name-map-align* [*simp*]: *typ-name* (*map-align f t*) = *typ-name t*
⟨*proof*⟩

lemma *typ-name-ti-pad-combine* [*simp*]:
typ-name (*ti-pad-combine n ti*) = *typ-name ti*
⟨*proof*⟩

lemma *field-names-final-pad*:
⟦ *typ-name t* ≠ *typ-name ti*; *hd* (*typ-name t*) ≠ *CHR* ''!'' ⟧ ⟹
field-names (*final-pad a ti*) *t* = *field-names ti t*
⟨*proof*⟩

lemma *field-names-adjust-ti*:

assumes *fg-cons f g*
shows
wf-fd ti $\longrightarrow$
field-names (*adjust-ti* (*ti*::'*a xtyp-info*) *f g*) *t* = *field-names ti t*
wf-fd-struct st $\longrightarrow$
field-names-struct ((*map-td-struct* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*)
(*st*::'*a xtyp-info-struct*))) *t* =
field-names-struct st t
wf-fd-list ts $\longrightarrow$
field-names-list (*map-td-list* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*)
(*ts*::'*a xtyp-info-tuple list*)) *t* =
field-names-list ts t
wf-fd-tuple x $\longrightarrow$
field-names-tuple (*map-td-tuple* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*)
(*x*::'*a xtyp-info-tuple*)) *t* =
field-names-tuple x t ⟨*proof*⟩

lemma *field-names-ti-typ-combine*:
⟦ *typ-name t* ≠ *typ-name ti*; *fg-cons f g* ⟧ ⟹
field-names (*ti-typ-combine* (*t-b*::'*b*::*mem-type itself*) *f g algn fn ti*) *t* =
field-names ti t @ *map* ((#) *fn*) (*field-names* (*typ-info-t TYPE*('*b*)) *t*)
⟨*proof*⟩

lemma *size-empty-typ-info* [*simp*]:
size (*empty-typ-info algn tn*) = *2*
⟨*proof*⟩

lemma *list-size-char*:
size-list (λ*c*. *0*) *xs* = *length xs*
⟨*proof*⟩

lemma *size-ti-extend-ti* [*simp*]:
aggregate ti ⟹ *size* (*extend-ti ti t algn fn d*) = *size ti* + *size t* + *2*
⟨*proof*⟩

lemma *typ-name-empty-typ-info* [*simp*]:
typ-name (*empty-typ-info algn tn*) = *tn*
⟨*proof*⟩

lemma *typ-name-extend-ti* [*simp*]:
typ-name (*extend-ti ti t algn fn d*) = *typ-name ti*
⟨*proof*⟩

lemma *typ-name-ti-typ-combine* [*simp*]:
typ-name (*ti-typ-combine* (*t-b*::'*b*::*c-type itself*) *f g algn fn ti*) = *typ-name ti*
⟨*proof*⟩

lemma *typ-name-ti-typ-pad-combine* [*simp*]:
typ-name (*ti-typ-pad-combine* (*t-b*::'*b*::*c-type itself*) *f g algn fn ti*) = *typ-name ti*
⟨*proof*⟩

lemma *typ-name-ti-final-pad* [*simp*]:
typ-name (*final-pad algn ti*) = *typ-name ti*
⟨*proof*⟩

lemma *typ-name-map-td* [*simp*]:
typ-name (*map-td f g td*) = *typ-name td*
⟨*proof*⟩

lemma *field-names-ti-typ-pad-combine*:
⟦ *typ-name t* ≠ *typ-name ti*; *fg-cons f g*; *aggregate ti*; *hd* (*typ-name t*) ≠ *CHR* ''!'' ⟧ ⟹
field-names (*ti-typ-pad-combine* (*t-b*::'*b*::*mem-type itself*) *f g algn fn ti*) *t* =
field-names ti t @ *map* ((#) *fn*) (*field-names* (*typ-info-t TYPE*('*b*)) *t*)
⟨*proof*⟩

lemma *field-names-empty-typ-info*:
typ-name t ≠ *tn* ⟹ *field-names* (*empty-typ-info algn tn*) *t* = []
⟨*proof*⟩

lemma *sep-heap-update-global-super-fl'*:
⟦ (*p* ↦$_g$ *u* ∧* *R*) (*lift-state* (*h*,*d*));
field-lookup (*typ-info-t TYPE*('*b*::*mem-type*)) *f 0* = *Some* (*t*,*n*);
export-uinfo t = (*typ-uinfo-t TYPE*('*a*));
w = *update-ti-t t* (*to-bytes-p v*) *u* ⟧ ⟹
((*p* ↦$_g$ *w*) ∧* *R*)
(*lift-state* (*heap-update* (*Ptr* &(*p*→*f*)) (*v*::'*a*::*mem-type*) *h*,*d*))
⟨*proof*⟩

lemma *sep-heap-update-global-super-fl'-inv*:
⟦ (*p* ↦$^i{}_g$ *u* ∧* *R*) (*lift-state* (*h*,*d*));
field-lookup (*typ-info-t TYPE*('*b*::*mem-type*)) *f 0* = *Some* (*t*,*n*);
export-uinfo t = (*typ-uinfo-t TYPE*('*a*));
w = *update-ti-t t* (*to-bytes-p v*) *u*⟧ ⟹
((*p* ↦$^i{}_g$ *w*) ∧* *R*) (*lift-state* (*heap-update* (*Ptr* &(*p*→*f*)) (*v*::'*a*::*mem-type*) *h*,*d*))
⟨*proof*⟩

lemma *sep-map'-field-map'*:
⟦ ((*p*::'*b*::*mem-type ptr*) ↪$_g$ *v*) *s*; *field-lookup* (*typ-info-t TYPE*('*b*)) *f 0*
= *Some* (*d*,*n*); *export-uinfo d* = *typ-uinfo-t TYPE*('*a*);
guard-mono g h ⟧ ⟹
((*Ptr* (&(*p*→*f*))::'*a*::*mem-type ptr*) ↪$_h$ *from-bytes* (*access-ti*$_0$ *d v*)) *s*
⟨*proof*⟩

lemma *sep-map'-field-map*:
⟦ ((*p*::'*b*::*mem-type ptr*) ↪$_g$ *v*) *s*; *field-lookup* (*typ-info-t TYPE*('*b*)) *f 0*

$= Some\ (d,n);\ export\text{-}uinfo\ d = typ\text{-}uinfo\text{-}t\ TYPE('a);$
$guard\text{-}mono\ g\ h;\ w{=}from\text{-}bytes\ (access\text{-}ti_0\ d\ v)\ ⟧ ⟹$
$((Ptr\ (\&(p{\rightarrow}f))::'a::mem\text{-}type\ ptr) \hookrightarrow_h w)\ s$
⟨*proof*⟩

lemma *inter-sub*:
$⟦\ Y \subseteq X;\ Y = Z\ ⟧ ⟹ X \cap Y = Z$
⟨*proof*⟩

lemma *sep-map'-field-map-inv*:
$⟦\ ((p::'b::mem\text{-}type\ ptr) \hookrightarrow^i_g v)\ s;\ field\text{-}lookup\ (typ\text{-}info\text{-}t\ TYPE('b))\ f\ 0 = Some\ (d,n);$
$export\text{-}uinfo\ d = typ\text{-}uinfo\text{-}t\ TYPE('a);\ guard\text{-}mono\ g\ h;\ w{=}from\text{-}bytes\ (access\text{-}ti_0\ d\ v)\ ⟧ ⟹$
$((Ptr\ \&(p{\rightarrow}f)::'a::mem\text{-}type\ ptr) \hookrightarrow^i_h w)\ s$
⟨*proof*⟩

lemma *guard-mono-True* [*simp*]:
$guard\text{-}mono\ f\ (\lambda x.\ True)$
⟨*proof*⟩

lemma *access-ti$_0$-to-bytes* [*simp*]:
$access\text{-}ti_0\ (typ\text{-}info\text{-}t\ TYPE('a::c\text{-}type)) = (to\text{-}bytes\text{-}p::'a \Rightarrow byte\ list)$
⟨*proof*⟩

lemma *update-ti-s-from-bytes*:
$length\ bs = size\text{-}of\ TYPE('a) ⟹$
$update\text{-}ti\text{-}t\ (typ\text{-}info\text{-}t\ TYPE('a::mem\text{-}type))\ bs\ x = from\text{-}bytes\ bs$
⟨*proof*⟩

lemma *access-ti$_0$-update-ti* [*simp*]:
$access\text{-}ti_0\ (adjust\text{-}ti\ ti\ f\ g) = access\text{-}ti_0\ ti \circ f$
⟨*proof*⟩

lemma *update-ti-s-adjust-ti*:
$⟦\ length\ bs = size\text{-}td\ ti;\ fg\text{-}cons\ f\ g\ ⟧ ⟹$
$update\text{-}ti\text{-}t\ (adjust\text{-}ti\ ti\ f\ g)\ bs\ v = g\ (update\text{-}ti\text{-}t\ ti\ bs\ (f\ v))\ v$
⟨*proof*⟩

lemma *update-ti-s-adjust-ti-to-bytes-p* [*simp*]:
$fg\text{-}cons\ f\ g ⟹$
$update\text{-}ti\text{-}t\ (adjust\text{-}ti\ (typ\text{-}info\text{-}t\ TYPE('a))\ f\ g)\ (to\text{-}bytes\text{-}p\ (v::'a::mem\text{-}type))\ w = g\ v\ w$
⟨*proof*⟩

definition
$pad\text{-}typ\text{-}name \equiv ''!pad\text{-}typ''$

primrec
td-names :: ('*a*,'*b*) *typ-desc* ⇒ (*char list*) *set* **and**
td-names-struct :: ('*a*,'*b*) *typ-struct* ⇒ (*char list*) *set* **and**
td-names-list :: ('*a*,'*b*) *typ-tuple list* ⇒ (*char list*) *set* **and**
td-names-tuple :: ('*a*,'*b*) *typ-tuple* ⇒ (*char list*) *set*
where
tnm0: *td-names* (*TypDesc algn st nm*) = ({*nm*} ∪ *td-names-struct st*) − {*pad-typ-name*}

| *tnm1*: *td-names-struct* (*TypScalar n algn d*) = {}
| *tnm2*: *td-names-struct* (*TypAggregate xs*) = *td-names-list xs*

| *tnm3*: *td-names-list* [] = {}
| *tnm4*: *td-names-list* (*x*#*xs*) = *td-names-tuple x* ∪ *td-names-list xs*

| *tnm5*: *td-names-tuple* (*DTuple t nm d*) = *td-names t*

lemma *td-set-td-names*:
⋀(*tp* :: ('*a*,'*b*) *typ-desc*) *n m*. ⟦(*tp*, *n*) ∈ *td-set tp*′ *m*; *typ-name tp* ≠ *pad-typ-name* ⟧ ⟹
typ-name tp ∈ *td-names tp*′ **and**
⋀(*tp* :: ('*a*,'*b*) *typ-desc*) *n m*. ⟦(*tp*, *n*) ∈ *td-set-struct tps m*; *typ-name tp* ≠ *pad-typ-name* ⟧ ⟹
typ-name tp ∈ *td-names-struct tps* **and**
⋀(*tp* :: ('*a*,'*b*) *typ-desc*) *n m*. ⟦(*tp*, *n*) ∈ *td-set-list tpl m*; *typ-name tp* ≠ *pad-typ-name* ⟧ ⟹
typ-name tp ∈ *td-names-list tpl* **and**
⋀(*tp* :: ('*a*,'*b*) *typ-desc*) *n m*. ⟦(*tp*, *n*) ∈ *td-set-tuple tpr m*; *typ-name tp* ≠ *pad-typ-name* ⟧⟹
typ-name tp ∈ *td-names-tuple tpr*
⟨*proof*⟩

lemma *td-names-map-td* [*simp*]:
td-names (*map-td f g tp*) = *td-names tp*
td-names-struct (*map-td-struct f g tps*) = *td-names-struct tps*
td-names-list (*map-td-list f g tpl*) = *td-names-list tpl*
td-names-tuple (*map-td-tuple f g tpr*) = *td-names-tuple tpr*
⟨*proof*⟩

lemma *td-names-list-append* [*simp*]:
td-names-list (*a* @ *b*) = *td-names-list a* ∪ *td-names-list b*
⟨*proof*⟩

lemma *pad-typ-name-td-names*:
A − {*pad-typ-name*} ∪ *td-names tp* = (*A* ∪ *td-names tp*) − {*pad-typ-name*}
⟨*proof*⟩

lemma *td-names-extend-ti* [*simp*]:
shows *td-names* (*extend-ti tp tp*′ *algn ls d*) = *td-names tp* ∪ *td-names tp*′ **and**

td-names-struct (*extend-ti-struct tps tp′ ls d*) = *td-names-struct tps* ∪ *td-names tp′*
⟨*proof*⟩

lemma *td-names-pad-combine* [*simp*]:
td-names (*ti-pad-combine m td*) = *td-names td*
⟨*proof*⟩

lemma *td-names-map-align* [*simp*]:
td-names (*map-align f td*) = *td-names td*
⟨*proof*⟩

lemma *td-names-final-pad* [*simp*]:
td-names (*final-pad algn td*) = *td-names td*
⟨*proof*⟩

lemma *td-names-adjust-ti* [*simp*]:
td-names (*adjust-ti td f u*) = *td-names td*
⟨*proof*⟩

lemma *td-names-typ-combine* [*simp*]:
fixes *its* :: (*'a* :: *c-type*) *itself*
shows *td-names* (*ti-typ-combine its f u algn fn td*) = (*td-names* (*typ-info-t TYPE*(*'a*)) ∪ *td-names td*)
⟨*proof*⟩

lemma *td-names-typ-pad-combine* [*simp*]:
fixes *its* :: (*'a* :: *c-type*) *itself*
shows *td-names* (*ti-typ-pad-combine its f u algn nm td*) = *td-names* (*typ-info-t TYPE*(*'a*)) ∪ *td-names td*
⟨*proof*⟩

lemma *td-names-empty-typ-info* [*simp*]:
shows *td-names* (*empty-typ-info algn nm*) = {*nm*} − {*pad-typ-name*}
⟨*proof*⟩

lemma *td-names-ptr* [*simp*]:
td-names (*typ-info-t TYPE*((*'a* :: *c-type*) *ptr*)) = {*typ-name-itself TYPE*(*'a*) @ *''+ptr''*}
⟨*proof*⟩

lemma *td-names-word8* [*simp*]:
fixes *x* :: *byte itself*
shows *td-names* (*typ-info-t x*) = {*''word00010''*}
⟨*proof*⟩

lemma *td-names-word8′* [*simp*]:
shows *td-names* (*typ-info-t TYPE*(*8 word*)) = {*''word00010''*}

⟨*proof*⟩

lemma *td-names-signed-word8* [*simp*]:
shows *td-names* (*typ-info-t TYPE*(*8 signed word*)) = {*''word00010''*}
⟨*proof*⟩

lemma *td-names-word16* [*simp*]:
shows *td-names* (*typ-info-t TYPE*(*16 word*)) = {*''word000010''*}
⟨*proof*⟩

lemma *td-names-signed-word16* [*simp*]:
shows *td-names* (*typ-info-t TYPE*(*16 signed word*)) = {*''word000010''*}
⟨*proof*⟩

lemma *td-names-word32* [*simp*]:
td-names (*typ-info-t TYPE*(*32 word*)) = {*''word0000010''*}
⟨*proof*⟩

lemma *td-names-signed-word32* [*simp*]:
td-names (*typ-info-t TYPE*(*32 signed word*)) = {*''word0000010''*}
⟨*proof*⟩

lemma *td-names-word64* [*simp*]:
td-names (*typ-info-t TYPE*(*64 word*)) = {*''word00000010''*}
⟨*proof*⟩

lemma *td-names-signed-word64* [*simp*]:
td-names (*typ-info-t TYPE*(*64 signed word*)) = {*''word00000010''*}
⟨*proof*⟩

lemma *td-names-export-uinfo* [*simp*]:
td-names (*export-uinfo td*) = *td-names td*
⟨*proof*⟩

lemma *typ-name-export-uinfo* [*simp*]:
typ-name (*export-uinfo td*) = *typ-name td*
⟨*proof*⟩

lemma *replicate-Suc-append*:
replicate (*Suc n*) *x* = *replicate n x* @ [*x*]
⟨*proof*⟩

lemma *list-eq-subset*:
xs = *ys* ⟹ *set ys* ⊆ *set xs* ⟨*proof*⟩

lemma *td-names-array-tag-n*:
td-names ((*array-tag-n n*) :: (('*a*::*c-type*,'*b*::*finite*) *array xtyp-info*)) =
{*typ-name* (*typ-info-t TYPE*('*a*)) @ *''-array-''* @ *nat-to-bin-string* (*card* (*UNIV*

:: 'b set))} ∪
    (if n = 0 then {} else td-names (typ-info-t TYPE('a)))
  ⟨proof⟩

lemma *td-names-array* [*simp*]:
  td-names (*typ-info-t TYPE*(('a :: *c-type*)['b :: *finite*])) =
    {*typ-name* (*typ-info-t TYPE*('a)) @ ''-array-'' @ *nat-to-bin-string* (*card* (*UNIV* :: 'b *set*))} ∪
    td-names (*typ-info-t TYPE*('a))
  ⟨*proof*⟩

lemma *tag-disj-via-td-name*:
  assumes *ta*: *typ-name* (*typ-info-t TYPE*('a :: *c-type*)) ≠ *pad-typ-name*
  and *tb*: *typ-name* (*typ-info-t TYPE*('b :: *c-type*)) ≠ *pad-typ-name*
  and *tina*: *typ-name* (*typ-info-t TYPE*('a :: *c-type*)) ∉ *td-names* (*typ-info-t TYPE*('b :: *c-type*))
  and *tinb*: *typ-name* (*typ-info-t TYPE*('b :: *c-type*)) ∉ *td-names* (*typ-info-t TYPE*('a :: *c-type*))
  shows *typ-uinfo-t TYPE*('a :: *c-type*) ⊥$_t$ *typ-uinfo-t TYPE*('b :: *c-type*)
  ⟨*proof*⟩

lemma *lift-t-hrs-mem-update-fld*:
  fixes *val* :: 'b :: *mem-type* **and** *ptr* :: 'a :: *mem-type ptr*
  assumes *fl*: *field-lookup* (*typ-info-t TYPE*('a)) *f* 0 ≡
      Some (*adjust-ti* (*typ-info-t TYPE*('b)) *xf* (*xfu* ∘ (λx -. x)), *m'*)
  and *xf-xfu*: *fg-cons xf* (*xfu* ∘ (λx -. x))
  and *cl*: *lift-t g hp ptr* = *Some z*
  shows (*lift-t g* (*hrs-mem-update* (*heap-update* (*Ptr* &(*ptr*→*f*)) *val*) *hp*)) =
      (*lift-t g hp*)(*ptr* ↦ *xfu* (λ-. *val*) *z*)
  (**is** *?LHS* = *?RHS*)
⟨*proof*⟩

declare *pad-typ-name-def* [*simp*]

lemma *typ-name-array-tag-n*:
  typ-name (*array-tag-n n* :: ('a ::*c-type* ['b :: *finite*]) *xtyp-info*) =
    typ-name (*typ-info-t TYPE*('a)) @ ''-array-'' @ *nat-to-bin-string* (*card* (*UNIV* :: 'b *set*))
  ⟨*proof*⟩

lemma *typ-name-array* [*simp*]:
  typ-name (*typ-info-t TYPE*('a::*c-type*['b :: *finite*])) =
    typ-name (*typ-info-t TYPE*('a)) @ ''-array-'' @ *nat-to-bin-string* (*card* (*UNIV* :: 'b *set*))
  ⟨*proof*⟩

definition
  size-align-td :: ('a,'b) *typ-desc* ⇒ *nat* ⇒ *nat* ⇒ *bool*

where
size-align-td t s a ≡ *size-td t* = *s* ∧ *align-td t* = *a*

lemma *size-align-td-size-td*:
size-align-td t s a ⟹ *size-td t* = *s*
⟨*proof*⟩

lemma *size-align-td-align-td*:
size-align-td t s a ⟹ *align-td t* = *a*
⟨*proof*⟩

lemma *size-align-td-empty-typ-infoI*:
size-align-td (*empty-typ-info algn l*) *0 algn*
⟨*proof*⟩

lemma *size-align-td-ti-typ-pad-combineI*:
fixes *t* :: ('*a* :: *c-type*) *itself*
shows ⟦ *size-align-td t' s' a'*; *aggregate t'*
; *s'* + *size-td* (*typ-info-t TYPE*('*a*)) + *padup* (*2* ^ (*max algn* (*align-td* (*typ-info-t TYPE*('*a*))))) *s'* = *s*
; *max a'* (*max algn* (*align-td* (*typ-info-t TYPE*('*a*)))) = *a*
⟧
⟹ *size-align-td* (*ti-typ-pad-combine t fa fu algn nm t'*) *s a*
⟨*proof*⟩

lemma *size-align-td-ti-final-padI*:
fixes *t* :: ('*a* :: *c-type*) *xtyp-info*
shows ⟦ *size-align-td t' s' a'*; *aggregate t'*; *s'* + *padup* (*2* ^ *max algn a'*) *s'* = *s*; *max algn a'* = *a* ⟧
⟹ *size-align-td* (*final-pad algn t'*) *s a*
⟨*proof*⟩

lemma *field-lookup-empty-typ-infoI*:
field-lookup (*empty-typ-info algn l*) [*fld*] *m* = *None*
⟨*proof*⟩

lemma *field-lookup-ti-typ-pad-combine-matchI*:
⟦ *n* # *nm* ∉ *set* (*CompoundCTypes.field-names-list t'*); *aggregate t'*; *size-align-td t' s a*
; *padup* (*max* (*2* ^ *algn*) (*align-of TYPE*('*b*))) *s* + *s* = *m'*; *n* ≠ *CHR* ''!'' ⟧
⟹ *field-lookup* (*ti-typ-pad-combine* (*t* :: '*b itself*) *fa fu algn* (*n*#*nm*) *t'*) [(*n*#*nm*)] *0* = *Some* (*adjust-ti* (*typ-info-t TYPE*('*b* :: *c-type*)) *fa fu*, *m'*)
⟨*proof*⟩

lemma *field-lookup-ti-typ-pad-combine-nomatchI*:
⟦ *aggregate t'*; *f* ≠ *CHR* ''!''; *f* # *fld* ≠ *nm* ; *field-lookup t'* [*f* # *fld*] *0* = *R* ⟧
⟹ *field-lookup* (*ti-typ-pad-combine t fa fu algn nm t'*) [*f* # *fld*] *0* = *R*
⟨*proof*⟩

lemma *field-lookup-final-padI*:
⟦ *aggregate t'*; *f* ≠ *CHR ''!''*; *field-lookup t' [f # fld] 0 = R* ⟧ ⟹ *field-lookup (final-pad align t') [f # fld] 0 = R*
⟨*proof*⟩

lemmas *field-lookup-ti-intros* =
field-lookup-ti-typ-pad-combine-matchI field-lookup-ti-typ-pad-combine-nomatchI field-lookup-final-padI field-lookup-empty-typ-infoI

lemma *notin-field-names-list-empty-typ-info*:
fld ∉ *set* (*CompoundCTypes.field-names-list* (*empty-typ-info align l*))
⟨*proof*⟩

lemma *notin-field-names-list-ti-typ-pad-combine*:
⟦ *f#fld* ∉ *set* (*CompoundCTypes.field-names-list t'*); *f#fld* ≠ *nm*; *f* ≠ *CHR ''!''* ⟧
⟹ *f#fld* ∉ *set* (*CompoundCTypes.field-names-list* (*ti-typ-pad-combine t fa fu align nm t'*))
⟨*proof*⟩

lemma *fold-typ-uinfo-t*: *export-uinfo* (*typ-info-t TYPE('a::c-type)*) = *typ-uinfo-t TYPE('a)*
⟨*proof*⟩

end

theory *ArrayAssertion*

imports
ArchArraysMemInstance
StructSupport

begin

lemma *array-tag-n-eq*:
(*array-tag-n n* :: ('*a* :: *c-type*['*b* :: *finite*]) *xtyp-info*) =
TypDesc (*align-td* (*typ-uinfo-t TYPE('a)*)) (*TypAggregate*
(*map* (λ*n*. *DTuple* (*adjust-ti* (*typ-info-t TYPE('a)*) (λ*x*. *index x n*)
(λ*x f*. *Arrays.update f n x*)) (*replicate n CHR ''1''*)
(|*field-access* = *xto-bytes* ∘ (λ*x*. *index x n*),
field-update = (λ*x f*. *Arrays.update f n x*) ∘ *xfrom-bytes*,
field-sz = *size-of TYPE('a::c-type)*|)) [*0..<n*]))
(*typ-name* (*typ-uinfo-t TYPE('a)*) @ *''-array-''* @ *nat-to-bin-string* (*card* (*UNIV* :: '*b* :: *finite set*)))

⟨*proof*⟩

lemma *typ-info-array'*:
typ-info-t TYPE (*'a* :: *c-type*[*'b* :: *finite*]) =
TypDesc (*align-td* (*typ-uinfo-t TYPE*(*'a*))) (*TypAggregate*
(*map* (λ*n*. *DTuple* (*adjust-ti* (*typ-info-t TYPE*(*'a*)) (λ*x*. *index x n*)
(λ*x f*. *Arrays.update f n x*)) (*replicate n CHR ''1''*)
(|*field-access* = *xto-bytes* ∘ (λ*x*. *index x n*),
field-update = (λ*x f*. *Arrays.update f n x*) ∘ *xfrom-bytes*,
field-sz = *size-of TYPE*(*'a*::*c-type*)|)) [*0*..<(*card* (*UNIV* :: *'b* :: *finite set*))]))
(*typ-name* (*typ-uinfo-t TYPE*(*'a*)) @ *''-array-''* @ *nat-to-bin-string* (*card* (*UNIV* :: *'b* :: *finite set*)))
⟨*proof*⟩

definition
uinfo-array-tag-n-m (*v* :: *'a itself*) *n m* = *TypDesc*
(*align-td* (*typ-uinfo-t TYPE*(*'a*)))
(*TypAggregate* (*map* (λ*i*. *DTuple* (*typ-uinfo-t TYPE*(*'a*)) (*replicate i CHR ''1''*) ()) [*0* ..< *n*]))
(*typ-name* (*typ-uinfo-t TYPE*(*'a* :: *c-type*)) @ *''-array-''* @ *nat-to-bin-string m*)

lemma *map-td-list-map*:
map-td-list f g = *map* (*map-td-tuple f g*)
⟨*proof*⟩

lemma *uinfo-array-tag-n-m-eq*:
n ≤ *CARD*(*'b*)
⟹ *export-uinfo* (*array-tag-n n* :: ((*'a* :: *wf-type*)[*'b* :: *finite*]) *xtyp-info*)
= *uinfo-array-tag-n-m TYPE* (*'a*) *n* (*CARD*(*'b*))
⟨*proof*⟩

lemma *typ-uinfo-array-tag-n-m-eq*:
typ-uinfo-t TYPE ((*'a* :: *wf-type*)[*'b* :: *finite*])
= *uinfo-array-tag-n-m TYPE* (*'a*) (*CARD*(*'b*)) (*CARD*(*'b*))
⟨*proof*⟩

Alternative to *h-t-valid* for arrays. This allows reasoning about arrays of variable width.

definition *h-t-array-valid* :: *heap-typ-desc* ⇒ (*'a* :: *c-type*) *ptr* ⇒ *nat* ⇒ *bool* **where**
h-t-array-valid htd ptr n = *valid-footprint htd* (*ptr-val ptr*) (*uinfo-array-tag-n-m TYPE* (*'a*) *n n*)

Assertion that pointer p is within an array that continues for at least n more elements.

definition
array-assertion (*p* :: (*'a* :: *c-type*) *ptr*) *n htd*

$= (\exists\, q\ i\ j.\ h\text{-}t\text{-}array\text{-}valid\ htd\ q\ j$
$\quad \wedge\ p = CTypesDefs.ptr\text{-}add\ q\ (int\ i) \wedge i < j \wedge i + n \leq j)$

lemma *array-assertion-shrink-right*:
array-assertion p n htd $\Longrightarrow n' \leq n \Longrightarrow$ *array-assertion p n' htd*
⟨*proof*⟩

lemma *array-assertion-shrink-leftD*:
array-assertion p n htd $\Longrightarrow j < n \Longrightarrow$ *array-assertion* (*CTypesDefs.ptr-add p* (*int j*)) $(n - j)$ *htd*
⟨*proof*⟩

lemma *array-assertion-shrink-leftI*:
array-assertion (*CTypesDefs.ptr-add p* $(-$ (*int j*))) $(n + j)$ *htd*
$\Longrightarrow n \neq 0 \Longrightarrow$ *array-assertion p n htd*
⟨*proof*⟩

lemma *h-t-array-valid*:
h-t-valid htd gd $(p ::$ (($'a ::$ *wf-type*)[$'b ::$ *finite*]) *ptr*)
$\Longrightarrow$ *h-t-array-valid htd* (*ptr-coerce p* $::$ $'a$ *ptr*) (*CARD*($'b$))
⟨*proof*⟩

lemma *array-ptr-valid-array-assertionD*:
h-t-valid htd gd $(p ::$ (($'a ::$ *wf-type*)[$'b ::$ *finite*]) *ptr*)
$\Longrightarrow$ *array-assertion* (*ptr-coerce p* $::$ $'a$ *ptr*) (*CARD*($'b$)) *htd*
⟨*proof*⟩

lemma *array-ptr-valid-array-assertionI*:
h-t-valid htd gd $(q ::$ (($'a ::$ *wf-type*)[$'b ::$ *finite*]) *ptr*)
$\Longrightarrow q =$ *ptr-coerce p*
$\Longrightarrow n \leq$ *CARD*($'b$)
$\Longrightarrow$ *array-assertion* $(p ::$ $'a$ *ptr*) *n htd*
⟨*proof*⟩

Derived from *array-assertion*, an appropriate assertion for performing a pointer addition, or for dereferencing a pointer addition (the strong case).

In either case, there must be an array connecting the two ends of the pointer transition, with the caveat that the last address can be just out of the array if the pointer is not dereferenced, thus the strong/weak distinction.

If the pointer doesn't actually move, nothing is learned.

definition *ptr-add-assertion* $::$ ($'a ::$ *c-type*) *ptr* $\Rightarrow$ *int* $\Rightarrow$ *bool* $\Rightarrow$ *heap-typ-desc* $\Rightarrow$ *bool* **where**
ptr-add-assertion ptr offs strong htd $\equiv$
offs $= 0 \vee$
array-assertion (*if offs* < 0 *then CTypesDefs.ptr-add ptr offs else ptr*)
(*if offs* < 0 *then nat* $(-$ *offs*) *else if strong then Suc* (*nat offs*) *else nat offs*)
htd

lemma *ptr-add-assertion-positive*:
offs $\geq$ *0* $\Longrightarrow$ *ptr-add-assertion ptr offs strong htd*
= (*offs* = *0* $\vee$ *array-assertion ptr* (*case strong of True* $\Rightarrow$ *Suc* (*nat offs*)
| *False* $\Rightarrow$ *nat offs*) *htd*)
⟨*proof*⟩

lemma *ptr-add-assertion-negative*:
offs < *0* $\Longrightarrow$ *ptr-add-assertion ptr offs strong htd*
= *array-assertion* (*CTypesDefs.ptr-add ptr offs*) (*nat* (− *offs*)) *htd*
⟨*proof*⟩

lemma *ptr-add-assertion-uint*[*simp*]:
ptr-add-assertion ptr (*uint offs*) *strong htd*
= (*offs* = *0* $\vee$ *array-assertion ptr*
(*case strong of True* $\Rightarrow$ *Suc* (*unat offs*) | *False* $\Rightarrow$ *unat offs*) *htd*)
⟨*proof*⟩

Ignore char and void pointers. The C standard specifies that arithmetic on char and void pointers doesn't create any special checks.

definition *ptr-add-assertion'* :: (*'a* :: *c-type*) *ptr* $\Rightarrow$ *int* $\Rightarrow$ *bool* $\Rightarrow$ *heap-typ-desc* $\Rightarrow$ *bool* **where**
ptr-add-assertion' ptr offs strong htd =
(*typ-uinfo-t TYPE*(*'a*) = *typ-uinfo-t TYPE*(*word8*)
$\vee$ *typ-uinfo-t TYPE* (*'a*) = *typ-uinfo-t TYPE*(*unit*)
$\vee$ *ptr-add-assertion ptr offs strong htd*)

lemma *td-diff-from-typ-name*:
typ-name td $\neq$ *typ-name td'* $\Longrightarrow$ *td* $\neq$ *td'*
⟨*proof*⟩

lemma *typ-name-void*:
typ-name (*typ-uinfo-t TYPE*(*unit*)) = *''unit''*
⟨*proof*⟩

lemmas *ptr-add-assertion' = ptr-add-assertion'-def td-diff-from-typ-name typ-name-void*

Mechanism for retyping a range of memory to a non-constant array size.

definition *ptr-arr-retyps* :: *nat* $\Rightarrow$ (*'a* :: *c-type*) *ptr* $\Rightarrow$ *heap-typ-desc* $\Rightarrow$ *heap-typ-desc* **where**
ptr-arr-retyps n p $\equiv$
htd-update-list (*ptr-val p*)
(*map* (λ*i*. *list-map* (*typ-slice-t* (*uinfo-array-tag-n-m TYPE*(*'a*) *n n*) *i*))
[*0*..<*n* $*$ *size-of TYPE*(*'a*)])

lemma *ptr-arr-retyps-to-retyp*:
n = *CARD*(*'b* :: *finite*)
$\Longrightarrow$ *ptr-arr-retyps n* (*p* :: (*'c* :: *wf-type*) *ptr*) = *ptr-retyp* (*ptr-coerce p* :: (*'c*['*b*]) *ptr*)

⟨*proof*⟩

definition
array-ptr-index :: ((*'a* :: *c-type*)[*'b* :: *finite*]) *ptr* ⇒ *bool* ⇒ *nat* ⇒ *'a ptr*
where
array-ptr-index p coerce n = *CTypesDefs.ptr-add* (*ptr-coerce p*)
(*if coerce* ∧ *n* ≥ *CARD* (*'b*) *then 0 else of-nat n*)

lemmas *array-ptr-index-simps*
= *array-ptr-index-def*[**where** *coerce*=*False*, *simplified*]
array-ptr-index-def[**where** *coerce*=*True*, *simplified*]

lemma *field-lookup-list-Some-again*:
dt-snd (*xs* ! *i*) = *f*
⟹ *i* < *length xs*
⟹ *f* ∉ *dt-snd* ' *set* ((*take i xs*))
⟹ *field-lookup-list xs* [*f*] *n*
= *Some* (*dt-fst* (*xs* ! *i*), *n* + *sum-list* (*map* (*size-td o dt-fst*) (*take i xs*)))
⟨*proof*⟩

lemma *field-lookup-array*:
n < *CARD*(*'b*) ⟹ *field-lookup* (*typ-info-t TYPE*((*'a* :: *c-type*)[*'b* :: *finite*]))
[*replicate n* (*CHR ''1''*)] *i* = *Some* (*adjust-ti* (*typ-info-t TYPE*(*'a*))
(λ*x*. *x*.[*n*]) (λ*x f*. *Arrays.update f n x*), *i* + *n* ∗ *size-of TYPE* (*'a*))
⟨*proof*⟩

lemma *field-ti-array*:
n < *CARD*(*'b*) ⟹
field-ti (*TYPE*((*'a* :: *c-type*)[*'b* :: *finite*])) [*replicate n* (*CHR ''1''*)] =
Some (*adjust-ti* (*typ-info-t TYPE*(*'a*)) (λ*x*. *x*.[*n*]) (λ*x f*. *Arrays.update f n x*))
⟨*proof*⟩

lemma *fg-cons-array* [*simp*]:
n < *card* (*UNIV* :: *'b* :: *finite set*) ⟹
fg-cons (λ*x*. *index x n*) (λ*x f*. *Arrays.update* (*f* :: *'a*[*'b*]) *n x*)
⟨*proof*⟩

lemma *ptr-val-array-field-lvalue-0*:
fixes *p*:: ((*'a* :: *c-type*)[*'b* :: *finite*]) *ptr*
shows &(*p*→[[]]) = *ptr-val p*
⟨*proof*⟩

end

theory *CProof*
imports

umm-heap/SepFrame
Simpl.Vcg
umm-heap/StructSupport
umm-heap/ArrayAssertion
AutoCorres-Utils
begin

⟨*ML*⟩

syntax
-quote :: *'b => ('a => 'b)* (‹(‹*notation=‹mixfix quote›*›[.[-].])› [*0*] *1000*)

syntax
-heap :: *'b ⇒ ('a ⇒ 'b)*
-heap-state :: *'a* (‹*ζ*›)
-heap-stateOld :: *('a ⇒ 'b) ⇒ 'b* (‹(‹*open-block notation=‹mixfix heap-state old*›› ¯*ζ*)›
[*100*] *100*)

-derefCur :: *('a ⇒ 'b) ⇒ 'b* (‹(‹*open-block notation=‹prefix deref*›› ⋆-)› [*100*] *100*)
-derefOld :: *'a ⇒ ('a ⇒ 'b) ⇒ 'b* (‹(‹*open-block notation=‹prefix deref old*›› ¯⋆-)›
[*100*,*100*] *100*)

translations
{|*b*|} => *CONST Collect* (*-quote* (*-heap b*))

definition *sep-app* :: (*heap-state ⇒ bool*) ⇒ *heap-state ⇒ bool* **where**
sep-app P s ≡ *P s*

definition *hrs-id* :: *heap-raw-state ⇒ heap-raw-state* **where**
hrs-id ≡ *id*

declare *hrs-id-def* [*simp add*]

⟨*ML*⟩

lemma *c-null-guard*:
c-null-guard (*p*::*'a*::*mem-type ptr*) ⟹ *p* ≠ *NULL*
⟨*proof*⟩

lemma (**in** *mem-type*) *c-guard-no-wrap*:
fixes *p* :: *'a ptr*
assumes *cgrd*: *c-guard p*
shows *ptr-val p* ≤ *ptr-val p* + *of-nat* (*size-of TYPE*(*'a*) − *1*)
⟨*proof*⟩

lemma *word-le-unat-bound*:
fixes *a*::*'a* ::*len word*
assumes $a \leq a + b$
shows *unat a* + *unat b* < 2 ^ *LENGTH*(*'a*)
⟨*proof*⟩

lemma (**in** *mem-type*) *c-guard-no-wrap'*:
fixes *p* :: *'a ptr*
assumes *cgrd*: *c-guard p*
shows *unat* (*ptr-val p*) + *size-of TYPE*(*'a*) ≤ *addr-card*
⟨*proof*⟩

definition
c-fnptr-guard-def: *c-fnptr-guard* (*fnptr*::*unit ptr*) ≡ *ptr-val fnptr* ≠ *0*

lemma *c-fnptr-guard-NULL* [*simp*]: *c-fnptr-guard NULL* = *False*
⟨*proof*⟩

lemma *c-guardD*:
c-guard (*p*::*'a*::*mem-type ptr*) ⟹ *ptr-aligned p* ∧ *p* ≠ *NULL*
⟨*proof*⟩

lemma *c-guard-ptr-aligned*:
c-guard p ⟹ *ptr-aligned p*
⟨*proof*⟩

lemma *c-guard-NULL*:
c-guard (*p*::*'a*::*mem-type ptr*) ⟹ *p* ≠ *NULL*
⟨*proof*⟩

lemma *c-guard-NULL-simp* [*simp*]:
¬ *c-guard* (*NULL*::*'a*::*mem-type ptr*)
⟨*proof*⟩

lemma *c-guard-mono*:
guard-mono (*c-guard*::*'a*::*mem-type ptr-guard*) (*c-guard*::*'b*::*mem-type ptr-guard*)
⟨*proof*⟩

lemma *c-guard-NULL-fl*:
⟦ *c-guard* (*p*::*'a*::*mem-type ptr*); *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*t*,*n*);
export-uinfo t = *typ-uinfo-t TYPE*(*'b*::*mem-type*) ⟧
⟹ *0* < &(*p*→*f*)
⟨*proof*⟩

lemma *c-guard-ptr-aligned-fl*:
⟦ *c-guard* (*p*::*'a*::*mem-type ptr*); *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some*

(t,n);
export-uinfo t = typ-uinfo-t TYPE('b::mem-type) ⟧ ⟹
ptr-aligned ((Ptr &(p→f))::'b ptr)
⟨*proof*⟩

syntax
-sep-map :: 'a::c-type ptr ⇒ 'a ⇒ heap-assert
(‹(‹*open-block notation=‹infix sep-map›*›- ↦ -)› [*56*,*51*] *56*)
-sep-map-any :: 'a::c-type ptr ⇒ heap-assert
(‹(‹*open-block notation=‹mixfix sep-map-any›*›- ↦ −)› [*56*] *56*)
-sep-map' :: 'a::c-type ptr ⇒ 'a ⇒ heap-assert
(‹(‹*open-block notation=‹infix sep-map'›*›- ↪ -)› [*56*,*51*] *56*)
-sep-map'-any :: 'a::c-type ptr ⇒ heap-assert
(‹(‹*open-block notation=‹mixfix sep-map'›*›- ↪ −)› [*56*] *56*)
-tagd :: 'a::c-type ptr ⇒ heap-assert
(‹(‹*open-block notation=‹mixfix tagd›*›⊢$_s$ -)› [*99*] *100*)

translations
p ↦ v == p ↦^{i}(CONST c-guard) v
p ↦ − == p ↦^{i}(CONST c-guard) −
p ↪ v == p ↪^{i}(CONST c-guard) v
p ↪ − == p ↪^{i}(CONST c-guard) −
⊢$_s$ *p == CONST c-guard* ⊢$_s{}^i$ *p*

term *x ↦ y*
term *(x ↦ y ∧* y ↦ z) s*
term *(x ↦ y ∧* y ↦ z) ∧* x ↦ y*
term ⊢$_s$ *p*

lemma *sep-map-NULL* [*simp*]:
NULL ↦ (v::'a::mem-type) = sep-false
⟨*proof*⟩

lemma *sep-map'-NULL-simp* [*simp*]:
NULL ↪ (v::'a::mem-type) = sep-false
⟨*proof*⟩

lemma *sep-map'-ptr-aligned*:
(p ↪ v) s ⟹ ptr-aligned p
⟨*proof*⟩

lemma *sep-map'-NULL*:
(p ↪ (v::'a::mem-type)) s ⟹ p ≠ NULL
⟨*proof*⟩

lemma *tagd-sep-false* [*simp*]:

$\vdash_s$ (*NULL*::'*a*::*mem-type ptr*) = *sep-false*
⟨*proof*⟩

syntax (**output**)
-Deref :: '*b* ⇒ '*b* (‹(‹*open-block notation*=‹*prefix Deref*››∗-)› [*1000*] *1000*)
-AssignH :: '*b* => '*b* => ('*a*,'*p*,'*f*) *com*
(‹(‹*indent*=*2 notation*=‹*mixfix AssignH*››∗- :==/ -)› [*30*, *30*] *23*)

⟨*ML*⟩

syntax *-h-t-valid* :: '*a*::*c-type ptr* ⇒ *bool*
(‹(‹*open-block notation*=‹*infix h-t-valid*›› $\models_t$ -)› [*99*] *100*)

abbreviation *lift-t-c* :: *heap-mem* × *heap-typ-desc* ⇒ '*a*::*c-type typ-heap* **where**
lift-t-c s == *lift-t c-guard s*

syntax *-h-t-valid* :: *heap-typ-desc* ⇒ '*a*::*c-type ptr* ⇒ *bool*
(‹(‹*open-block notation*=‹*infix h-t-valid*›› - $\models_t$ -)› [*99*,*99*] *100*)
translations
d $\models_t$ *p* == *d*,*CONST c-guard* $\models_t$ *p*

lemma *h-t-valid-c-guard*:
d $\models_t$ *p* ⟹ *c-guard p*
⟨*proof*⟩

lemma *h-t-valid-NULL* [*simp*]:
¬ *d* $\models_t$ (*NULL*::'*a*::*mem-type ptr*)
⟨*proof*⟩

lemma *h-t-valid-ptr-aligned*:
d $\models_t$ *p* ⟹ *ptr-aligned p*
⟨*proof*⟩

lemma *lift-t-NULL*:
lift-t-c s (*NULL*::'*a*::*mem-type ptr*) = *None*
⟨*proof*⟩

lemma *lift-t-ptr-aligned* [*simp*]:
lift-t-c s p = *Some v* ⟹ *ptr-aligned p*
⟨*proof*⟩

lemmas *c-typ-rewrs* = *typ-rewrs h-t-valid-ptr-aligned lift-t-NULL*

datatype '*gx c-exntype* = *Break* | *Return* | *Continue* | *Goto string* | *Nonlocal* '*gx*

definition *is-local x* = (*x* = *Break* ∨ *x* = *Return* ∨ *x* = *Continue* ∨ (∃ *l*. *x* = *Goto*

l))

lemma *is-local-simps* [*simp*]:
is-local Break
is-local Return
is-local Continue
is-local (*Goto l*)
¬*is-local* (*Nonlocal x*)
⟨*proof*⟩

primrec *the-Nonlocal* **where**
the-Nonlocal (*Nonlocal x*) = *x*

datatype *strictc-errortype* =
Div-0
| *C-Guard*
| *MemorySafety*
| *ShiftError*
| *SideEffects*
| *ArrayBounds*
| *SignedArithmetic*
| *DontReach*
| *GhostStateError*
| *UnspecifiedSyntax*
| *OwnershipError*
| *UndefinedFunction*
| *AdditionalError string*
| *ImpossibleSpec*
| *AssumeError*
| *StackOverflow*

definition *unspecified-syntax-error* :: *string* ⇒ *bool* **where**
unspecified-syntax-error s = *False*

record (*'g*, *'l*, *'e*) *state* = (*'g*, *'l*) *StateSpace.state-locals* +
global-exn-var'-' :: *'e c-exntype*

lemmas *hrs-simps* = *hrs-mem-update-def hrs-mem-def hrs-htd-update-def hrs-htd-def*

lemma *sep-map'-lift-lift*:
fixes *pa* :: *'a* :: *mem-type ptr* **and** *xf* :: *'a* ⇒ *'b* :: *mem-type*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* ≡
Some (*adjust-ti* (*typ-info-t TYPE*(*'b*)) *xf* (*xfu* ∘ (λ*x* -. *x*)), *m'*)
and *xf-xfu*: *fg-cons xf* (*xfu* ∘ (λ*x* -. *x*))
shows (*pa* ↪ *v*)(*lift-state h*) ⟹ *CTypesDefs.lift* (*fst h*) (*Ptr* &(*pa*→*f*)) = *xf v*
⟨*proof*⟩

lemma *lift-t-update-fld-other*:
fixes *val* :: *'b* :: *mem-type* **and** *ptr* :: *'a* :: *mem-type ptr*
assumes *fl*: *field-ti TYPE('a) f = Some (adjust-ti (typ-info-t TYPE('b)) xf (xfu ∘ (λx -. x)))*
and *xf-xfu*: *fg-cons xf (xfu ∘ (λx -. x))*
and *disj*: *typ-uinfo-t TYPE('c :: mem-type) ⊥t typ-uinfo-t TYPE('b)*
and *cl*: *lift-t c-guard hp ptr = Some z*
shows *(lift-t c-guard (hrs-mem-update (heap-update (Ptr &(ptr→f)) val) hp)) = (lift-t c-guard hp :: 'c :: mem-type typ-heap)*
(is *?LHS = ?RHS)*
⟨*proof*⟩

lemma *idupdate-compupdate-fold-congE*:
assumes *idu*: *⋀r. upd (λx. accr r) r = r*
assumes *cpu*: *⋀f f' r. upd f (upd f' r) = upd (f o f') r*
and *r*: *r = r'* **and** *v*: *accr r' = v'*
and *f*: *⋀v. v' = v ⟹ f v = f' v*
shows *upd f r = upd f' r'*
⟨*proof*⟩

lemma *field-lookup-field-ti*:
field-lookup (typ-info-t TYPE('a :: c-type)) f 0 ≡ Some (a, b) ⟹ field-ti TYPE('a) f = Some a
⟨*proof*⟩

definition *lvar-nondet-init* ::
(('a ⇒ 'a) ⇒ (('g, 'l, 'e, 'z) state-scheme ⇒ ('g, 'l, 'e, 'z) state-scheme))
⇒ (('g, 'l, 'e, 'z) state-scheme, 'f, 'x) com **where**
lvar-nondet-init upd ≡ Spec {(s, t). ∃ v. t = (upd (λ-. v)) s}

axiomatization
asm-semantics :: *string ⇒ addr list ⇒ (heap-mem × 'a) ⇒ (addr × (heap-mem × 'a)) set* **and**
asm-fetch :: *'s ⇒ (heap-mem × 'a)* **and**
asm-store :: *('s ⇒ 'b) ⇒ (heap-mem × 'a) ⇒ 's ⇒ 's*
where
asm-semantics-enabled: *∀ iv. asm-semantics nm addr iv ≠ {}* **and**
asm-store-eq: *∀ x s. ghost-data (asm-store ghost-data x s) = ghost-data s*

definition
asm-spec :: *'a itself ⇒ ('g ⇒ 'b) ⇒ bool ⇒ string*
⇒ (addr ⇒ ('g, 'l, 'e, 'z) state-scheme ⇒ ('g, 'l, 'e, 'z) state-scheme)
⇒ (('g, 'l, 'e, 'z) state-scheme ⇒ addr list)
⇒ (('g, 'l, 'e, 'z) state-scheme × ('g, 'l, 'e, 'z) state-scheme) set
where
asm-spec ti gdata vol spec lhs vs
= {(s, s'). ∃ (v', (gl' :: (heap-mem × 'a)))
∈ asm-semantics spec (vs s) (asm-fetch (globals s)).
s' = lhs v' (globals-update (asm-store gdata gl') s)}

lemma *asm-spec-enabled*:
$\exists s'.\ (s,\ s') \in$ *asm-spec ti gdata vol spec lhs vs*
⟨*proof*⟩

lemma *asm-specE*:
$[\![\ (s,\ s') \in$ *asm-spec* $(ti :: 'a\ itself)$ *gdata vol spec lhs vs*;
$\bigwedge v'\ gl'.\ [\![\ (v',\ (gl' :: (heap\text{-}mem \times 'a))) \in$ *asm-semantics spec* $(vs\ s)$ $(asm\text{-}fetch\ (globals\ s))$;
s′ = lhs v′ (*globals-update* (*asm-store gdata gl′*) *s*);
gdata (*asm-store gdata gl′* (*globals s*)) = *gdata* (*globals s*) $]\!]$
$\Longrightarrow P\ ]\!]$
$\Longrightarrow P$
⟨*proof*⟩

lemmas *state-eqE* = *arg-cong*[**where** *f*=$\lambda s.$ (*globals s*, *state.more s*), *elim-format*]

lemmas *asm-store-eq-helper*
= *arg-cong2*[**where** *f*=(=) **and** *a*=*asm-store f v s*]
arg-cong2[**where** *f*=(=) **and** *c*=*asm-store f v s*] **for** *f v s*

definition *asm-semantics-ok-to-ignore* :: $'a\ itself \Rightarrow bool \Rightarrow string \Rightarrow bool$ **where**
asm-semantics-ok-to-ignore ti volatile specifier
= ($\forall$ *xs gl. snd ‘ asm-semantics specifier xs* (*gl* :: (*heap-mem* × *′a*)) = {*gl*})

lemma *heap-list-nth*:
m < *n* $\Longrightarrow$ *heap-list hp n p* ! *m* = *hp* (*p* + *of-nat m*)
⟨*proof*⟩

lemma *heap-update-list-id′*:
heap-list hp n ptr = *xs* $\Longrightarrow$ *heap-update-list ptr xs hp* = *hp*
⟨*proof*⟩

lemma *heap-update-list-concat-fold*:
assumes *ptr′* = *ptr* + *of-nat* (*length ys*)
shows *heap-update-list ptr′ xs* (*heap-update-list ptr ys s*)
= *heap-update-list ptr* (*ys* @ *xs*) *s*
⟨*proof*⟩

lemma *heap-update-list-concat-fold-hrs-mem*:
ptr′ = *ptr* + *of-nat* (*length ys*) $\Longrightarrow$
hrs-mem-update (*heap-update-list ptr′ xs*)
(*hrs-mem-update* (*heap-update-list ptr ys*) *s*)
= *hrs-mem-update* (*heap-update-list ptr* (*ys* @ *xs*)) *s*
⟨*proof*⟩

lemmas *heap-update-list-concat-unfold*
= *heap-update-list-concat-fold*[*OF refl*, *symmetric*]

lemma *intvl-nowrap*:
fixes $x :: 'a::len\ word$
shows $[\![y \neq 0;\ unat\ y + z \leq 2\ \hat{}\ len\text{-}of\ TYPE('a)]\!] \Longrightarrow x \notin \{x + y\ ..+\ z\}$
⟨*proof*⟩

lemma *intvl-off-disj*:
fixes $x :: addr$
assumes *ylt*: $y \leq off$
and *zoff*: $z + off < 2\ \hat{}\ word\text{-}bits$
shows $\{x\ ..+\ y\} \cap \{x + of\text{-}nat\ off\ ..+\ z\} = \{\}$
⟨*proof*⟩

lemma *intvl-empty2*:
$(\{p\ ..+\ n\} = \{\}) = (n = 0)$
⟨*proof*⟩

lemma *heap-update-list-commute*:
$\{p\ ..+\ length\ xs\} \cap \{q\ ..+\ length\ ys\} = \{\}$
$\Longrightarrow$ *heap-update-list p xs* (*heap-update-list q ys hp*)
$=$ *heap-update-list q ys* (*heap-update-list p xs hp*)
⟨*proof*⟩

lemma *heap-update-commute*:
$[\![\ \{ptr\text{-}val\ p\ ..+\ size\text{-}of\ TYPE('a)\} \cap \{ptr\text{-}val\ q\ ..+\ size\text{-}of\ TYPE('b)\} = \{\};$
wf-fd (*typ-info-t TYPE*('*a*)); *wf-fd* (*typ-info-t TYPE*('*b*)) $]\!]$
$\Longrightarrow$ *heap-update p v* (*heap-update q* (*u* :: '*b* :: *c-type*) *h*)
$=$ *heap-update q u* (*heap-update p* (*v* :: '*a* :: *c-type*) *h*)
⟨*proof*⟩

lemma *heap-update-padding-commute*:
$[\![\ \{ptr\text{-}val\ p\ ..+\ size\text{-}of\ TYPE('a)\} \cap \{ptr\text{-}val\ q\ ..+\ size\text{-}of\ TYPE('b)\} = \{\};$
length bs $=$ *size-of TYPE*('*a*); *length bs*′ $=$ *size-of TYPE*('*b*);
wf-fd (*typ-info-t TYPE*('*a*)); *wf-fd* (*typ-info-t TYPE*('*b*))$]\!]$
$\Longrightarrow$ *heap-update-padding p v bs* (*heap-update-padding q* (*u* :: '*b* :: *c-type*) *bs*′ *h*)
$=$ *heap-update-padding q u bs*′ (*heap-update-padding p* (*v* :: '*a* :: *c-type*) *bs h*)
⟨*proof*⟩

lemma *heap-update-padding-heap-update-commute*:
$[\![\ \{ptr\text{-}val\ p\ ..+\ size\text{-}of\ TYPE('a)\} \cap \{ptr\text{-}val\ q\ ..+\ size\text{-}of\ TYPE('b)\} = \{\};$
length bs $=$ *size-of TYPE*('*a*);
wf-fd (*typ-info-t TYPE*('*a*)); *wf-fd* (*typ-info-t TYPE*('*b*))$]\!]$
$\Longrightarrow$ *heap-update-padding p v bs* (*heap-update q* (*u* :: '*b* :: *c-type*) *h*)
$=$ *heap-update q u* (*heap-update-padding p* (*v* :: '*a* :: *c-type*) *bs h*)

⟨*proof*⟩

lemma *heap-update-heap-update-padding-commute*:
⟦ {*ptr-val p ..+ size-of TYPE('a)*} ∩ {*ptr-val q ..+ size-of TYPE('b)*} = {};
length bs' = *size-of TYPE('b)*;
wf-fd (*typ-info-t TYPE('a)*); *wf-fd* (*typ-info-t TYPE('b)*)⟧
⟹ *heap-update p v* (*heap-update-padding q* (*u* :: *'b* :: *c-type*) *bs' h*)
= *heap-update-padding q u bs'* (*heap-update p* (*v* :: *'a* :: *c-type*) *h*)
⟨*proof*⟩

lemma *addr-card-wb*:
addr-card = *2* ^ *word-bits*
⟨*proof*⟩

lemma *fold-cong'*:
a = *b* ⟹ *xs* = *ys* ⟹ (⋀*x*. *x* ∈ *set xs* =*simp*=> *f x* = *g x*)
⟹ *fold f xs a* = *fold g ys b*
⟨*proof*⟩

lemma *arg-cong-meta*: *x* = *y* ⟹ (*f x* = *f y*) ≡ *True*
⟨*proof*⟩

⟨*ML*⟩

declare [[*simproc del*: *arg-cong*]]

lemma *fun-cong-meta*: *f* = *g* ⟹ (*f x* = *g x*) ≡ *True*
⟨*proof*⟩

⟨*ML*⟩

declare [[*simproc del*: *fun-cong*]]

abbreviation
ptr-span :: *'a*::*c-type ptr* ⇒ *addr set* **where**
ptr-span p ≡ {*ptr-val p ..+ size-of TYPE('a)*}

lemma *nat-index-bound*:
j ∗ *a* + *k* < *a* ∗ *b* **if** *jk*: *j* < *b k* < *a* **for** *j k* :: *nat*
⟨*proof*⟩

lemma *disj-ptr-span-ptr-neq*:
disjnt (*ptr-span* (*p*::*'a*::*mem-type ptr*)) (*ptr-span* (*q*::*'a*::*mem-type ptr*)) ⟹
p ≠ *q*
⟨*proof*⟩

lemma *field-lvalue-same-conv'*: &(*p*::*'a*:: *c-type ptr*→*f*) = &(*q*::*'a*:: *c-type ptr*→*f*)
⟷ *p* = *q*

⟨*proof*⟩

11.27 (Partial) Pointer Lenses

11.27.1 *pointer-lense*

named-theorems
read-write-same **and**
read-write-other **and**
write-cong **and**
map-other-commute

locale *pointer-lense* =
fixes *r*::'*s* ⇒ '*a*::*mem-type ptr* ⇒ '*b*
fixes *w*::'*a ptr* ⇒ ('*b* ⇒ '*b*) ⇒ '*s* ⇒ '*s*
assumes *read-write-same*[*read-write-same*]: *r* (*w p f s*) *p* = *f* (*r s p*)
assumes *write-same*: *f* (*r s p*) = (*r s p*) ⟹ *w p f s* = *s*
assumes *write-comp*[*update-compose*, *simp*]: *w p f* (*w p g s*) = *w p* (*f o g*) *s*
assumes *write-other-commute*[*map-other-commute*]: *disjnt* (*ptr-span p*) (*ptr-span q*) ⟹ *w q g* (*w p f s*) = *w p f* (*w q g s*)
begin

lemma *read-write-other*[*read-write-other*]:
disjnt (*ptr-span p*) (*ptr-span p'*) ⟹ *r* (*w p f s*) *p'* = *r s p'*
⟨*proof*⟩

lemma *write-cong*[*write-cong*]: *f* (*r s p*) = *f'* (*r s p*) ⟹ *w p f s* = *w p f' s*
⟨*proof*⟩

lemma *write-read*: *w p* (λ-. *r s p*) *s* = *s*
⟨*proof*⟩

lemma *write-id*: *w p* (λ*x*. *x*) *s* = *s*
⟨*proof*⟩

lemma *read-write-pointwise-id*: *r* (*w p* (λ*x*. *x*) *s*) = *r s*
⟨*proof*⟩

end

11.27.2 *partial-pointer-lense*

locale *partial-pointer-lense* = *is-scene m* **for** *m* :: '*a*::*c-type scene* +
fixes *r* :: '*h* ⇒ '*a ptr* ⇒ '*a upd*
fixes *w* :: '*a ptr* ⇒ '*a* ⇒ '*h upd*
assumes *r-w*[*simp*]: *r* (*w p x h*) *p y* = *m x y*
assumes *w-w*[*simp*]: *w p x* (*w p y h*) = *w p x h*
assumes *w-r*[*simp*]: *w p* (*r h p x*) *h* = *h*
assumes *w-m*[*simp*]: *w p* (*m x y*) *h* = *w p x h*

assumes *w-w-disj*: *disjnt* (*ptr-span p*) (*ptr-span q*) $\Longrightarrow$ *w p x* (*w q y h*) = *w q y* (*w p x h*)
begin

lemma *m-r*[*simp*]: *m* (*r h p x*) *y* = *r h p y*
⟨*proof*⟩

lemma *r-w-disj*[*simp*]: *disjnt* (*ptr-span p*) (*ptr-span q*) $\Longrightarrow$ *r* (*w q x h*) *p y* = *r h p y*
⟨*proof*⟩

lemma *r-m*: *r h p* (*m x y*) = *r h p y*
⟨*proof*⟩

end

lemma *partial-pointer-lenseI*:
fixes *get upd r w*
assumes *lense get upd*
assumes *pointer-lense r w*
shows *partial-pointer-lense*
(λ*a b*. *upd* (λ-. *get a*) *b*)
(λ*h p*. *upd* (λ-. *r h p*))
(λ*p x*. *w p* (λ-. *get x*))
⟨*proof*⟩

lemma *pointer-lense-of-lense-fld*:
assumes *lense r w*
shows *pointer-lense* (λ*h p*. *r h* (*PTR*(*'a*) &(*p*→*f*))) (λ*p v*. *w* (*upd-fun* (*PTR*(*'a*) &(*p*→*f*)) *v*))
⟨*proof*⟩

lemma *partial-pointer-lenseI-fld*:
fixes *get* :: *'a*::*mem-type* ⇒ *'b* **and** *upd r w*
assumes *1*: *lense get upd*
assumes *2*: *lense r w*
shows *partial-pointer-lense*
(λ*a b*. *upd* (λ-. *get a*) *b*)
(λ*h p*. *upd* (λ-. *r h* (*PTR*(*'b*) &(*p*→*f*))))
(λ*p x*. *w* (*upd-fun* (*PTR*(*'b*) &(*p*→*f*)) (λ-. *get x*)))
⟨*proof*⟩

lemma *partial-pointer-lense-compose*:
assumes *partial-pointer-lense m1 r1 w1*
assumes *partial-pointer-lense m2 r2 w2*
assumes *m*[*simp*]: $\bigwedge$*a b c*. *m1 a* (*m2 b c*) = *m2 b* (*m1 a c*)
assumes *w*[*simp*]: $\bigwedge$*p a q b h*. *p* = *q* ∨ *disjnt* (*ptr-span p*) (*ptr-span q*) $\Longrightarrow$
w1 p a (*w2 q b h*) = *w2 q b* (*w1 p a h*)
shows *partial-pointer-lense* (λ*a b*. *m1 a* (*m2 a b*))

(λh p x. r1 h p (r2 h p x))
(λp x h. w1 p x (w2 p x h))
⟨proof⟩

lemma *partial-pointer-lense-id*:
partial-pointer-lense (λa. *id*) (λh p. *id*) (λp x. *id*)
⟨*proof*⟩

lemma *partial-pointer-lense-fold*:
fixes *rs* :: ('h ⇒ 'a::*c-type ptr* ⇒ 'a ⇒ 'a) *list*
assumes *length ms* = *length rs length ms* = *length ws*
assumes *list-all* (λ(m, r, w). *partial-pointer-lense m r w*) (*zip ms* (*zip rs ws*))
assumes ∀ *a b c*. *distinct-prop* (λ*m1 m2*. *m1 a* (*m2 b c*) = *m2 b* (*m1 a c*)) *ms*
assumes ∀ *p a q b h*.
distinct-prop (λ*w1 w2*. *p* = *q* ∨ *disjnt* (*ptr-span p*) (*ptr-span q*) ⟶
w1 p a (*w2 q b h*) = *w2 q b* (*w1 p a h*)) *ws*
shows *partial-pointer-lense* (λ*a b*. *fold* (λ*m*. *m a*) *ms b*)
(λ*h p x*. *fold* (λ*r*. *r h p*) *rs x*)
(λ*p x h*. *fold* (λ*w*. *w p x*) *ws h*)
⟨*proof*⟩

lemma *pointer-lense-of-partial-pointer-lense*:
assumes *partial-pointer-lense m r w*
assumes [*simp*]: ⋀*a b*. *m a b* = *a*
shows *pointer-lense* (λ*h p*. *r h p x*) (λ*p f h*. *w p* (*f* (*r h p x*)) *h*)
⟨*proof*⟩

lemma *pointer-lense-of-partials*:
fixes *rs* :: ('h ⇒ 'a::*mem-type ptr* ⇒ 'a ⇒ 'a) *list*
assumes ∗:
length ms = *length rs length ms* = *length ws*
list-all (λ(m, r, w). *partial-pointer-lense m r w*) (*zip ms* (*zip rs ws*))
and ∗∗:
⋀*a b c*. *distinct-prop* (λ*m1 m2*. *m1 a* (*m2 b c*) = *m2 b* (*m1 a c*)) *ms*
⋀*p a q b h*. *distinct-prop* (λ*w1 w2*. *p* = *q* ∨ *disjnt* (*ptr-span p*) (*ptr-span q*) ⟶
w1 p a (*w2 q b h*) = *w2 q b* (*w1 p a h*)) *ws*
assumes *ms*: ⋀*a b*. *fold* (λ*m*. *m a*) *ms b* = *a*
assumes *R*: ⋀*h p*. *R h p* = *fold* (λ*r*. *r h p*) *rs x*
assumes *W*: ⋀*p f h*. *W p f h* = *fold* (λ*w*. *w p* (*f* (*R h p*))) *ws h*
shows *pointer-lense R W*
⟨*proof*⟩

end

theory *Padding-Equivalence*
imports
TypHeap
SepCode

CProof
More-Lib
begin

lemma *field-lookup-padding-field-name*:
fixes
t:: ('*a*, '*b*) *typ-info* **and**
st :: ('*a*, '*b*) *typ-info-struct* **and**
ts :: ('*a*, '*b*) *typ-info-tuple list* **and**
x :: ('*a*, '*b*) *typ-info-tuple*
shows
field-lookup t [*f*] *n* = *Some* (*s*, *m*) ⟹ *padding-field-name f* ⟹ *wf-padding t* ⟹
is-padding-tag s
field-lookup-struct st [*f*] *n* = *Some* (*s*, *m*) ⟹ *padding-field-name f* ⟹ *wf-padding-struct st* ⟹
is-padding-tag s
field-lookup-list ts [*f*] *n* = *Some* (*s*, *m*) ⟹ *padding-field-name f* ⟹ *wf-padding-list ts* ⟹
is-padding-tag s
field-lookup-tuple x [*f*] *n* = *Some* (*s*, *m*) ⟹ *padding-field-name f* ⟹ *wf-padding-tuple x* ⟹
is-padding-tag s
⟨*proof*⟩

lemma *field-lookup-access-ti-take-drop'*:
fixes *t*::('*a*, '*b*) *typ-info*
and *st*::('*a*, '*b*) *typ-info-struct*
and *ts*::('*a*, '*b*) *typ-info-tuple list*
and *x*::('*a*, '*b*) *typ-info-tuple*
shows
field-lookup t f m = *Some* (*s*, *m* + *n*) ⟹ *wf-fd t* ⟹ *wf-desc t* ⟹ *wf-size-desc t* ⟹ *length bs* = *size-td t* ⟹
access-ti s v (*take* (*size-td s*) (*drop n bs*)) =
(*take* (*size-td s*) (*drop n* (*access-ti t v bs*)))

field-lookup-struct st f m = *Some* (*s*, *m* + *n*) ⟹ *wf-fd-struct st* ⟹ *wf-desc-struct st* ⟹ *wf-size-desc-struct st* ⟹ *length bs* = *size-td-struct st* ⟹
access-ti s v (*take* (*size-td s*) (*drop n bs*)) =
(*take* (*size-td s*) (*drop n* (*access-ti-struct st v bs*)))

field-lookup-list ts f m = *Some* (*s*, *m* + *n*) ⟹ *wf-fd-list ts* ⟹ *wf-desc-list ts* ⟹ *wf-size-desc-list ts* ⟹ *length bs* = *size-td-list ts* ⟹
access-ti s v (*take* (*size-td s*) (*drop n bs*)) =
(*take* (*size-td s*) (*drop n* (*access-ti-list ts v bs*)))

field-lookup-tuple x f m = *Some* (*s*, *m* + *n*) ⟹ *wf-fd-tuple x* ⟹ *wf-desc-tuple x* ⟹ *wf-size-desc-tuple x* ⟹ *length bs* = *size-td-tuple x* ⟹
access-ti s v (*take* (*size-td s*) (*drop n bs*)) =
(*take* (*size-td s*) (*drop n* (*access-ti-tuple x v bs*)))

⟨*proof*⟩

lemma *field-lookup-access-ti-take-drop*:
field-lookup t f 0 = Some (s, n) ⟹ wf-fd t ⟹ wf-desc t ⟹ wf-size-desc t ⟹ length bs = size-td t ⟹
access-ti s v (take (size-td s) (drop n bs)) =
(take (size-td s) (drop n (access-ti t v bs)))
⟨*proof*⟩

lemma *field-lookup-nth-focus'*:
fixes *t*::('*a*, '*b*) *typ-info*
and *st*::('*a*, '*b*) *typ-info-struct*
and *ts*::('*a*, '*b*) *typ-info-tuple list*
and *x*::('*a*, '*b*) *typ-info-tuple*
shows
⟦*field-lookup t f m = Some (s, m + n)*; *n ≤ i*; *i < n + size-td s*; *length bs = size-td t*;
wf-fd t; *wf-desc t*; *wf-size-desc t*⟧ ⟹
access-ti t v bs ! i = access-ti s v (take (size-td s) (drop n bs)) ! (i − n)

⟦*field-lookup-struct st f m = Some (s, m + n)*; *n ≤ i*; *i < n + size-td s*; *length bs = size-td-struct st*;
wf-fd-struct st; *wf-desc-struct st*; *wf-size-desc-struct st*⟧ ⟹
access-ti-struct st v bs ! i = access-ti s v (take (size-td s) (drop n bs)) ! (i − n)

⟦*field-lookup-list ts f m = Some (s, m + n)*; *n ≤ i*; *i < n + size-td s*; *length bs = size-td-list ts*;
wf-fd-list ts; *wf-desc-list ts*; *wf-size-desc-list ts*⟧ ⟹
access-ti-list ts v bs ! i = access-ti s v (take (size-td s) (drop n bs)) ! (i − n)

⟦*field-lookup-tuple x f m = Some (s, m + n)*; *n ≤ i*; *i < n + size-td s*; *length bs = size-td-tuple x*;
wf-fd-tuple x; *wf-desc-tuple x*; *wf-size-desc-tuple x*⟧ ⟹
access-ti-tuple x v bs ! i = access-ti s v (take (size-td s) (drop n bs)) ! (i − n)
⟨*proof*⟩

lemma *field-lookup-nth-focus*:
⟦*field-lookup t f 0 = Some (s, n)*; *n ≤ i*; *i < n + size-td s*; *length bs = size-td t*;
wf-fd t; *wf-desc t*; *wf-size-desc t*⟧ ⟹
access-ti t v bs ! i = access-ti s v (take (size-td s) (drop n bs)) ! (i − n)
⟨*proof*⟩

lemma *field-lookup-nth-update-focus'*:
fixes *t*::('*a*, '*b*) *typ-info*
and *st*::('*a*, '*b*) *typ-info-struct*
and *ts*::('*a*, '*b*) *typ-info-tuple list*
and *x*::('*a*, '*b*) *typ-info-tuple*
shows

⟦*field-lookup t f m* = *Some* (*s*, *m* + *n*); *n* ≤ *i*; *i* < *n* + *size-td s*; *length bs* = *size-td t*;
wf-fd t; *wf-desc t*; *wf-size-desc t*⟧ ⟹
access-ti t v (*bs*[*i* := *b*]) =
super-update-bs (*access-ti s v* ((*take* (*size-td s*) (*drop n bs*))[*i* − *n* := *b*]))
(*access-ti t v bs*) *n*

⟦*field-lookup-struct st f m* = *Some* (*s*, *m* + *n*); *n* ≤ *i*; *i* < *n* + *size-td s*; *length bs* = *size-td-struct st*;
wf-fd-struct st; *wf-desc-struct st*; *wf-size-desc-struct st*⟧ ⟹
access-ti-struct st v (*bs*[*i* := *b*]) =
super-update-bs (*access-ti s v* ((*take* (*size-td s*) (*drop n bs*))[*i* − *n* := *b*]))
(*access-ti-struct st v bs*) *n*

⟦*field-lookup-list ts f m* = *Some* (*s*, *m* + *n*); *n* ≤ *i*; *i* < *n* + *size-td s*; *length bs* = *size-td-list ts*;
wf-fd-list ts; *wf-desc-list ts*; *wf-size-desc-list ts*⟧ ⟹
access-ti-list ts v (*bs*[*i* := *b*]) =
super-update-bs (*access-ti s v* ((*take* (*size-td s*) (*drop n bs*))[*i* − *n* := *b*]))
(*access-ti-list ts v bs*) *n*

⟦*field-lookup-tuple x f m* = *Some* (*s*, *m* + *n*); *n* ≤ *i*; *i* < *n* + *size-td s*; *length bs* = *size-td-tuple x*;
wf-fd-tuple x; *wf-desc-tuple x*; *wf-size-desc-tuple x*⟧ ⟹
access-ti-tuple x v (*bs*[*i* := *b*]) =
super-update-bs (*access-ti s v* ((*take* (*size-td s*) (*drop n bs*))[*i* − *n* := *b*]))
(*access-ti-tuple x v bs*) *n*
⟨*proof*⟩

lemma *field-lookup-nth-update-focus*:
shows
⟦*field-lookup t f 0* = *Some* (*s*, *n*); *n* ≤ *i*; *i* < *n* + *size-td s*; *length bs* = *size-td t*;
wf-fd t; *wf-desc t*; *wf-size-desc t*⟧ ⟹
access-ti t v (*bs*[*i* := *b*]) =
super-update-bs (*access-ti s v* ((*take* (*size-td s*) (*drop n bs*))[*i* − *n* := *b*]))
(*access-ti t v bs*) *n*
⟨*proof*⟩

context *mem-type*
begin
lemma *mem-type-field-lookup-access-ti-take-drop*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*)
assumes *lbs* : *length bs* = *size-of TYPE*('*a*)
shows *access-ti s v* (*take* (*size-td s*) (*drop n bs*)) =
take (*size-td s*) (*drop n* (*access-ti* (*typ-info-t TYPE*('*a*)) *v bs*))
⟨*proof*⟩

lemma *mem-type-update-ti-super-update-bs*:

assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*)
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lv*: *length v* = *size-td s*
shows *update-ti s v* (*update-ti* (*typ-info-t TYPE*('*a*)) *bs w*) =
update-ti (*typ-info-t TYPE*('*a*)) (*super-update-bs v bs n*) *w*
⟨*proof*⟩

lemma *mem-type-update-ti-from-bytes-super-update-bs*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*)
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lv*: *length v* = *size-td s*
shows *update-ti s v* (*from-bytes bs*) = *update-ti* (*typ-info-t TYPE*('*a*)) (*super-update-bs v bs n*) *undefined*
⟨*proof*⟩

lemma *mem-type-field-lookup-nth-focus*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*)
assumes *i-lower*: *n* ≤ *i*
assumes *i-upper*: *i* < *n* + *size-td s*
assumes *lbs* : *length bs* = *size-of TYPE*('*a*)
shows *access-ti* (*typ-info-t TYPE*('*a*)) *v bs* ! *i* =
access-ti s v (*take* (*size-td s*) (*drop n bs*)) ! (*i* − *n*)
⟨*proof*⟩

lemma *mem-type-field-lookup-nth-update-focus*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*)
assumes *i-lower*: *n* ≤ *i*
assumes *i-upper*: *i* < *n* + *size-td s*
assumes *lbs* : *length bs* = *size-of TYPE*('*a*)
shows
access-ti (*typ-info-t TYPE*('*a*)) *v* (*bs*[*i* := *b*]) =
super-update-bs (*access-ti s v* ((*take* (*size-td s*) (*drop n bs*))[*i* − *n* := *b*]))
(*access-ti* (*typ-info-t TYPE*('*a*)) *v bs*) *n*
⟨*proof*⟩
end

⟨*ML*⟩

lemma *nth-take-drop-index-shift*:
n ≤ *i* ⟹ *i* < *m* + *n* ⟹ *m* + *n* ≤ *length xs* ⟹ *take m* (*drop n xs*) ! (*i* − *n*) = *xs* ! *i*
⟨*proof*⟩

lemma *super-update-bs-update-index-shift*: *n* ≤ *i* ⟹ *i* − *n* < *length bs* ⟹ *n* + *length bs* ≤ *length xbs* ⟹

(*super-update-bs bs xbs n*)[*i* := *b*] = *super-update-bs* (*bs*[*i* − *n* := *b*]) *xbs n*
⟨*proof*⟩

lemma *super-update-bs-nth-shift*:
n ≤ *i* ⟹ *i* − *n* < *length bs* ⟹ *n* + *length bs* ≤ *length xbs* ⟹ *super-update-bs bs xbs n* ! *i* = *bs* ! (*i* − *n*)
⟨*proof*⟩

lemma (**in** *mem-type*) *field-lookup-is-padding-byte-outer-to-inner*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f* 0 = *Some* (*s*, *n*)
assumes *lower-bound-x*: *n* ≤ *x*
assumes *upper-bound-x*: *x* < *n* + *size-td s*
assumes *is-padding*: *padding-base.is-padding-byte* (*access-ti* (*typ-info-t* (*TYPE*('*a*)))) (*update-ti* (*typ-info-t* (*TYPE*('*a*)))) (*size-td* (*typ-info-t* (*TYPE*('*a*)))) *x*
shows *padding-base.is-padding-byte* (*access-ti s*) (*update-ti s*) (*size-td s*) (*x* − *n*)
⟨*proof*⟩

lemma (**in** *mem-type*) *field-lookup-is-padding-byte-inner-to-outer*:
assumes *fl*: *field-lookup* (*typ-info-t* (*TYPE*('*a*))) *f* 0 = *Some* (*s*, *n*)
assumes *lower-bound-x*: *n* ≤ *x*
assumes *upper-bound-x*: *x* < *n* + *size-td s*
assumes *is-padding*: *padding-base.is-padding-byte* (*access-ti s*) (*update-ti s*) (*size-td s*) (*x* − *n*)
shows *padding-base.is-padding-byte* (*access-ti* (*typ-info-t* (*TYPE*('*a*)))) (*update-ti* (*typ-info-t* (*TYPE*('*a*)))) (*size-td* (*typ-info-t* (*TYPE*('*a*)))) *x*
⟨*proof*⟩

lemma (**in** *mem-type*) *field-lookup-is-padding-byte*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f* 0 = *Some* (*s*, *n*)
assumes *lower-bound-x*: *n* ≤ *x*
assumes *upper-bound-x*: *x* < *n* + *size-td s*
shows
padding-base.is-padding-byte (*access-ti s*) (*update-ti s*) (*size-td s*) (*x* − *n*) ⟷
padding-base.is-padding-byte
(*access-ti* (*typ-info-t* (*TYPE*('*a*)))) (*update-ti* (*typ-info-t* (*TYPE*('*a*)))) (*size-td* (*typ-info-t* (*TYPE*('*a*)))) *x*
⟨*proof*⟩

lemma (**in** *mem-type*) *field-lookup-is-value-byte-outer-to-inner*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f* 0 = *Some* (*s*, *n*)
assumes *lower-bound-x*: *n* ≤ *x*
assumes *upper-bound-x*: *x* < *n* + *size-td s*
assumes *is-value*: *padding-base.is-value-byte* (*access-ti* (*typ-info-t* (*TYPE*('*a*)))) (*update-ti* (*typ-info-t* (*TYPE*('*a*)))) (*size-td* (*typ-info-t* (*TYPE*('*a*)))) *x*
shows *padding-base.is-value-byte* (*access-ti s*) (*update-ti s*) (*size-td s*) (*x* − *n*)
⟨*proof*⟩

lemma (**in** *mem-type*) *field-lookup-is-value-byte-inner-to-outer*:

assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*)
assumes *lower-bound-x*: $n \leq x$
assumes *upper-bound-x*: $x < n +$ *size-td s*
assumes *is-value*: *padding-base.is-value-byte* (*access-ti s*) (*update-ti s*) (*size-td s*) ($x - n$)
shows *padding-base.is-value-byte* (*access-ti* (*typ-info-t* (*TYPE*('*a*)))) (*update-ti* (*typ-info-t* (*TYPE*('*a*)))) (*size-td* (*typ-info-t* (*TYPE*('*a*)))) *x*
⟨*proof*⟩

lemma (**in** *mem-type*) *field-lookup-is-value-byte*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*)
assumes *lower-bound-x*: $n \leq x$
assumes *upper-bound-x*: $x < n +$ *size-td s*
shows *padding-base.is-value-byte* (*access-ti s*) (*update-ti s*) (*size-td s*) ($x - n$) ⟷
padding-base.is-value-byte (*access-ti* (*typ-info-t* (*TYPE*('*a*))))
(*update-ti* (*typ-info-t* (*TYPE*('*a*)))) (*size-td* (*typ-info-t* (*TYPE*('*a*)))) *x*
⟨*proof*⟩

thm *padding-base.eq-padding-def*
thm *padding-base.eq-upto-padding-def*

lemma *td-set-component-descs-independent*:
fixes *t*::'*a xtyp-info*
and *st*::'*a xtyp-info-struct*
and *ts*::'*a xtyp-info-tuple list*
and *x*::'*a xtyp-info-tuple*
shows
(*s*, *n*) ∈ *td-set t m* ⟹ *component-descs-independent t* ⟹ *component-descs-independent s*
(*s*, *n*) ∈ *td-set-struct st m* ⟹ *component-descs-independent-struct st* ⟹ *component-descs-independent s*
(*s*, *n*) ∈ *td-set-list ts m* ⟹ *component-descs-independent-list ts* ⟹ *component-descs-independent s*
(*s*, *n*) ∈ *td-set-tuple x m* ⟹ *component-descs-independent-tuple x* ⟹ *component-descs-independent s*
⟨*proof*⟩

lemma *td-set-wf-component-descs*:
fixes *t*::'*a xtyp-info*
and *st*::'*a xtyp-info-struct*
and *ts*::'*a xtyp-info-tuple list*
and *x*::'*a xtyp-info-tuple*
shows
(*s*, *n*) ∈ *td-set t m* ⟹ *wf-component-descs t* ⟹ *wf-component-descs s*

$(s, n) \in$ *td-set-struct st m* $\Longrightarrow$ *wf-component-descs-struct st* $\Longrightarrow$ *wf-component-descs s*
$(s, n) \in$ *td-set-list ts m* $\Longrightarrow$ *wf-component-descs-list ts* $\Longrightarrow$ *wf-component-descs s*
$(s, n) \in$ *td-set-tuple x m* $\Longrightarrow$ *wf-component-descs-tuple x* $\Longrightarrow$ *wf-component-descs s*
⟨*proof*⟩

lemma *td-set-field-descs*:
fixes *t*::'*a xtyp-info*
and *st*::'*a xtyp-info-struct*
and *ts*::'*a xtyp-info-tuple list*
and *x*::'*a xtyp-info-tuple*
shows
$(s, n) \in$ *td-set t m* $\Longrightarrow$ *wf-component-descs t* $\Longrightarrow$ *component-desc s* $\in$ *insert* (*component-desc t*) (*set* (*field-descs t*))
$(s, n) \in$ *td-set-struct st m* $\Longrightarrow$ *wf-component-descs-struct st* $\Longrightarrow$ *component-desc s* $\in$ (*set* (*field-descs-struct st*))
$(s, n) \in$ *td-set-list ts m* $\Longrightarrow$ *wf-component-descs-list ts* $\Longrightarrow$ *component-desc s* $\in$ (*set* (*field-descs-list ts*))
$(s, n) \in$ *td-set-tuple x m* $\Longrightarrow$ *wf-component-descs-tuple x* $\Longrightarrow$ *component-desc s* $\in$ (*set* (*field-descs-tuple x*))
⟨*proof*⟩

lemma *td-set-wf-field-descs*:
fixes *t*::'*a xtyp-info*
and *st*::'*a xtyp-info-struct*
and *ts*::'*a xtyp-info-tuple list*
and *x*::'*a xtyp-info-tuple*
shows
$(s, n) \in$ *td-set t m* $\Longrightarrow$ *wf-field-descs* (*set* (*field-descs t*)) $\Longrightarrow$ *wf-field-descs* (*set* (*field-descs s*))
$(s, n) \in$ *td-set-struct st m* $\Longrightarrow$ *wf-field-descs* (*set* (*field-descs-struct st*)) $\Longrightarrow$ *wf-field-descs* (*set* (*field-descs s*))
$(s, n) \in$ *td-set-list ts m* $\Longrightarrow$ *wf-field-descs* (*set* (*field-descs-list ts*)) $\Longrightarrow$ *wf-field-descs* (*set* (*field-descs s*))
$(s, n) \in$ *td-set-tuple x m* $\Longrightarrow$ *wf-field-descs* (*set* (*field-descs-tuple x*)) $\Longrightarrow$ *wf-field-descs* (*set* (*field-descs s*))
⟨*proof*⟩

context *xmem-type*
begin
lemma *xmem-type-td-set-field-descs*:
$(s,n) \in$ *td-set* (*typ-info-t TYPE*('*a*)) *m* $\Longrightarrow$
component-desc s $\in$ *insert* (*component-desc* (*typ-info-t TYPE*('*a*))) (*set* (*field-descs* (*typ-info-t TYPE*('*a*))))
⟨*proof*⟩

lemma *field-lookup-component-desc*:

field-lookup (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*) ⟹
component-desc s ∈ *insert* (*component-desc* (*typ-info-t TYPE*('*a*))) (*set* (*field-descs* (*typ-info-t TYPE*('*a*))))
⟨*proof*⟩

lemma *field-lookup-wf-field-desc*:
field-lookup (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*) ⟹
wf-field-desc (*component-desc s*)
⟨*proof*⟩

lemma *field-lookup-component-descs-independent*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*)
shows *component-descs-independent s*
⟨*proof*⟩

lemma *field-lookup-wf-component-descs*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*)
shows *wf-component-descs s*
⟨*proof*⟩

lemma *field-lookup-wf-field-descs*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*)
shows *wf-field-descs* (*set* (*field-descs s*))
⟨*proof*⟩

lemma *field-lookup-field-access-access-ti*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*)
shows *field-access* (*component-desc s*) = *access-ti s*
⟨*proof*⟩

lemma *field-lookup-field-update-update-ti*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*)
shows *field-update* (*component-desc s*) = *update-ti s*
⟨*proof*⟩

lemma *field-lookup-field-sz-size-td*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*)
shows *field-sz* (*component-desc s*) = *size-td s*
⟨*proof*⟩

lemma *field-lookup-component-desc-complement-padding*:
field-lookup (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*) ⟹
complement-padding (*field-access* (*component-desc s*)) (*field-update* (*component-desc s*)) (*field-sz* (*component-desc s*))
⟨*proof*⟩

lemma *field-lookup-component-desc-complement-padding'*:

assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*)
shows *complement-padding* (*access-ti s*) (*update-ti s*) (*size-td s*)
⟨*proof*⟩

lemma *field-lookup-padding-lense*:
field-lookup (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*) ⟹
padding-lense (*access-ti s*) (*update-ti s*) (*size-td s*)
⟨*proof*⟩

sublocale *lense*: *padding-lense*
access-ti (*typ-info-t TYPE*('*a*))
update-ti (*typ-info-t TYPE*('*a*))
size-of TYPE('*a*)
⟨*proof*⟩

end

lemma *eq-padding-access-update-field-cancel*:
assumes *fl*: *field-lookup* (*typ-info-t* (*TYPE*('*a*::*xmem-type*))) *f 0* = *Some* (*s*, *n*)
assumes *lower-bound-x*: *n* ≤ *x*
assumes *upper-bound-x*: *x* < *n* + *size-of TYPE*('*b*)
assumes *match*: *export-uinfo s* = *typ-uinfo-t TYPE*('*b*::*xmem-type*)
assumes *lbs*: *length bs* = *size-of TYPE*('*b*)
assumes *lbs'*: *length bs'* = *size-of TYPE*('*a*)
assumes *eq-padding*: *padding-base.eq-padding* (*access-ti s*) (*size-td s*) *bs* (*take* (*size-of TYPE*('*b*)) (*drop n bs'*))
shows *access-ti* (*typ-info-t TYPE*('*a*::*xmem-type*)) (*update-ti s bs v*) *bs'* ! *x* = *bs* ! (*x* − *n*)
⟨*proof*⟩

context *xmem-type*
begin

sublocale *xmem-type-padding*: *complement-padding*
access-ti (*typ-info-t TYPE*('*a*))
update-ti (*typ-info-t TYPE*('*a*))
size-of TYPE('*a*)
⟨*proof*⟩
end

lemma *drop-heap-list-le2*:
heap-list h n (*x* + *of-nat k*)
= *drop k* (*heap-list h* (*n* + *k*) *x*)
⟨*proof*⟩

lemma *heap-list-take-drop*:
assumes *N-bound*: *unat a* + *N* ≤ *2* ^ *len-of TYPE*(*addr-bitsize*)
shows *n* + *m* ≤ *N* ⟹ *take m* (*drop n* (*heap-list hp N a*)) =

heap-list hp m (a + word-of-nat n)
⟨*proof*⟩

lemma *heap-list-take-drop'*:
assumes *N-bound*: *unat a + N ≤ addr-card*
assumes *bound*: *n + m ≤ N*
shows *take m (drop n (heap-list hp N a)) =*
heap-list hp m (a + word-of-nat n)
⟨*proof*⟩

experiment
fixes *proj*:: *'a*::*xmem-type ⇒ 'b*::*xmem-type*
fixes *t*::*'a xtyp-info*

assumes *fl*: *field-lookup (typ-info-t TYPE('a)) f 0 = Some (t, n)*
and *eu*: *export-uinfo t = typ-uinfo-t TYPE('b)*
assumes *access-comp*: *access-ti t v = access-ti (typ-info-t TYPE('b)) (proj v)*
assumes *surj*: *surj proj*
begin

lemma *sz*: *size-td t = size-of TYPE('b)*
⟨*proof*⟩

lemma *padding-base.eq-padding (access-ti t) (size-td t) =*
padding-base.eq-padding (access-ti (typ-info-t TYPE('b))) (size-of TYPE('b))
⟨*proof*⟩
end

definition *is-padding-byte*::*typ-uinfo ⇒ nat ⇒ bool* **where**
is-padding-byte t i ≡ i < size-td t ∧
(∀ bs b. length bs = size-td t ⟶ norm-tu t (bs[i := b]) = norm-tu t bs)

definition *is-value-byte*::*typ-uinfo ⇒ nat ⇒ bool* **where**
is-value-byte t i ≡ i < size-td t ∧
(∃ bs b. length bs = size-td t ∧ norm-tu t (bs[i := b]) ≠ norm-tu t bs)

lemma *is-padding-byteI*:
assumes *i < size-td t*
assumes *⋀i bs b. length bs = size-td t ⟹ norm-tu t (bs[i := b]) = norm-tu t bs*
shows *is-padding-byte t i*
⟨*proof*⟩

lemma *complement-tu-padding*:
i < size-td t ⟹ is-padding-byte t i ≠ is-value-byte t i
⟨*proof*⟩

definition *eq-padding*::*typ-uinfo ⇒ byte list ⇒ byte list ⇒ bool* **where**

eq-padding t bs bs′ ≡ *length bs* = *size-td t* ∧ *length bs′* = *size-td t* ∧
(∀ *i*. *is-padding-byte t i* ⟶ *bs* ! *i* = *bs′* ! *i*)

definition *eq-upto-padding*::*typ-uinfo* ⇒ *byte list* ⇒ *byte list* ⇒ *bool* **where**
eq-upto-padding t bs bs′ ≡ *length bs* = *size-td t* ∧ *length bs′* = *size-td t* ∧
(∀ *i*. *is-value-byte t i* ⟶ *bs* ! *i* = *bs′* ! *i*)

lemma *eq-padding-refl*[*simp*]: *length bs* = *size-td t* ⟹ *eq-padding t bs bs*
⟨*proof*⟩

lemma *eq-upto-padding-refl*[*simp*]: *length bs* = *size-td t* ⟹ *eq-upto-padding t bs bs*
⟨*proof*⟩

lemma *eq-padding-sym*: *eq-padding t bs bs′* ⟷ *eq-padding t bs′ bs*
⟨*proof*⟩

lemma *eq-padding-symp*: *symp* (*eq-padding t*)
⟨*proof*⟩

lemma *eq-upto-padding-sym*: *eq-upto-padding t bs bs′* ⟷ *eq-upto-padding t bs′ bs*
⟨*proof*⟩

lemma *eq-upto-padding-symp*: *symp* (*eq-upto-padding t*)
⟨*proof*⟩

lemma *eq-padding-trans*: *eq-padding t bs1 bs2* ⟹ *eq-padding t bs2 bs3* ⟹ *eq-padding t bs1 bs3*
⟨*proof*⟩

lemma *eq-padding-transp*: *transp* (*eq-padding t*)
⟨*proof*⟩

lemma *eq-upto-padding-trans*: *eq-upto-padding t bs1 bs2* ⟹ *eq-upto-padding t bs2 bs3* ⟹ *eq-upto-padding t bs1 bs3*
⟨*proof*⟩

lemma *eq-upto-padding-transp*: *transp* (*eq-upto-padding t*)
⟨*proof*⟩

lemma *eq-padding-eq-upto-padding-eq*: *eq-padding t bs bs′* ⟹ *eq-upto-padding t bs bs′* ⟹ *bs* = *bs′*
⟨*proof*⟩

thm *padding-base.is-padding-byte-def*

lemma *is-padding-byte-access-ti'*:
fixes *t*::'*a xtyp-info*
and *st*::'*a xtyp-info-struct*
and *ts*::'*a xtyp-info-tuple list*
and *x*::'*a xtyp-info-tuple*
shows
⟦ *wf-desc t*; *wf-size-desc t*; *wf-field-descs* (*set* (*field-descs t*)); *wf-component-descs t*; *wf-fd t*;
i < *size-td t*; *length bs* = *size-td t*;
∀ *bs b*. *length bs* = *size-td t* ⟶ *norm-tu* (*map-td field-norm* (λ-. ()) *t*) (*bs*[*i* := *b*]) = *norm-tu* (*export-uinfo t*) *bs*⟧
⟹ *access-ti t v bs* ! *i* = *bs* ! *i*

⟦ *wf-desc-struct st*; *wf-size-desc-struct st*; *wf-field-descs* (*set* (*field-descs-struct st*)); *wf-component-descs-struct st*; *wf-fd-struct st*; *i* < *size-td-struct st*; *length bs* = *size-td-struct st*;
∀ *bs b*. *length bs* = *size-td-struct st* ⟶ *norm-tu-struct* (*map-td-struct field-norm* (λ-. ()) *st*) (*bs*[*i* := *b*]) = *norm-tu-struct* ((*map-td-struct field-norm* (λ-. ()) *st*)) *bs*⟧
⟹ *access-ti-struct st v bs* ! *i* = *bs* ! *i*

⟦ *wf-desc-list ts*; *wf-size-desc-list ts*; *wf-field-descs* (*set* (*field-descs-list ts*)); *wf-component-descs-list ts*; *wf-fd-list ts*; *i* < *size-td-list ts*; *length bs* = *size-td-list ts*;
∀ *bs b*. *length bs* = *size-td-list ts* ⟶ *norm-tu-list* (*map-td-list field-norm* (λ-. ()) *ts*) (*bs*[*i* := *b*]) = *norm-tu-list* ((*map-td-list field-norm* (λ-. ()) *ts*)) *bs*⟧
⟹ *access-ti-list ts v bs* ! *i* = *bs* ! *i*

⟦ *wf-desc-tuple x*; *wf-size-desc-tuple x*; *wf-field-descs* (*set* (*field-descs-tuple x*)); *wf-component-descs-tuple x*; *wf-fd-tuple x*; *i* < *size-td-tuple x*; *length bs* = *size-td-tuple x*;
∀ *bs b*. *length bs* = *size-td-tuple x* ⟶ *norm-tu-tuple* (*map-td-tuple field-norm* (λ-. ()) *x*) (*bs*[*i* := *b*]) = *norm-tu-tuple* ((*map-td-tuple field-norm* (λ-. ()) *x*)) *bs*⟧
⟹ *access-ti-tuple x v bs* ! *i* = *bs* ! *i*
⟨*proof*⟩

lemma *is-padding-byte-access-ti*:
assumes *wf*: *wf-desc* (*t*::'*a xtyp-info*)
and *wf-sz*: *wf-size-desc t*
and *wf-descs*: *wf-field-descs* (*set* (*field-descs t*))
and *wf-comp*: *wf-component-descs t*
and *wf-fd*: *wf-fd t*
and *i-bound*: *i* < *size-td t*
and *lbs*: *length bs* = *size-td t*
and *is-padding*: *is-padding-byte* (*export-uinfo t*) *i*
shows *access-ti t v bs* ! *i* = *bs* ! *i*
⟨*proof*⟩

context *xmem-type*

begin
lemma *xmem-type-is-padding-byte-access-ti*:
assumes *padding*: *is-padding-byte* (*typ-uinfo-t TYPE*('*a*)) *i*
and *i-bound*: *i* < *size-of TYPE*('*a*)
and *lbs*: *length bs* = *size-of TYPE*('*a*)
shows *access-ti* (*typ-info-t TYPE*('*a*)) *v bs* ! *i* = *bs* ! *i*
⟨*proof*⟩

end

lemma *is-padding-byte-update-ti-id'*:
fixes *t*::'*a xtyp-info*
and *st*::'*a xtyp-info-struct*
and *ts*::'*a xtyp-info-tuple list*
and *x*::'*a xtyp-info-tuple*
shows
⟦ *wf-desc t*; *wf-size-desc t*; *wf-field-descs* (*set* (*field-descs t*)); *wf-component-descs t*; *wf-fd t*;
i < *size-td t*; *length bs* = *size-td t*;
∀ *bs b*. *length bs* = *size-td t* ⟶ *norm-tu* (*map-td field-norm* (λ-. ()) *t*) (*bs*[*i* := *b*]) = *norm-tu* (*export-uinfo t*) *bs*⟧
⟹ *update-ti t* (*bs*[*i* := *b*]) *v* = *update-ti t bs v*

⟦ *wf-desc-struct st*; *wf-size-desc-struct st*; *wf-field-descs* (*set* (*field-descs-struct st*));
wf-component-descs-struct st; *wf-fd-struct st*; *i* < *size-td-struct st*; *length bs* = *size-td-struct st*;
∀ *bs b*. *length bs* = *size-td-struct st* ⟶ *norm-tu-struct* (*map-td-struct field-norm* (λ-. ()) *st*) (*bs*[*i* := *b*]) = *norm-tu-struct* ((*map-td-struct field-norm* (λ-. ()) *st*)) *bs*⟧
⟹ *update-ti-struct st* (*bs*[*i* := *b*]) *v* = *update-ti-struct st bs v*

⟦ *wf-desc-list ts*; *wf-size-desc-list ts*; *wf-field-descs* (*set* (*field-descs-list ts*));
wf-component-descs-list ts; *wf-fd-list ts*; *i* < *size-td-list ts*; *length bs* = *size-td-list ts*;
∀ *bs b*. *length bs* = *size-td-list ts* ⟶ *norm-tu-list* (*map-td-list field-norm* (λ-. ()) *ts*) (*bs*[*i* := *b*]) = *norm-tu-list* ((*map-td-list field-norm* (λ-. ()) *ts*)) *bs*⟧
⟹ *update-ti-list ts* (*bs*[*i* := *b*]) *v* = *update-ti-list ts bs v*

⟦ *wf-desc-tuple x*; *wf-size-desc-tuple x*; *wf-field-descs* (*set* (*field-descs-tuple x*));
wf-component-descs-tuple x; *wf-fd-tuple x*; *i* < *size-td-tuple x*; *length bs* = *size-td-tuple x*;
∀ *bs b*. *length bs* = *size-td-tuple x* ⟶ *norm-tu-tuple* (*map-td-tuple field-norm* (λ-. ()) *x*) (*bs*[*i* := *b*]) = *norm-tu-tuple* ((*map-td-tuple field-norm* (λ-. ()) *x*)) *bs*⟧
⟹ *update-ti-tuple x* (*bs*[*i* := *b*]) *v* = *update-ti-tuple x bs v*
⟨*proof*⟩

lemma *is-padding-byte-update-ti-id*:
  assumes *wf*: *wf-desc* (*t*::'*a xtyp-info*)
  and *wf-sz*: *wf-size-desc t*
  and *wf-descs*: *wf-field-descs* (*set* (*field-descs t*))
  and *wf-comp*: *wf-component-descs t*
  and *wf-fd*: *wf-fd t*
  and *i-bound*: *i* < *size-td t*
  and *lbs*: *length bs* = *size-td t*
  and *is-padding*: *is-padding-byte* (*export-uinfo t*) *i*
shows *update-ti t* (*bs*[*i* := *b*]) *v* = *update-ti t bs v*
  ⟨*proof*⟩

context *xmem-type*
begin
lemma *xmem-type-is-padding-byte-update-ti-id*:
  assumes *padding*: *is-padding-byte* (*typ-uinfo-t TYPE*('*a*)) *i*
  and *i-bound*: *i* < *size-of TYPE*('*a*)
  and *lbs*: *length bs* = *size-of TYPE*('*a*)
shows *update-ti* (*typ-info-t TYPE*('*a*)) (*bs*[*i*:=*b*]) *v* = *update-ti* (*typ-info-t TYPE*('*a*)) *bs v*
⟨*proof*⟩
end

lemma *heap-update-fold*:
sz = *size-of TYPE*('*a*) ⟹
  heap-update-list a (*to-bytes* (*v*::'*a*:: *c-type*) (*heap-list h sz a*)) *h* = *heap-update* (*Ptr a*) *v h*
  ⟨*proof*⟩

lemma *heap-update-padding-fold*:
sz = *size-of TYPE*('*a*) ⟹
  heap-update-list a (*to-bytes* (*v*::'*a*:: *c-type*) *bs*) *h* = *heap-update-padding* (*Ptr a*) *v bs h*
  ⟨*proof*⟩

context *padding-base*
begin
thm *is-padding-byte-def*
thm *is-value-byte-def*
end

lemma *is-value-byte-access-ti-id'*:
  fixes *t*::'*a xtyp-info*
  and *st*::'*a xtyp-info-struct*
  and *ts*::'*a xtyp-info-tuple list*
  and *x*::'*a xtyp-info-tuple*

shows
⟦ *wf-desc t*; *wf-size-desc t*; *wf-field-descs* (*set* (*field-descs t*)); *wf-component-descs t*; *wf-fd t*;
i < *size-td t*; *length bs* = *size-td t*;
∃ *bs b*. *length bs* = *size-td t* ∧ *norm-tu* (*map-td field-norm* (λ-. ()) *t*) (*bs*[*i* := *b*]) ≠ *norm-tu* (*export-uinfo t*) *bs*⟧
⟹ *access-ti t v* (*bs*[*i* := *b*]) = *access-ti t v bs*

⟦ *wf-desc-struct st*; *wf-size-desc-struct st*; *wf-field-descs* (*set* (*field-descs-struct st*));
wf-component-descs-struct st; *wf-fd-struct st*; *i* < *size-td-struct st*; *length bs* = *size-td-struct st*;
∃ *bs b*. *length bs* = *size-td-struct st* ∧ *norm-tu-struct* (*map-td-struct field-norm* (λ-. ()) *st*) (*bs*[*i* := *b*]) ≠ *norm-tu-struct* ((*map-td-struct field-norm* (λ-. ()) *st*)) *bs*⟧
⟹ *access-ti-struct st v* (*bs*[*i* := *b*]) = *access-ti-struct st v bs*

⟦ *wf-desc-list ts*; *wf-size-desc-list ts*; *wf-field-descs* (*set* (*field-descs-list ts*));
wf-component-descs-list ts; *wf-fd-list ts*; *i* < *size-td-list ts*; *length bs* = *size-td-list ts*;
∃ *bs b*. *length bs* = *size-td-list ts* ∧ *norm-tu-list* (*map-td-list field-norm* (λ-. ()) *ts*) (*bs*[*i* := *b*]) ≠ *norm-tu-list* ((*map-td-list field-norm* (λ-. ()) *ts*)) *bs*⟧
⟹ *access-ti-list ts v* (*bs*[*i* := *b*]) = *access-ti-list ts v bs*

⟦ *wf-desc-tuple x*; *wf-size-desc-tuple x*; *wf-field-descs* (*set* (*field-descs-tuple x*));
wf-component-descs-tuple x; *wf-fd-tuple x*; *i* < *size-td-tuple x*; *length bs* = *size-td-tuple x*;
∃ *bs b*. *length bs* = *size-td-tuple x* ∧ *norm-tu-tuple* (*map-td-tuple field-norm* (λ-. ()) *x*) (*bs*[*i* := *b*]) ≠ *norm-tu-tuple* ((*map-td-tuple field-norm* (λ-. ()) *x*)) *bs*⟧
⟹ *access-ti-tuple x v* (*bs*[*i* := *b*]) = *access-ti-tuple x v bs*
⟨*proof*⟩

lemma *is-value-byte-access-ti-id*:
assumes *wf*: *wf-desc t*
assumes *wf-sz*: *wf-size-desc t*
assumes *wf-descs*: *wf-field-descs* (*set* (*field-descs t*))
assumes *wf-comps*: *wf-component-descs t*
assumes *wf-fd*: *wf-fd t*
assumes *i-bound*: *i* < *size-td t*
assumes *lbs*: *length bs* = *size-td t*
assumes *is-value*: *is-value-byte* (*export-uinfo t*) *i*
shows *access-ti t v* (*bs*[*i* := *b*]) = *access-ti t v bs*
⟨*proof*⟩

lemma *is-value-byte-access-ti-update-ti-cancel'*:
fixes *t*::'*a xtyp-info*
and *st*::'*a xtyp-info-struct*
and *ts*::'*a xtyp-info-tuple list*
and *x*::'*a xtyp-info-tuple*

shows
⟦ *wf-desc t*; *wf-size-desc t*; *wf-field-descs* (*set* (*field-descs t*)); *wf-component-descs t*; *wf-fd t*;
i < *size-td t*; *length bs* = *size-td t*; *length bs′* = *size-td t*;
∃ *bs b*. *length bs* = *size-td t* ∧ *norm-tu* (*map-td field-norm* (λ-. ()) *t*) (*bs*[*i* := *b*]) ≠ *norm-tu* (*export-uinfo t*) *bs*⟧
⟹ *access-ti t* (*update-ti t bs v*) *bs′* ! *i* = *bs* ! *i*

⟦ *wf-desc-struct st*; *wf-size-desc-struct st*; *wf-field-descs* (*set* (*field-descs-struct st*));
wf-component-descs-struct st; *wf-fd-struct st*; *i* < *size-td-struct st*; *length bs* = *size-td-struct st*; *length bs′* = *size-td-struct st*;
∃ *bs b*. *length bs* = *size-td-struct st* ∧ *norm-tu-struct* (*map-td-struct field-norm* (λ-. ()) *st*) (*bs*[*i* := *b*]) ≠ *norm-tu-struct* ((*map-td-struct field-norm* (λ-. ()) *st*)) *bs*⟧
⟹ *access-ti-struct st* (*update-ti-struct st bs v*) *bs′* ! *i* = *bs* ! *i*

⟦ *wf-desc-list ts*; *wf-size-desc-list ts*; *wf-field-descs* (*set* (*field-descs-list ts*));
wf-component-descs-list ts; *wf-fd-list ts*; *i* < *size-td-list ts*; *length bs* = *size-td-list ts*; *length bs′* = *size-td-list ts*;
∃ *bs b*. *length bs* = *size-td-list ts* ∧ *norm-tu-list* (*map-td-list field-norm* (λ-. ()) *ts*) (*bs*[*i* := *b*]) ≠ *norm-tu-list* ((*map-td-list field-norm* (λ-. ()) *ts*)) *bs*⟧
⟹ *access-ti-list ts* (*update-ti-list ts bs v*) *bs′* ! *i* = *bs* ! *i*

⟦ *wf-desc-tuple x*; *wf-size-desc-tuple x*; *wf-field-descs* (*set* (*field-descs-tuple x*));
wf-component-descs-tuple x; *wf-fd-tuple x*; *i* < *size-td-tuple x*; *length bs* = *size-td-tuple x*; *length bs′* = *size-td-tuple x*;
∃ *bs b*. *length bs* = *size-td-tuple x* ∧ *norm-tu-tuple* (*map-td-tuple field-norm* (λ-. ()) *x*) (*bs*[*i* := *b*]) ≠ *norm-tu-tuple* ((*map-td-tuple field-norm* (λ-. ()) *x*)) *bs*⟧
⟹ *access-ti-tuple x* (*update-ti-tuple x bs v*) *bs′* ! *i* = *bs* ! *i*
⟨*proof*⟩

lemma *is-value-byte-access-ti-update-ti-cancel*:
assumes *wf*: *wf-desc t*
assumes *wf-sz*: *wf-size-desc t*
assumes *wf-descs*: *wf-field-descs* (*set* (*field-descs t*))
assumes *wf-comps*: *wf-component-descs t*
assumes *wf-fd*: *wf-fd t*
assumes *i-bound*: *i* < *size-td t*
assumes *lbs*: *length bs* = *size-td t*
assumes *lbs′*: *length bs′*= *size-td t*
assumes *is-value*: *is-value-byte* (*export-uinfo t*) *i*
shows *access-ti t* (*update-ti t bs v*) *bs′* ! *i* = *bs* ! *i*
⟨*proof*⟩

lemma *field-lookup-is-padding-byte-focus′*:
fixes *t*::('*a*, '*b*) *typ-info*
and *st*::('*a*, '*b*) *typ-info-struct*
and *ts*::('*a*, '*b*) *typ-info-tuple list*
and *x*::('*a*, '*b*) *typ-info-tuple*

shows
⟦*field-lookup t f m* = *Some* (*s*, *m* + *n*); *n* ≤ *i*; *i* < *n* + *size-td s*; *length bs* = *size-td t*;
wf-desc t; *wf-size-desc t*⟧ ⟹
norm-tu (*map-td field-norm* (λ-. ()) *t*) (*bs*[*i* := *b*]) = *norm-tu* (*map-td field-norm* (λ-. ()) *t*) *bs*
⟷
norm-tu (*export-uinfo s*) ((*take* (*size-td s*) (*drop n bs*))[*i* − *n* := *b*]) =
norm-tu (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*))

⟦*field-lookup-struct st f m* = *Some* (*s*, *m* + *n*); *n* ≤ *i*; *i* < *n* + *size-td s*; *length bs* = *size-td-struct st*;
wf-desc-struct st; *wf-size-desc-struct st*⟧ ⟹
norm-tu-struct (*map-td-struct field-norm* (λ-. ()) *st*) (*bs*[*i* := *b*]) = *norm-tu-struct* (*map-td-struct field-norm* (λ-. ()) *st*) *bs*
⟷
norm-tu (*export-uinfo s*) ((*take* (*size-td s*) (*drop n bs*))[*i* − *n* := *b*]) =
norm-tu (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*))

⟦*field-lookup-list ts f m* = *Some* (*s*, *m* + *n*); *n* ≤ *i*; *i* < *n* + *size-td s*; *length bs* = *size-td-list ts*;
wf-desc-list ts; *wf-size-desc-list ts*⟧ ⟹
norm-tu-list (*map-td-list field-norm* (λ-. ()) *ts*) (*bs*[*i* := *b*]) = *norm-tu-list* (*map-td-list field-norm* (λ-. ()) *ts*) *bs*
⟷
norm-tu (*export-uinfo s*) ((*take* (*size-td s*) (*drop n bs*))[*i* − *n* := *b*]) =
norm-tu (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*))

⟦*field-lookup-tuple x f m* = *Some* (*s*, *m* + *n*); *n* ≤ *i*; *i* < *n* + *size-td s*; *length bs* = *size-td-tuple x*;
wf-desc-tuple x; *wf-size-desc-tuple x*⟧ ⟹
norm-tu-tuple (*map-td-tuple field-norm* (λ-. ()) *x*) (*bs*[*i* := *b*]) = *norm-tu-tuple* (*map-td-tuple field-norm* (λ-. ()) *x*) *bs*
⟷
norm-tu (*export-uinfo s*) ((*take* (*size-td s*) (*drop n bs*))[*i* − *n* := *b*]) =
norm-tu (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*))
⟨*proof*⟩

lemma *field-lookup-is-padding-byte-focus*:
assumes *fl*: *field-lookup t f 0* = *Some* (*s*, *n*)
assumes *i-lower*: *n* ≤ *i*
assumes *i-upper* :*i* < *n* + *size-td s*
assumes *wf*: *wf-desc t*
assumes *wf-sz*: *wf-size-desc t*
shows *is-padding-byte* (*export-uinfo t*) *i* = *is-padding-byte* (*export-uinfo s*) (*i* − *n*)
⟨*proof*⟩

lemma *field-lookup-is-value-byte-focus*:

assumes *fl*: *field-lookup t f 0 = Some* (*s*, *n*)
assumes *i-lower*: $n \leq i$
assumes *i-upper* :$i < n + size\text{-}td\ s$
assumes *wf*: *wf-desc t*
assumes *wf-sz*: *wf-size-desc t*
shows *is-value-byte* (*export-uinfo t*) *i* = *is-value-byte* (*export-uinfo s*) ($i - n$)
⟨*proof*⟩

lemma *field-lookup-eq-padding-focus*:
assumes *fl*: *field-lookup t f 0 = Some* (*s*, *n*)
assumes *lbs*: *length bs = size-td t*
assumes *lbs'*: *length bs' = size-td t*
assumes *wf*: *wf-desc t*
assumes *wf-sz*: *wf-size-desc t*
assumes *eq-padding*: *eq-padding* (*export-uinfo t*) *bs bs'*
shows *eq-padding* (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs'*))
⟨*proof*⟩

lemma *field-lookup-eq-padding-focus-eq*:
assumes *fl*: *field-lookup t f 0 = Some* (*s*, *n*)
assumes *lbs*: *length bs = size-td t*
assumes *lbs'*: *length bs' = size-td t*
assumes *pfx-eq*: $\bigwedge i.\ i < n \Longrightarrow bs\ !\ i = bs'\ !\ i$
assumes *sfx-eq*: $\bigwedge i.\ n + size\text{-}td\ s \leq i \Longrightarrow i < size\text{-}td\ t \Longrightarrow bs\ !\ i = bs'\ !\ i$
assumes *wf*: *wf-desc t*
assumes *wf-sz*: *wf-size-desc t*
shows *eq-padding* (*export-uinfo t*) *bs bs'* =
eq-padding (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs'*))
⟨*proof*⟩

lemma *field-lookup-eq-padding-super-update-bs*:
assumes *fl*: *field-lookup t f 0 = Some* (*s*, *n*)
assumes *lxbs*: *length xbs = size-td t*
assumes *lbs*: *length bs = size-td s*
assumes *lbs'*: *length bs' = size-td s*
assumes *wf*: *wf-desc t*
assumes *wf-sz*: *wf-size-desc t*
shows *eq-padding* (*export-uinfo t*) (*super-update-bs bs xbs n*) (*super-update-bs bs' xbs n*) =
eq-padding (*export-uinfo s*) *bs bs'*
⟨*proof*⟩

lemma *field-lookup-eq-upto-padding-focus*:

assumes *fl*: *field-lookup t f 0 = Some (s, n)*
assumes *lbs*: *length bs = size-td t*
assumes *lbs'*: *length bs' = size-td t*
assumes *wf*: *wf-desc t*
assumes *wf-sz*: *wf-size-desc t*
assumes *eq-upto-padding*: *eq-upto-padding (export-uinfo t) bs bs'*
shows *eq-upto-padding (export-uinfo s) (take (size-td s) (drop n bs)) (take (size-td s) (drop n bs'))*
⟨*proof*⟩

lemma *field-lookup-eq-upto-padding-focus-eq*:
assumes *fl*: *field-lookup t f 0 = Some (s, n)*
assumes *lbs*: *length bs = size-td t*
assumes *lbs'*: *length bs' = size-td t*
assumes *pfx-eq*: $\bigwedge i.\ i < n \Longrightarrow bs\ !\ i = bs'\ !\ i$
assumes *sfx-eq*: $\bigwedge i.\ n + size\text{-}td\ s \leq i \Longrightarrow i < size\text{-}td\ t \Longrightarrow bs\ !\ i = bs'\ !\ i$
assumes *wf*: *wf-desc t*
assumes *wf-sz*: *wf-size-desc t*
shows *eq-upto-padding (export-uinfo t) bs bs' =*
eq-upto-padding (export-uinfo s) (take (size-td s) (drop n bs)) (take (size-td s) (drop n bs'))
⟨*proof*⟩

lemma *field-lookup-eq-upto-padding-super-update-bs*:
assumes *fl*: *field-lookup t f 0 = Some (s, n)*
assumes *lxbs*: *length xbs = size-td t*
assumes *lbs*: *length bs = size-td s*
assumes *lbs'*: *length bs' = size-td s*
assumes *wf*: *wf-desc t*
assumes *wf-sz*: *wf-size-desc t*
shows *eq-upto-padding (export-uinfo t) (super-update-bs bs xbs n) (super-update-bs bs' xbs n) =*
eq-upto-padding (export-uinfo s) bs bs'
⟨*proof*⟩

lemma *field-lookup-wf-size-desc-pres*:
fixes *t*::('*a*, '*b*) *typ-desc*
and *st*::('*a*, '*b*) *typ-struct*
and *ts*::('*a*, '*b*) *typ-tuple list*
and *x*::('*a*, '*b*) *typ-tuple*
shows
wf-size-desc t ⟹ field-lookup t f n = Some (s, m) ⟹ wf-size-desc s
wf-size-desc-struct st ⟹ field-lookup-struct st f n = Some (s, m) ⟹ wf-size-desc s
wf-size-desc-list ts ⟹ field-lookup-list ts f n = Some (s, m) ⟹ wf-size-desc s
wf-size-desc-tuple x ⟹ field-lookup-tuple x f n = Some (s, m) ⟹ wf-size-desc s
⟨*proof*⟩

lemma *field-lookup-update-ti-super-update-bs-conv*:
fixes *t*::('*a*,'*b*) *typ-info*
and *st*::('*a*,'*b*) *typ-info-struct*
and *ts*::('*a*,'*b*) *typ-info-tuple list*
and *x*::('*a*,'*b*) *typ-info-tuple*
shows
⟦*field-lookup t f m* = *Some* (*s*, *m* + *n*); *length bs* = *size-td s*; *length xbs* = *size-td t*;
wf-fd t; *wf-desc t*; *wf-size-desc t*⟧ ⟹
update-ti s bs v =
update-ti t (*super-update-bs bs* (*access-ti t v xbs*) *n*) *v*

⟦*field-lookup-struct st f m* = *Some* (*s*, *m* + *n*); *length bs* = *size-td s*; *length xbs* = *size-td-struct st*;
wf-fd-struct st; *wf-desc-struct st*; *wf-size-desc-struct st*⟧ ⟹
update-ti s bs v =
update-ti-struct st (*super-update-bs bs* (*access-ti-struct st v xbs*) *n*) *v*

⟦*field-lookup-list ts f m* = *Some* (*s*, *m* + *n*); *length bs* = *size-td s*; *length xbs* = *size-td-list ts*;
wf-fd-list ts; *wf-desc-list ts*; *wf-size-desc-list ts*⟧ ⟹
update-ti s bs v =
update-ti-list ts (*super-update-bs bs* (*access-ti-list ts v xbs*) *n*) *v*

⟦*field-lookup-tuple x f m* = *Some* (*s*, *m* + *n*); *length bs* = *size-td s*; *length xbs* = *size-td-tuple x*;
wf-fd-tuple x; *wf-desc-tuple x*; *wf-size-desc-tuple x*⟧ ⟹
update-ti s bs v =
update-ti-tuple x (*super-update-bs bs* (*access-ti-tuple x v xbs*) *n*) *v*
⟨*proof*⟩

lemma *access-ti-super-update-bs-of-wf*:
fixes *t*::('*a*, '*b*) *typ-info*
and *st*::('*a*, '*b*) *typ-info-struct*
and *ts*::('*a*, '*b*) *typ-info-tuple list*
and *x*::('*a*, '*b*) *typ-info-tuple*
shows
⟦*field-lookup t f m* = *Some* (*s*, *m* + *n*); *length bs'* = *size-td s*; *length bs* = *size-td t*;
wf-fd t; *wf-desc t*; *wf-size-desc t*⟧ ⟹
access-ti t (*update-ti s* (*access-ti*$_0$ *s w*) *v*) (*super-update-bs bs' bs n*) =
super-update-bs (*access-ti s w bs'*) (*access-ti t v bs*) *n*

⟦*field-lookup-struct st f m* = *Some* (*s*, *m* + *n*); *length bs'* = *size-td s*; *length bs* = *size-td-struct st*;
wf-fd-struct st; *wf-desc-struct st*; *wf-size-desc-struct st*⟧ ⟹
access-ti-struct st (*update-ti s* (*access-ti*$_0$ *s w*) *v*) (*super-update-bs bs' bs n*) =
super-update-bs (*access-ti s w bs'*) (*access-ti-struct st v bs*) *n*

⟦*field-lookup-list ts f m = Some (s, m + n)*; *length bs′ = size-td s*; *length bs = size-td-list ts*;
wf-fd-list ts; *wf-desc-list ts*; *wf-size-desc-list ts*⟧ ⟹
access-ti-list ts (*update-ti s* (*access-ti*$_0$ *s w*) *v*) (*super-update-bs bs′ bs n*) =
super-update-bs (*access-ti s w bs′*) (*access-ti-list ts v bs*) *n*

⟦*field-lookup-tuple x f m = Some (s, m + n)*; *length bs′ = size-td s*; *length bs = size-td-tuple x*;
wf-fd-tuple x; *wf-desc-tuple x*; *wf-size-desc-tuple x*⟧ ⟹
access-ti-tuple x (*update-ti s* (*access-ti*$_0$ *s w*) *v*) (*super-update-bs bs′ bs n*) =
super-update-bs (*access-ti s w bs′*) (*access-ti-tuple x v bs*) *n*
⟨*proof*⟩

lemma *heap-update-list-same*:
assumes *p-no-overflow*: *unat p + length bs ≤ addr-card*
assumes *same-pfx*: ⋀*i*. *i < length bs* ⟹ *bs ! i = hp* (*p + word-of-nat i*)
shows *heap-update-list p bs hp = hp*
⟨*proof*⟩

lemma *heap-update-list-same-prefix*:
assumes *p-no-overflow*: *unat p + length pfx + length bs ≤ addr-card*
assumes *same-pfx*: ⋀*i*. *i < length pfx* ⟹ *pfx ! i = hp* (*p + word-of-nat i*)
shows *heap-update-list p* (*pfx @ bs*) *hp = heap-update-list* (*p + word-of-nat* (*length pfx*)) *bs hp*
⟨*proof*⟩

lemma *heap-update-list-same-suffix*:
assumes *p-no-overflow*: *unat p + length sfx + length bs ≤ addr-card*
assumes *same-sfx*: ⋀*i*. *i < length sfx* ⟹ *sfx ! i = hp* (*p + word-of-nat* (*length bs + i*))
shows *heap-update-list p* (*bs @ sfx*) *hp = heap-update-list p bs hp*
⟨*proof*⟩

lemma *heap-update-list-same-prefix-suffix*:
assumes *p-no-overflow*: *unat p + length pfx + length bs + length sfx ≤ addr-card*
assumes *same-pfx*: ⋀*i*. *i < length pfx* ⟹ *pfx ! i = hp* (*p + word-of-nat i*)
assumes *same-sfx*: ⋀*i*. *i < length sfx* ⟹ *sfx ! i = hp* (*p + word-of-nat* (*length pfx + length bs + i*))
shows *heap-update-list p* (*pfx @ bs @ sfx*) *hp = heap-update-list* (*p + word-of-nat* (*length pfx*)) *bs hp*
⟨*proof*⟩

lemma *heap-update-list-super-update-bs-heap-list*:
assumes *p-no-overflow*: *unat p + m ≤ addr-card*
assumes *n-m*: *n + length bs ≤ m*
shows *heap-update-list* (*p + word-of-nat n*) *bs hp = heap-update-list p* (*super-update-bs bs* (*heap-list hp m p*) *n*) *hp*
⟨*proof*⟩

lemma *append-take-dropI*: *xs* = *take* (*length xs*) *zs* ⟹ *ys* = *drop* (*length xs*) *zs* ⟹ *xs* @ *ys* = *zs*
⟨*proof*⟩

lemma *heap-list-heap-update-list-id*:
assumes *bound*: *n* ≤ *addr-card*
assumes *lbs*: *length bs* = *n*
shows (*heap-list* (*heap-update-list a bs h*) *n a*) = *bs*
⟨*proof*⟩

lemma *heap-update-list-nth-conv*:
length bs ≤ *addr-card* ⟹
heap-update-list a bs h a' =
(*if* (*unat a'* + (*if a'* < *a then addr-card else 0*) − *unat a*) < *length bs then*
bs ! (*unat a'* + (*if a'* < *a then addr-card else 0*) − *unat a*)
else
h a')
⟨*proof*⟩

lemma *heap-update-list-overwrite-all-nth-conv*:
assumes *lbs*: *length bs* = *addr-card*
shows *heap-update-list a bs h a'* =
bs ! (*unat a'* + (*if a'* < *a then addr-card else 0*) − *unat a*)
⟨*proof*⟩

lemma *heap-update-list-overwrite'*:
assumes *bound*: *length bs* ≤ *addr-card*
assumes *len-eq*: *length bs* = *length bs'*
shows (*heap-update-list a bs* (*heap-update-list a bs' h*)) = (*heap-update-list a bs h*)
⟨*proof*⟩

lemma *heap-list-tail-addr-card*:
assumes *len*: *length ys* = *addr-card*
shows *heap-update-list p* (*xs* @ *ys*) *h* = *heap-update-list* (*p* + *word-of-nat* (*length xs*)) *ys h*
⟨*proof*⟩

lemma *heap-update-list-overwrite*:
assumes *leq*: *length bs* = *length bs'*
shows *heap-update-list p bs* (*heap-update-list p bs' h*) = *heap-update-list p bs h*
⟨*proof*⟩

context *xmem-type*

begin

lemma *xmem-type-is-value-byte-access-ti-update-ti-cancel*:
assumes *i-bound*: $i < size\text{-}of\ TYPE('a)$
assumes *lbs*: $length\ bs = size\text{-}of\ TYPE('a)$
assumes *lbs'*: $length\ bs' = size\text{-}of\ TYPE('a)$
assumes *is-value*: $is\text{-}value\text{-}byte\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ i$
shows $access\text{-}ti\ (typ\text{-}info\text{-}t\ TYPE('a))\ (update\text{-}ti\ (typ\text{-}info\text{-}t\ TYPE('a))\ bs\ v)\ bs'\ !\ i = bs\ !\ i$
⟨*proof*⟩

lemma *xmem-type-is-value-byte-access-ti-id*:
assumes *i-bound*: $i < size\text{-}of\ TYPE('a)$
assumes *lbs*: $length\ bs = size\text{-}of\ TYPE('a)$
assumes *is-value*: $is\text{-}value\text{-}byte\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ i$
shows $access\text{-}ti\ (typ\text{-}info\text{-}t\ TYPE('a))\ v\ (bs[i:=b]) = access\text{-}ti\ (typ\text{-}info\text{-}t\ TYPE('a))\ v\ bs$
⟨*proof*⟩

lemma *is-padding-byte-to-lense*: $is\text{-}padding\text{-}byte\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ i \Longrightarrow lense.is\text{-}padding\text{-}byte\ i$
⟨*proof*⟩

lemma *is-value-byte-to-lense*: $is\text{-}value\text{-}byte\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ i \Longrightarrow lense.is\text{-}value\text{-}byte\ i$
⟨*proof*⟩

lemma *is-padding-byte-from-lense*:
assumes *padding*: $lense.is\text{-}padding\text{-}byte\ i$
shows $is\text{-}padding\text{-}byte\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ i$
⟨*proof*⟩

lemma *is-padding-byte-lense-conv*: $is\text{-}padding\text{-}byte\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ i = lense.is\text{-}padding\text{-}byte\ i$
⟨*proof*⟩

lemma *is-value-byte-from-lense*:
assumes *is-value*: $lense.is\text{-}value\text{-}byte\ i$
shows $is\text{-}value\text{-}byte\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ i$
⟨*proof*⟩

lemma *is-value-byte-lense-conv*: $is\text{-}value\text{-}byte\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ i = lense.is\text{-}value\text{-}byte\ i$
⟨*proof*⟩

lemma *eq-padding-lense-conv*: $eq\text{-}padding\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ bs\ bs' = lense.eq\text{-}padding\ bs\ bs'$

⟨*proof*⟩

lemma *eq-upto-padding-lense-conv*: *eq-upto-padding* (*typ-uinfo-t TYPE*('*a*)) *bs bs*′ = *lense.eq-upto-padding bs bs*′
⟨*proof*⟩

lemma *lense-eq-upto-padding-from-bytes-eq*:
assumes *eq-upto-padding*: *lense.eq-upto-padding bs bs*′
shows ((*from-bytes bs*)::′*a*) = *from-bytes bs*′
⟨*proof*⟩

lemma *eq-upto-padding-from-bytes-eq*:
assumes *eq-upto-padding*: *eq-upto-padding* (*typ-uinfo-t TYPE*('*a*)) *bs bs*′
shows ((*from-bytes bs*)::′*a*) = *from-bytes bs*′
⟨*proof*⟩

lemma *lense-eq-padding-to-bytes-eq*:
assumes *eq-padding*: *lense.eq-padding bs bs*′
shows (*to-bytes* (*v*::′*a*) *bs*) = *to-bytes v bs*′
⟨*proof*⟩

lemma *eq-padding-to-bytes-eq*:
assumes *eq-padding*: *eq-padding* (*typ-uinfo-t TYPE*('*a*)) *bs bs*′
shows (*to-bytes* (*v*::′*a*) *bs*) = *to-bytes v bs*′
⟨*proof*⟩

lemma *eq-padding-to-bytes*:
length bs = *size-of TYPE*('*a*) ⟹ *eq-padding* (*typ-uinfo-t TYPE*('*a*)) (*to-bytes* (*v*::′*a*) *bs*) *bs*
⟨*proof*⟩

lemma *lense-eq-padding-to-bytes*:
length bs = *size-of TYPE*('*a*) ⟹ *lense.eq-padding* (*to-bytes* (*v*::′*a*) *bs*) *bs*
⟨*proof*⟩

lemma *heap-list-heap-update-id*:
fixes *p*::′*a ptr*
shows (*heap-list*
(*heap-update-list* (*ptr-val p*)
(*to-bytes* (*u*::′*a*) (*heap-list h* (*size-of TYPE*('*a*)) (*ptr-val p*))) *h*)
(*size-of TYPE*('*a*)) (*ptr-val p*)) =
(*to-bytes u* (*heap-list h* (*size-of TYPE*('*a*)) (*ptr-val p*)))
⟨*proof*⟩

lemma *heap-update-collapse*:
fixes *p*::′*a ptr*
shows *heap-update p v* (*heap-update p u h*) = *heap-update p v h*

⟨*proof*⟩

lemma *heap-update-padding-collapse*:
fixes *p*::'*a ptr*
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lbs'*: *length bs'* = *size-of TYPE*('*a*)
shows *heap-update-padding p v bs* (*heap-update-padding p u bs' h*) = *heap-update-padding p v bs h*
⟨*proof*⟩

lemma *heap-update-padding-heap-update-collapse*:
fixes *p*::'*a ptr*
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
shows *heap-update-padding p v bs* (*heap-update p u h*) = *heap-update-padding p v bs h*
⟨*proof*⟩

lemma *to-bytes-from-bytes-id*:
length bs = *size-of TYPE*('*a*) ⟹ *to-bytes* ((*from-bytes bs*)::'*a*) *bs* = *bs*
⟨*proof*⟩

lemma *to-bytes-heap-list-id*:
to-bytes ((*from-bytes* (*heap-list h* (*size-of TYPE*('*a*)) *a*))::'*a*)
(*heap-list h* (*size-of TYPE*('*a*)) *a*) =
heap-list h (*size-of TYPE*('*a*)) *a*
⟨*proof*⟩

lemma *heap-update-id*:
fixes *p*::'*a ptr*
shows *heap-update p* (*h-val h p*) *h* = *h*
⟨*proof*⟩

context
fixes *s f n*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*)
begin
interpretation *flense*: *padding-lense access-ti s update-ti s size-td s*
⟨*proof*⟩ **lemma** *n-t*: *n* < *size-of TYPE*('*a*)
⟨*proof*⟩ **lemma** *wf-desc-s*: *wf-desc s*
⟨*proof*⟩ **lemma** *wf-size-desc-s*: *wf-size-desc s*
⟨*proof*⟩ **lemma** *wf-field-descs-s*: *wf-field-descs* (*set* (*field-descs s*))
⟨*proof*⟩ **lemma** *wf-component-descs-s*: *wf-component-descs s*
⟨*proof*⟩ **lemma** *wf-fd-s*: *wf-fd s*
⟨*proof*⟩

lemma *xmem-type-field-lookup-is-padding-byte-focus*:
assumes *i-lower*: *n* ≤ *i*

assumes *i-upper* :$i < n$ + *size-td s*
shows *is-padding-byte* (*typ-uinfo-t TYPE*('*a*)) *i* = *is-padding-byte* (*export-uinfo s*) (*i* − *n*)
⟨*proof*⟩

lemma *xmem-type-field-lookup-is-padding-byte-focus-rev1*:
assumes *is-padding*: *is-padding-byte* (*export-uinfo s*) *i*
shows *is-padding-byte* (*typ-uinfo-t TYPE*('*a*)) (*i* + *n*)
⟨*proof*⟩

lemma *xmem-type-field-lookup-is-padding-byte-focus-rev2*:
assumes *is-padding*: *is-padding-byte* (*typ-uinfo-t TYPE*('*a*)) (*i* + *n*)
assumes *i-bound*: *i* < *size-td s*
shows *is-padding-byte* (*export-uinfo s*) *i*
⟨*proof*⟩

lemma *xmem-type-field-lookup-lense-is-padding-byte-focus-rev1*:
assumes *is-padding*: *flense.is-padding-byte i*
shows *lense.is-padding-byte* (*i* + *n*)
⟨*proof*⟩

lemma *xmem-type-field-lookup-lense-is-padding-byte-focus-rev2*:
assumes *is-padding*: *lense.is-padding-byte* (*i* + *n*)
assumes *i-bound*: *i* < *size-td s*
shows *flense.is-padding-byte* *i*
⟨*proof*⟩

lemma *xmem-type-field-lookup-is-value-byte-focus*:
assumes *i-lower*: *n* ≤ *i*
assumes *i-upper* :$i < n$ + *size-td s*
shows *is-value-byte* (*typ-uinfo-t TYPE*('*a*)) *i* = *is-value-byte* (*export-uinfo s*) (*i* − *n*)
⟨*proof*⟩

lemma *xmem-type-field-lookup-is-value-byte-focus-rev1*:
assumes *is-value*: *is-value-byte* (*export-uinfo s*) *i*
shows *is-value-byte* (*typ-uinfo-t TYPE*('*a*)) (*i* + *n*)
⟨*proof*⟩

lemma *xmem-type-field-lookup-is-value-byte-focus-rev2*:
assumes *is-value*: *is-value-byte* (*typ-uinfo-t TYPE*('*a*)) (*i* + *n*)
assumes *i-bound*: *i* < *size-td s*
shows *is-value-byte* (*export-uinfo s*) *i*
⟨*proof*⟩

lemma *xmem-type-field-lookup-lense-is-value-byte-focus-rev1*:
assumes *is-value*: *flense.is-value-byte i*

shows *lense.is-value-byte* $(i + n)$
⟨*proof*⟩

lemma *xmem-type-field-lookup-lense-is-value-byte-focus-rev2*:
assumes *is-value*: *lense.is-value-byte* $(i + n)$
assumes *i-bound*: $i <$ *size-td s*
shows *flense.is-value-byte i*
⟨*proof*⟩

lemma *field-lookup-is-padding-byte-access-ti*:
assumes *i-bound*: $i <$ *size-td s*
assumes *lbs*: *length bs* $=$ *size-td s*
assumes *is-padding*: *is-padding-byte* (*export-uinfo s*) *i*
shows *access-ti s v bs* ! *i* $=$ *bs* ! *i*
⟨*proof*⟩

lemma *field-lookup-is-padding-byte-update-ti-id*:
assumes *i-bound*: $i <$ *size-td s*
assumes *lbs*: *length bs* $=$ *size-td s*
assumes *is-padding*: *is-padding-byte* (*export-uinfo s*) *i*
shows *update-ti s* (*bs*[*i* := *b*]) *v* $=$ *update-ti s bs v*
⟨*proof*⟩

lemma *field-lookup-is-value-byte-access-ti-update-ti-cancel*:
assumes *i-bound*: $i <$ *size-td s*
assumes *lbs*: *length bs* $=$ *size-td s*
assumes *lbs'*: *length bs'* $=$ *size-td s*
assumes *is-value*: *is-value-byte* (*export-uinfo s*) *i*
shows *access-ti s* (*update-ti s bs v*) *bs'* ! *i* $=$ *bs* ! *i*
⟨*proof*⟩

lemma *field-lookup-is-value-byte-access-ti-id*:
assumes *i-bound*: $i <$ *size-td s*
assumes *lbs*: *length bs* $=$ *size-td s*
assumes *is-value*: *is-value-byte* (*export-uinfo s*) *i*
shows *access-ti s v* (*bs*[*i* := *b*]) $=$ *access-ti s v bs*
⟨*proof*⟩

lemma *field-lookup-is-padding-byte-to-lense*:
assumes *is-padding*: *is-padding-byte* (*export-uinfo s*) *i*
shows *flense.is-padding-byte i*
⟨*proof*⟩

lemma *field-lookup-is-padding-byte-from-lense*:
assumes *is-padding*: *flense.is-padding-byte i*
shows *is-padding-byte* (*export-uinfo s*) *i*
⟨*proof*⟩

lemma *field-lookup-is-padding-byte-lense-conv*:
is-padding-byte (*export-uinfo s*) *i* ⟷ *flense.is-padding-byte i*
⟨*proof*⟩

lemma *field-lookup-is-value-byte-from-lense*:
assumes *is-value*: *flense.is-value-byte i*
shows *is-value-byte* (*export-uinfo s*) *i*
⟨*proof*⟩

lemma *field-lookup-is-value-byte-to-lense*:
assumes *is-value*: *is-value-byte* (*export-uinfo s*) *i*
shows *flense.is-value-byte i*
⟨*proof*⟩

lemma *field-lookup-is-value-byte-lense-conv*:
is-value-byte (*export-uinfo s*) *i* ⟷ *flense.is-value-byte i*
⟨*proof*⟩

lemma *field-lookup-eq-padding-lense-conv*: *eq-padding* (*export-uinfo s*) *bs bs′* ⟷ *flense.eq-padding bs bs′*
⟨*proof*⟩

lemma *field-lookup-eq-upto-padding-lense-conv*: *eq-upto-padding* (*export-uinfo s*) *bs bs′* ⟷ *flense.eq-upto-padding bs bs′*
⟨*proof*⟩

lemma *xmem-type-field-lookup-eq-padding-focus*:
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lbs′*: *length bs′* = *size-of TYPE*('*a*)
assumes *eq-padding*: *eq-padding* (*typ-uinfo-t TYPE*('*a*)) *bs bs′*
shows *eq-padding* (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs′*))
⟨*proof*⟩

lemma *xmem-type-field-lookup-eq-padding-focus-eq*:
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lbs′*: *length bs′* = *size-of TYPE*('*a*)
assumes *pfx-eq*: ⋀*i*. *i* < *n* ⟹ *bs* ! *i* = *bs′* ! *i*
assumes *sfx-eq*: ⋀*i*. *n* + *size-td s* ≤ *i* ⟹ *i* < *size-of TYPE*('*a*) ⟹ *bs* ! *i* = *bs′* ! *i*
shows *eq-padding* (*typ-uinfo-t TYPE*('*a*)) *bs bs′* =
eq-padding (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs′*))
⟨*proof*⟩

lemma *xmem-type-field-lookup-lense-eq-padding-focus*:
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lbs′*: *length bs′* = *size-of TYPE*('*a*)
assumes *eq-padding*: *lense.eq-padding bs bs′*

shows *flense.eq-padding* (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs'*))
⟨*proof*⟩

lemma *xmem-type-field-lookup-lense-eq-padding-focus-eq*:
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lbs'*: *length bs'* = *size-of TYPE*('*a*)
assumes *pfx-eq*: ⋀*i*. *i* < *n* ⟹ *bs* ! *i* = *bs'* ! *i*
assumes *sfx-eq*: ⋀*i*. *n* + *size-td s* ≤ *i* ⟹ *i* < *size-of TYPE*('*a*) ⟹ *bs* ! *i* = *bs'* ! *i*
shows *lense.eq-padding bs bs'* =
flense.eq-padding (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs'*))
⟨*proof*⟩

lemma *xmem-type-field-lookup-eq-upto-padding-focus*:
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lbs'*: *length bs'* = *size-of TYPE*('*a*)
assumes *eq-upto-padding*: *eq-upto-padding* (*typ-uinfo-t TYPE*('*a*)) *bs bs'*
shows *eq-upto-padding* (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs'*))
⟨*proof*⟩

lemma *xmem-type-field-lookup-eq-upto-padding-focus-eq*:
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lbs'*: *length bs'* = *size-of TYPE*('*a*)
assumes *pfx-eq*: ⋀*i*. *i* < *n* ⟹ *bs* ! *i* = *bs'* ! *i*
assumes *sfx-eq*: ⋀*i*. *n* + *size-td s* ≤ *i* ⟹ *i* < *size-of TYPE*('*a*) ⟹ *bs* ! *i* = *bs'* ! *i*
shows *eq-upto-padding* (*typ-uinfo-t TYPE*('*a*)) *bs bs'* =
eq-upto-padding (*export-uinfo s*) (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs'*))
⟨*proof*⟩

lemma *xmem-type-field-lookup-lense-eq-upto-padding-focus*:
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lbs'*: *length bs'* = *size-of TYPE*('*a*)
assumes *eq-upto-padding*: *lense.eq-upto-padding bs bs'*
shows *flense.eq-upto-padding* (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs'*))
⟨*proof*⟩

lemma *xmem-type-field-lookup-lense-eq-upto-padding-focus-eq*:
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lbs'*: *length bs'* = *size-of TYPE*('*a*)
assumes *pfx-eq*: ⋀*i*. *i* < *n* ⟹ *bs* ! *i* = *bs'* ! *i*
assumes *sfx-eq*: ⋀*i*. *n* + *size-td s* ≤ *i* ⟹ *i* < *size-of TYPE*('*a*) ⟹ *bs* ! *i* = *bs'* ! *i*
shows *lense.eq-upto-padding bs bs'* =
flense.eq-upto-padding (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs'*))

⟨*proof*⟩

lemma *xmem-type-field-lookup-eq-padding-super-update-bs*:
assumes *lxbs*: *length xbs* = *size-of TYPE*('*a*)
assumes *lbs*: *length bs* = *size-td s*
assumes *lbs'*: *length bs'* = *size-td s*
shows *eq-padding* (*typ-uinfo-t TYPE*('*a*)) (*super-update-bs bs xbs n*) (*super-update-bs bs' xbs n*) ⟷
eq-padding (*export-uinfo s*) *bs bs'*
⟨*proof*⟩

lemma *xmem-type-field-lookup-lense-eq-padding-super-update-bs*:
assumes *lxbs*: *length xbs* = *size-of TYPE*('*a*)
assumes *lbs*: *length bs* = *size-td s*
assumes *lbs'*: *length bs'* = *size-td s*
shows *lense.eq-padding* (*super-update-bs bs xbs n*) (*super-update-bs bs' xbs n*) ⟷
flense.eq-padding bs bs'
⟨*proof*⟩

lemma *xmem-type-field-lookup-eq-upto-padding-super-update-bs*:
assumes *lxbs*: *length xbs* = *size-of TYPE*('*a*)
assumes *lbs*: *length bs* = *size-td s*
assumes *lbs'*: *length bs'* = *size-td s*
shows *eq-upto-padding* (*typ-uinfo-t TYPE*('*a*)) (*super-update-bs bs xbs n*) (*super-update-bs bs' xbs n*) ⟷
eq-upto-padding (*export-uinfo s*) *bs bs'*
⟨*proof*⟩

lemma *xmem-type-field-lookup-lense-eq-upto-padding-super-update-bs*:
assumes *lxbs*: *length xbs* = *size-of TYPE*('*a*)
assumes *lbs*: *length bs* = *size-td s*
assumes *lbs'*: *length bs'* = *size-td s*
shows *lense.eq-upto-padding* (*super-update-bs bs xbs n*) (*super-update-bs bs' xbs n*) ⟷
flense.eq-upto-padding bs bs'
⟨*proof*⟩

lemma *access-ti-update-ti-lense-eq-upto-padding*:
assumes *lbs*: *length bs* = *size-td s*
assumes *lbs'*: *length bs'* = *size-td s*
shows *flense.eq-upto-padding* (*access-ti s* (*update-ti s bs v*) *bs'*) *bs*
⟨*proof*⟩

lemma *access-ti-update-ti-eq-upto-padding*:
assumes *lbs*: *length bs* = *size-td s*
assumes *lbs'*: *length bs'* = *size-td s*
shows *eq-upto-padding* (*export-uinfo s*) (*access-ti s* (*update-ti s bs v*) *bs'*) *bs*
⟨*proof*⟩

lemma *access-ti-update-ti-lense-eq-padding*:
assumes *flense.eq-padding bs bs'*
shows *access-ti s* (*update-ti s bs v*) *bs' = bs*
⟨*proof*⟩

lemma *access-ti-update-ti-eq-padding*:
assumes *eq-padding* (*export-uinfo s*) *bs bs'*
shows *access-ti s* (*update-ti s bs v*) *bs' = bs*
⟨*proof*⟩

lemma
assumes *match*: *export-uinfo s = typ-uinfo-t TYPE('b::xmem-type)*
assumes *lbs*: *length bs = size-td s*
shows *access-ti s v bs = to-bytes* ((*from-bytes* (*access-ti s v bs*))::*'b*) *bs*
⟨*proof*⟩

context
assumes *match*: *export-uinfo s = typ-uinfo-t TYPE('b::xmem-type)*
begin

lemma *field-lookup-lense-eq-padding-fieldtyp-conv*:
flense.eq-padding bs bs' = xmem-type-class.lense.eq-padding TYPE('b) bs bs'
⟨*proof*⟩

lemma *field-lookup-lense-eq-upto-padding-fieldtyp-conv*:
flense.eq-upto-padding bs bs' = xmem-type-class.lense.eq-upto-padding TYPE('b) bs bs'
⟨*proof*⟩

lemma *field-lookup-is-value-byte-fieldtyp-conv*:
flense.is-value-byte i = xmem-type-class.lense.is-value-byte TYPE('b) i
⟨*proof*⟩

lemma *field-lookup-is-padding-byte-fieldtyp-conv*:
flense.is-padding-byte i = xmem-type-class.lense.is-padding-byte TYPE('b) i
⟨*proof*⟩

lemma *field-lookup-access-ti-to-bytes-field-conv*:
assumes *eq-upto-padding*: *eq-upto-padding* (*export-uinfo s*) (*access-ti s v bs*) *vs*
assumes *eq-padding*: *eq-padding* (*export-uinfo s*) *bs bs'*
shows *access-ti s v bs = to-bytes* ((*from-bytes vs*)::*'b*) *bs'*
⟨*proof*⟩

lemma *field-lookup-access-ti-eq-upto-padding*:
length bs = size-td s ⟹
eq-upto-padding (*export-uinfo s*) (*access-ti s v bs*) (*access-ti s v bs'*)
⟨*proof*⟩

lemma *field-lookup-access-ti-eq-padding-value*:
length bs = size-td s ⟹ eq-padding (export-uinfo s) (access-ti s v bs) (access-ti s v′ bs)
⟨*proof*⟩

lemma *field-lookup-access-ti-eq-padding-bytes*:
length bs = size-td s ⟹ eq-padding (export-uinfo s) (access-ti s v bs) bs
⟨*proof*⟩

lemma *field-lookup-access-ti-to-bytes-field-conv′*:
assumes *eq-upto-padding*: *eq-upto-padding (export-uinfo s) (access-ti s v bs) vs*
assumes *lbs*: *length bs = size-td s*
shows *access-ti s v bs = to-bytes ((from-bytes vs)::′b) bs*
⟨*proof*⟩

lemma *field-lookup-update-ti-from-bytes-field-conv*:
fixes *v*::′*a* **and** *vf*::′*b*
assumes *lbs*: *length bs = size-td s*
assumes *lxbs*: *length xbs = size-of TYPE(′a)*
shows *update-ti (typ-info-t TYPE(′b)) bs vf =*
(from-bytes (access-ti s (update-ti s bs v) xbs))
⟨*proof*⟩

lemma *xmem-type-field-lookup-update-ti-super-update-bs-conv*:
fixes *v*::′*a*
assumes *lbs*: *length bs = size-td s*
assumes *lxbs*: *length xbs = size-of TYPE(′a)*
shows *update-ti s bs v =*
update-ti (typ-info-t TYPE(′a)) (super-update-bs bs (access-ti (typ-info-t TYPE(′a)) v xbs) n) v
⟨*proof*⟩

lemma *heap-update-list-field-to-root*:
fixes *p*::′*a ptr*
assumes *cgrd*: *c-guard p*
assumes *lbs*: *length bs = size-td s*
shows *heap-update-list (&(p→f)) bs hp =*
heap-update-list (ptr-val p) (super-update-bs bs (heap-list hp (size-of TYPE(′a)) (ptr-val p)) n) hp
⟨*proof*⟩

lemma *heap-list-field-to-root*:
fixes *p*::′*a ptr*
shows *heap-list hp (size-td s) &(p→f) =*
take (size-td s) ((drop n) (heap-list hp (size-of TYPE(′a)) (ptr-val (p::′a

ptr))))
⟨*proof*⟩

lemma *heap-list-field-super-update-bs-root-conv*:
fixes *p*::'*a ptr*
shows *super-update-bs* (*heap-list hp* (*size-td s*) (&(*p*→*f*))) (*heap-list hp* (*size-of TYPE*('*a*)) (*ptr-val p*)) *n* =
(*heap-list hp* (*size-of TYPE*('*a*)) (*ptr-val p*))
⟨*proof*⟩

lemma *heap-update-field-root-conv*:
fixes *p*::'*a ptr*
assumes *cgrd*: *c-guard p*
shows *heap-update* (*PTR*('*b*) &(*p*→*f*)) *v hp* =
heap-update p (*update-ti s* (*to-bytes v* (*heap-list hp* (*size-of TYPE*('*b*)) (&(*p*→*f*)))) (*h-val hp p*)) *hp*
⟨*proof*⟩

lemma *heap-update-padding-field-root-conv*:
fixes *p*::'*a ptr*
assumes *cgrd*: *c-guard p*
assumes *lbs*: *length bs* = *size-of TYPE* ('*b*)
shows *heap-update-padding* (*PTR*('*b*) &(*p*→*f*)) *v bs hp* =
heap-update-padding p (*update-ti s* (*to-bytes v bs*) (*h-val hp p*))
(*super-update-bs bs* (*heap-list hp* (*size-of TYPE*('*a*)) (*ptr-val p*)) *n*) *hp*
⟨*proof*⟩

lemma *heap-update-field-root-conv'*:
fixes *p*::'*a ptr*
assumes *cgrd*: *c-guard p*
shows *heap-update* (*PTR*('*b*) &(*p*→*f*)) *v hp* =
heap-update p (*update-ti s* (*to-bytes-p v*) (*h-val hp p*)) *hp*
⟨*proof*⟩

lemma *heap-update-padding-field-root-conv'*:
fixes *p*::'*a ptr*
assumes *cgrd*: *c-guard p*
assumes *lbs*: *length bs* = *size-of TYPE* ('*b*)
shows *heap-update-padding* (*PTR*('*b*) &(*p*→*f*)) *v bs hp* =
heap-update-padding p (*update-ti s* (*to-bytes-p v*) (*h-val hp p*))
(*super-update-bs bs* (*heap-list hp* (*size-of TYPE*('*a*)) (*ptr-val p*)) *n*) *hp*
⟨*proof*⟩

end
end

lemma *heap-update-field-root-conv''*:
fixes *p*::*'a ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*adjust-ti* (*typ-info-t TYPE*(*'b*::*xmem-type*)) *fld* (*fld-update* ∘ (λ*x* -. *x*)), *n*)
assumes *fg-cons*: *fg-cons fld* (*fld-update* ∘ (λ*x* -. *x*))
assumes *cgrd*: *c-guard p*
shows *heap-update* (*PTR*(*'b*) &(*p*→*f*)) *v hp* =
heap-update p (*fld-update* (λ-. *v*) (*h-val hp p*)) *hp*
⟨*proof*⟩

lemma *heap-update-field-root-conv-pointless'*:
fixes *p*::*'a ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*adjust-ti* (*typ-info-t TYPE*(*'b*::*xmem-type*)) *fld* (*fld-update* ∘ (λ*x* -. *x*)), *n*)
assumes *fg-cons*: *fg-cons fld* (*fld-update* ∘ (λ*x* -. *x*))
assumes *cgrd*: *c-guard p*
shows *heap-update* (*PTR*(*'b*) &(*p*→*f*)) *v* =
(λ*hp*. *heap-update p* (*fld-update* (λ-. *v*) (*h-val hp p*)) *hp*)
⟨*proof*⟩

lemma *heap-update-padding-field-root-conv''*:
fixes *p*::*'a ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*adjust-ti* (*typ-info-t TYPE*(*'b*::*xmem-type*)) *fld* (*fld-update* ∘ (λ*x* -. *x*)), *n*)
assumes *fg-cons*: *fg-cons fld* (*fld-update* ∘ (λ*x* -. *x*))
assumes *cgrd*: *c-guard p*
assumes *lbs*: *length bs* = *size-of TYPE* (*'b*)
shows *heap-update-padding* (*PTR*(*'b*) &(*p*→*f*)) *v bs hp* =
heap-update-padding p (*fld-update* (λ-. *v*) (*h-val hp p*))
(*super-update-bs bs* (*heap-list hp* (*size-of TYPE*(*'a*)) (*ptr-val p*)) *n*) *hp*
⟨*proof*⟩

lemma *heap-update-field-root-conv'''*:
fixes *p*::*'a ptr*
assumes *fl*: *field-ti TYPE*(*'a*) *f* = *Some s*
assumes *cgrd*: *c-guard p*
assumes *match*: *export-uinfo s* = *typ-uinfo-t TYPE*(*'b*::*xmem-type*)
shows *heap-update* (*PTR*(*'b*) &(*p*→*f*)) *v hp* =
heap-update p (*update-ti s* (*to-bytes-p v*) (*h-val hp p*)) *hp*
⟨*proof*⟩

lemma *heap-update-padding-field-root-conv'''*:
fixes *p*::*'a ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*s*, *n*)
assumes *cgrd*: *c-guard p*
assumes *match*: *export-uinfo s* = *typ-uinfo-t TYPE*(*'b*::*xmem-type*)
assumes *lbs*: *length bs* = *size-of TYPE* (*'b*)
shows *heap-update-padding* (*PTR*(*'b*) &(*p*→*f*)) *v bs hp* =

heap-update-padding p (update-ti s (to-bytes-p v) (h-val hp p))
(super-update-bs bs (heap-list hp (size-of TYPE('a)) (ptr-val p)) n) hp
⟨*proof*⟩

end

lemma *length-array-to-bytes*:
fixes *arr*::('*a*::*array-outer-max-size*['*b*::*array-max-count*])
shows *length* (*to-bytes arr* (*heap-list h* (*CARD*('*b*) ∗ *size-of TYPE*('*a*)) (*ptr-val p*))) =
size-of TYPE('*a*) ∗ *CARD*('*b*)
⟨*proof*⟩

lemma *take-drop-append-first*: *m* + *n* ≤ *length xs* ⟹ *take n* (*drop m* (*xs* @ *ys*)) = *take n* (*drop m xs*)
⟨*proof*⟩

lemma *size-td-list-array*:
size-td-list
(*map* (λ*n*. *DTuple*
(*adjust-ti* (*typ-info-t TYPE*('*a*::*xmem-type*)) (λ*x*. *x*.[*n*])
(λ*x f*. *Arrays.update f n x*))
(*replicate n CHR* ''*1*'')
(|*field-access* = *xto-bytes* ∘ (λ*x*. *x*.[*n*]),
field-update = (λ*x f*. *Arrays.update f n x*) ∘ *xfrom-bytes*,
field-sz = *size-of TYPE*('*a*)|))
[*0*..<*n*]) = *n* ∗ *size-of TYPE*('*a*)
⟨*proof*⟩

lemma *length-access-ti-list-array*:
fixes *arr*::('*a*::*array-outer-max-size*['*b*::*array-max-count*])
assumes *lxbs*: *length xbs* = *n* ∗ *size-of TYPE*('*a*)
shows
length (*access-ti-list*
(*map* (λ*n*. *DTuple*
(*adjust-ti* (*typ-info-t TYPE*('*a*)) (λ*x*. *x*.[*n*])
(λ*x f*. *Arrays.update f n x*))
(*replicate n CHR* ''*1*'')
(|*field-access* = *xto-bytes* ∘ (λ*x*. *x*.[*n*]),
field-update = (λ*x f*. *Arrays.update f n x*) ∘ *xfrom-bytes*,
field-sz = *size-of TYPE*('*a*)|))
[*0*..<*n*])
arr xbs) = *n* ∗ *size-of TYPE*('*a*)
⟨*proof*⟩

lemma *take-drop-take*: *m* + *k* ≤ *n* ⟹ *n* ≤ *length xbs* ⟹ *take m* (*drop k* (*take n xbs*)) = *take m* (*drop k xbs*)
⟨*proof*⟩

lemma *access-ti-array-index*:
fixes *arr*::('*a*::*array-outer-max-size*['*b*::*array-max-count*])
assumes *bound*: $i < n$
assumes *lxbs*: *length xbs* = n * *size-of TYPE*('*a*)
assumes *bs*: *bs* = *take* (*size-of TYPE*('*a*)) (*drop* (i * (*size-of TYPE*('*a*))) *xbs*)
shows
access-ti (*typ-info-t TYPE*('*a*)) (*arr*.[*i*])
(*take* (*size-of TYPE*('*a*)) (*drop* (i * *size-of TYPE*('*a*)) *xbs*)) =
take (*size-of TYPE*('*a*))
(*drop* (i * *size-of TYPE*('*a*)) (*access-ti* (*array-tag-n n*) *arr xbs*))
⟨*proof*⟩

lemma *access-ti-array-index'*:
fixes *arr*::('*a*::*array-outer-max-size*['*b*::*array-max-count*])
assumes *bound*: $i <$ *CARD*('*b*)
assumes *lbs*: *length xbs* = (*CARD*('*b*) * *size-of TYPE*('*a*))
assumes *bs*: *bs* = *take* (*size-of TYPE*('*a*)) (*drop* (i * (*size-of TYPE*('*a*))) *xbs*)
shows
access-ti (*typ-info-t TYPE*('*a*)) (*arr*.[*i*]) *bs* =
take (*size-of TYPE*('*a*))
(*drop* (i * *size-of TYPE*('*a*))
(*access-ti* (*typ-info-t TYPE*('*a*['*b*])) *arr xbs*))
⟨*proof*⟩

lemma *fold-index-shift*: *fold f* [*n*..<*n* + *m*] = *fold* (λ*i*. *f* (*n* + *i*)) [*0*..<*m*]
⟨*proof*⟩

lemma *fold-Suc-index-shift*: *fold f* [*1*..<*Suc n*] = *fold* (λ*i*. *f* (*Suc i*)) [*0*..<*n*]
⟨*proof*⟩

lemma *sum-list-index-shift*: *sum-list* (*map f* [*n*..<*n*+*m*]) = *sum-list* (*map* (λ*i*. *f* (*n* + *i*)) [*0*..<*m*])
⟨*proof*⟩

lemma *sum-list-Suc-index-shift*: *sum-list* (*map f* [*1*..<*Suc n*]) = *sum-list* (*map* (λ*i*. *f* (*Suc i*)) [*0*..<*n*])
⟨*proof*⟩

lemma *upt-Suc-snoc*: [*0*..<*Suc n*] = [*0*..<*n*] @ [*n*]
⟨*proof*⟩

lemma *sum-list-le-prefix*:
fixes *sz*:: *nat* ⇒ *nat*
assumes *lower*: $n \leq m$
assumes *le*: $m \leq k$

shows *sum-list* (*map sz* [*n*..<*m*]) ≤ *sum-list* (*map sz* [*n*..<*k*])
⟨*proof*⟩

lemma *intvl-off-disj'*:
fixes *x* :: *addr*
assumes *ylt*: *y* ≤ *off*
and *zoff*: *unat x* + *off* + *z* ≤ *addr-card*
shows {*x* ..+ *y*} ∩ {*x* + *of-nat off* ..+ *z*} = {}
⟨*proof*⟩

lemma *heap-update-list-padding-fold-partition*:
fixes *v*:: *nat* ⇒ *byte list* ⇒ *byte list*
and *sz*:: *nat* ⇒ *nat*
and *off*:: *nat* ⇒ *nat*
assumes *no-overflow*: *unat a* + *length bs* ≤ *addr-card*
assumes *partition*: *bs* = *concat* (*map* (λ*i*. *take* (*sz i*) (*drop* (*off i*) *bs*)) [*0*..<*m*])
assumes *lbs*: *length bs* = *sum-list* (*map sz* [*0*..<*m*])
assumes *partition-pbs*: *pbs* = *concat* (*map* (λ*i*. *take* (*sz i*) (*drop* (*off i*) *pbs*)) [*0*..<*m*])
assumes *lpbs*: *length pbs* = *sum-list* (*map sz* [*0*..<*m*])
assumes *off-sz*: ⋀*i*. *i* < *m* ⟹ *off i* = *sum-list* (*map sz* [*0*..<*i*])
assumes *val*: ⋀*i*. *i* < *m* ⟹
v i (*take* (*sz i*) (*drop* (*off i*) *pbs*)) =
(*take* (*sz i*) (*drop* (*off i*) *bs*))
shows
heap-update-list a bs h =
fold (λ*i h*. *heap-update-list*
(*a* + *word-of-nat* (*off i*))
(*v i* (*take* (*sz i*) (*drop* (*off i*) *pbs*)))
h)
[*0*..<*m*] *h*
⟨*proof*⟩

lemma *heap-update-list-fold-partition*:
fixes *v*:: *nat* ⇒ *byte list* ⇒ *byte list*
and *sz*:: *nat* ⇒ *nat*
and *off*:: *nat* ⇒ *nat*
assumes *no-overflow*: *unat a* + *length bs* ≤ *addr-card*
assumes *partition*: *bs* = *concat* (*map* (λ*i*. *take* (*sz i*) (*drop* (*off i*) *bs*)) [*0*..<*m*])
assumes *lbs*: *length bs* = *sum-list* (*map sz* [*0*..<*m*])
assumes *off-sz*: ⋀*i*. *i* < *m* ⟹ *off i* = *sum-list* (*map sz* [*0*..<*i*])
assumes *val*: ⋀*i*. *i* < *m* ⟹
v i (*take* (*sz i*) (*drop* (*off i*) (*heap-list h* (*length bs*) *a*))) =
(*take* (*sz i*) (*drop* (*off i*) *bs*))
shows
heap-update-list a bs h =
fold (λ*i h*. *heap-update-list*
(*a* + *word-of-nat* (*off i*))
(*v i* ((*heap-list h* (*sz i*) (*a* + *word-of-nat* (*off i*)))))

h)
$[0..<m]$ h
⟨*proof*⟩

lemma *heap-list-map-partition*:
fixes *sz*:: *nat* ⇒ *nat*
and *off*:: *nat* ⇒ *nat*
assumes *no-overflow*: *unat a* + *n* ≤ *addr-card*
assumes *lbs*: *n* = *sum-list* (*map sz* [*0..<m*])
assumes *off-sz*: ⋀*i*. *i* < *m* ⟹ *off i* = *sum-list* (*map sz* [*0..<i*])
shows
heap-list h n a =
concat (*map* (λ*i*. *heap-list h* (*sz i*) (*a* + *word-of-nat* (*off i*))) [*0..<m*])
⟨*proof*⟩

lemma *array-partition*:
assumes *lbs*: *length bs* = *m* ∗ *n*
shows *concat* (*map* (λ*i*. *take n* (*drop* (*i* ∗ *n*) *bs*)) [*0..<m*]) = *bs*
⟨*proof*⟩

lemma *sum-list-const-fun*: *sum-list* (*map* (λ-. *n*::*nat*) [*0..<m*]) = *m* ∗ *n*
⟨*proof*⟩

lemmas *export-uinfo-adjust-ti* [*simp*] = *export-tag-adjust-ti*(*1*)[*rule-format*]

lemma *heap-update-list-array*:
fixes *arr*:: (*'a*::*array-outer-max-size*[*'b*::*array-max-count*])
fixes *p*:: (*'a*[*'b*]) *ptr*
assumes *cgrd*: *c-guard p*
shows
heap-update-list (*ptr-val p*) (*to-bytes arr* (*heap-list h* (*size-of TYPE*(*'a*[*'b*])) (*ptr-val p*))) *h* =
fold
(λ*i h*. *heap-update-list* (*ptr-val* (*array-ptr-index p False i*))
(*to-bytes* (*arr*.[*i*])
(*heap-list h* (*size-of TYPE*(*'a*)) (*ptr-val* (*array-ptr-index p False i*))))
h)
[*0..<CARD*(*'b*)] *h*
⟨*proof*⟩

lemma *heap-update-array*:
fixes *arr*:: (*'a*::*array-outer-max-size*[*'b*::*array-max-count*])
fixes *p*:: (*'a*[*'b*]) *ptr*
assumes *cgrd*: *c-guard p*
shows
heap-update p arr h =
fold
(λ*i h*. *heap-update* (*array-ptr-index p False i*) (*arr*.[*i*]) *h*)

$[0..<CARD('b)]$ h
⟨*proof*⟩

lemma *heap-update-array-pointless*:
fixes *arr*:: ('*a*::*array-outer-max-size*['*b*::*array-max-count*])
fixes *p*:: ('*a*['*b*]) *ptr*
assumes *cgrd*: *c-guard p*
shows
heap-update p arr =
fold
(λ*i h*. *heap-update* (*array-ptr-index p False i*) (*arr*.[*i*]) *h*)
$[0..<CARD('b)]$
⟨*proof*⟩

lemma *heap-update-padding-list-array*:
fixes *arr*:: ('*a*::*array-outer-max-size*['*b*::*array-max-count*])
fixes *p*:: ('*a*['*b*]) *ptr*
assumes *cgrd*: *c-guard p*
assumes *lbs*: *length bs* = *CARD*('*b*) ∗ *size-of TYPE*('*a*)
shows
heap-update-list (*ptr-val p*) (*to-bytes arr bs*) *h* =
fold
(λ*i h*. *heap-update-list* (*ptr-val* (*array-ptr-index p False i*))
(*to-bytes* (*arr*.[*i*])
(*take* (*size-of TYPE*('*a*)) (*drop* (*i* ∗ *size-of TYPE*('*a*)) *bs*)))
h)
$[0..<CARD('b)]$ h
⟨*proof*⟩

lemma *heap-update-padding-array*:
fixes *arr*:: ('*a*::*array-outer-max-size*['*b*::*array-max-count*])
fixes *p*:: ('*a*['*b*]) *ptr*
assumes *cgrd*: *c-guard p*
assumes *lbs*: *length bs* = *CARD*('*b*) ∗ *size-of TYPE*('*a*)
shows
heap-update-padding p arr bs h =
fold
(λ*i h*. *heap-update-padding* (*array-ptr-index p False i*) (*arr*.[*i*]) (*take* (*size-of* *TYPE*('*a*)) (*drop* (*i* ∗ *size-of TYPE*('*a*)) *bs*)) *h*)
$[0..<CARD('b)]$ h
⟨*proof*⟩

lemma *heap-update-padding-array-pointless*:
fixes *arr*:: ('*a*::*array-outer-max-size*['*b*::*array-max-count*])
fixes *p*:: ('*a*['*b*]) *ptr*
assumes *cgrd*: *c-guard p*
assumes *lbs*: *length bs* = *CARD*('*b*) ∗ *size-of TYPE*('*a*)

shows
heap-update-padding p arr bs =
fold
(λ*i h*. *heap-update-padding* (*array-ptr-index p False i*) (*arr.*[*i*]) (*take* (*size-of TYPE*('*a*)) (*drop* (*i* ∗ *size-of TYPE*('*a*)) *bs*)) *h*)
[*0*..<*CARD*('*b*)]
⟨*proof*⟩

lemma *c-guard-array-ptr-index*:
fixes *p*:: (('*a* :: *mem-type*)['*b* :: *finite*]) *ptr*
assumes *cgrd*: *c-guard p*
assumes *bound*: *n* < *CARD*('*b*)
shows *c-guard* (*array-ptr-index p coerce n*)
⟨*proof*⟩

lemma *heap-update-array-update*:
assumes *n*: *n* < *CARD*('*b* :: *array-max-count*)
assumes *size*: *CARD*('*b*) ∗ *size-of TYPE*('*a* :: *array-outer-max-size*) < *2* ^ *addr-bitsize*
assumes *cgrd*: *c-guard p*
shows *heap-update p* (*Arrays.update* (*arr* :: '*a*['*b*]) *n v*) *hp*
= *heap-update* (*array-ptr-index p False n*) *v* (*heap-update p arr hp*)
⟨*proof*⟩

lemma *heap-update-array-element''*:
fixes *p'* :: (('*a* ::*array-outer-max-size*)['*b*::*array-max-count*]) *ptr*
fixes *p* :: ('*a* :: *array-outer-max-size*) *ptr*
fixes *hp w*
assumes *p*: *p* = *array-ptr-index p' False n*
assumes *n*: *n* < *CARD*('*b*)
assumes *cgrd*: *c-guard p'*
assumes *size*: *CARD*('*b*) ∗ *size-of TYPE*('*a*) < *2* ^ *addr-bitsize*
shows *heap-update p'* (*Arrays.update* (*h-val hp p'*) *n w*) *hp*
= *heap-update p w hp*
⟨*proof*⟩

lemmas *heap-update-array-element'*
= *heap-update-array-element''*[*simplified array-ptr-index-simps*]

lemma (**in** *array-outer-max-size*) *array-count-size*:
CARD('*b* :: *array-max-count*) ∗ *size-of TYPE*('*a*) < *2* ^ *addr-bitsize*
⟨*proof*⟩

lemmas *heap-update-array-element*
= *heap-update-array-element'*[*OF refl - - array-count-size*]

primrec
field-names-u :: *typ-uinfo* ⇒ *typ-uinfo* ⇒

(*qualified-field-name*) *list* **and**
field-names-struct-u :: *typ-uinfo-struct* ⇒ *typ-uinfo* ⇒
(*qualified-field-name*) *list* **and**
field-names-list-u :: *typ-uinfo-tuple list* ⇒ *typ-uinfo* ⇒
(*qualified-field-name*) *list* **and**
field-names-tuple-u :: *typ-uinfo-tuple* ⇒ *typ-uinfo* ⇒
(*qualified-field-name*) *list*
where
tufs0: *field-names-u* (*TypDesc algn st nm*) *t* = (*if t*= (*TypDesc algn st nm*) *then*
[[]] *else field-names-struct-u st t*)

| *tufs1*: *field-names-struct-u* (*TypScalar m algn d*) *t* = []
| *tufs2*: *field-names-struct-u* (*TypAggregate xs*) *t* = *field-names-list-u xs t*

| *tufs3*: *field-names-list-u* [] *t* = []
| *tufs4*: *field-names-list-u* (*x*#*xs*) *t* = *field-names-tuple-u x t*@*field-names-list-u xs t*

| *tufs5*: *field-names-tuple-u* (*DTuple s f d*) *t* = *map* (λ*fs*. *f*#*fs*) (*field-names-u s t*)

lemma *field-names-u-field-names-export-uinfo-conv*:
fixes *t* :: ('*a*, '*b*) *typ-info* **and**
st :: ('*a*, '*b*) *typ-info-struct* **and**
ts :: ('*a*, '*b*) *typ-info-tuple list* **and**
x :: ('*a*, '*b*) *typ-info-tuple*
shows
field-names-u (*export-uinfo t*) *s* = *field-names t s*
field-names-struct-u (*map-td-struct field-norm* (λ-. ()) *st*) *s* = *field-names-struct st s*
field-names-list-u (*map-td-list field-norm* (λ-. ()) *ts*) *s* = *field-names-list ts s*
field-names-tuple-u (*map-td-tuple field-norm* (λ-. ()) *x*) *s* = *field-names-tuple x s*
⟨*proof*⟩

primrec
all-field-names :: ('*a*, '*b*) *typ-desc* ⇒
(*qualified-field-name*) *list* **and**
all-field-names-struct :: ('*a*, '*b*) *typ-struct* ⇒
(*qualified-field-name*) *list* **and**
all-field-names-list :: (('*a*, '*b*) *typ-desc*, *field-name*, '*b*) *dt-tuple list* ⇒
(*qualified-field-name*) *list* **and**
all-field-names-tuple :: (('*a*, '*b*) *typ-desc*, *field-name*, '*b*) *dt-tuple* ⇒
(*qualified-field-name*) *list*
where
afs0: *all-field-names* (*TypDesc algn st nm*) =
[[]] @ *all-field-names-struct st*

| *afs1*: *all-field-names-struct* (*TypScalar m algn d*) = []

| *afs2*: *all-field-names-struct* (*TypAggregate xs*) = *all-field-names-list xs*

| *afs3*: *all-field-names-list* [] = []
| *afs4*: *all-field-names-list* (*x*#*xs*) = *all-field-names-tuple x* @ *all-field-names-list xs*

| *afs5*: *all-field-names-tuple* (*DTuple s f d*) = *map* (λ*fs*. *f*#*fs*) (*all-field-names s*)

lemma *field-lookup-all-field-names*:
  fixes *t*::('*a*, '*b*) *typ-desc*
  and *st*::('*a*, '*b*) *typ-struct*
  and *ts*::('*a*, '*b*) *typ-tuple list*
  and *x*::('*a*, '*b*) *typ-tuple*
shows
  field-lookup t f m = *Some* (*s*, *n*) ⟹ *f* ∈ *set* (*all-field-names t*) **and**
  field-lookup-struct st f m = *Some* (*s*, *n*) ⟹ *f* ∈ *set* (*all-field-names-struct st*)
and
  field-lookup-list ts f m = *Some* (*s*, *n*) ⟹ *f* ∈ *set* (*all-field-names-list ts*) **and**
  field-lookup-tuple x f m = *Some* (*s*, *n*) ⟹ *f* ∈ *set* (*all-field-names-tuple x*)
⟨*proof*⟩

lemma *field-names-subset-all-field-names*:
  fixes *t* :: *typ-uinfo* **and**
        st :: *typ-uinfo-struct* **and**
        ts :: *typ-uinfo-tuple list* **and**
        x :: *typ-uinfo-tuple*
      shows
 set (*field-names-u t s*) ⊆ *set* (*all-field-names t*)
 set (*field-names-struct-u st s*) ⊆ *set* (*all-field-names-struct st*)
 set (*field-names-list-u ts s*) ⊆ *set* (*all-field-names-list ts*)
 set (*field-names-tuple-u x s*) ⊆ *set* (*all-field-names-tuple x*)
    ⟨*proof*⟩

lemma *empty-all-field-names*:
  fixes *t* :: *typ-uinfo* **and**
        st :: *typ-uinfo-struct* **and**
        ts :: *typ-uinfo-tuple list* **and**
        x :: *typ-uinfo-tuple*
      shows
 [] ∈ *set* (*all-field-names t*)
 [] ∉ *set* (*all-field-names-struct st*)
 [] ∉ *set* (*all-field-names-list ts*)
 [] ∉ *set* (*all-field-names-tuple x*)
  ⟨*proof*⟩

lemma *empty-field-names-u*:
  fixes *t* :: *typ-uinfo* **and**
        st :: *typ-uinfo-struct* **and**

ts :: *typ-uinfo-tuple list* **and**
x :: *typ-uinfo-tuple*
shows
True
[] ∉ *set* (*field-names-struct-u st s*)
[] ∉ *set* (*field-names-list-u ts s*)
[] ∉ *set* (*field-names-tuple-u x s*)
⟨*proof*⟩

lemma *non-empty-field-names-u*:
fixes *t* :: *typ-uinfo* **and**
st :: *typ-uinfo-struct* **and**
ts :: *typ-uinfo-tuple list* **and**
x :: *typ-uinfo-tuple*
shows
field-names-u t s ≠ [] ⟹ ∃ *n*. (*s*, *n*) ∈ *td-set t m*
(*field-names-struct-u st s*) ≠ [] ⟹ ∃ *n*. (*s*, *n*) ∈ *td-set-struct st m*
(*field-names-list-u ts s*) ≠ [] ⟹ ∃ *n*. (*s*, *n*) ∈ *td-set-list ts m*
(*field-names-tuple-u x s*) ≠ [] ⟹ ∃ *n*. (*s*, *n*) ∈ *td-set-tuple x m*
⟨*proof*⟩

lemma *td-set-size*:
(*s*, *n*) ∈ *td-set t m* ⟹ *size s* ≤ *size t*
(*s*, *n*) ∈ *td-set-struct st m* ⟹ *size s* ≤ *size st*
(*s*, *n*) ∈ *td-set-list ts m* ⟹ ∃ *t* ∈ *set ts*. *size s* ≤ *size* (*dt-fst t*)
(*s*, *n*) ∈ *td-set-tuple x m* ⟹ *size s* ≤ *size* (*dt-fst x*)
⟨*proof*⟩

lemma *td-set-field-names-nonempty*:
(*s*, *n*) ∈ *td-set t m* ⟹ *field-names t* (*export-uinfo s*) ≠ []
(*s*, *n*) ∈ *td-set-struct st m* ⟹ *field-names-struct st* (*export-uinfo s*) ≠ []
(*s*, *n*) ∈ *td-set-list ts m* ⟹ *field-names-list ts* (*export-uinfo s*) ≠ []
(*s*, *n*) ∈ *td-set-tuple x m* ⟹ *field-names-tuple x* (*export-uinfo s*) ≠ []
⟨*proof*⟩

lemma *sub-typ-field-names-nonempty*:
assumes *s-t*: *s* ≤ *t*
shows *field-names t* (*export-uinfo s*) ≠ []
⟨*proof*⟩

lemma *sub-typ-export-uinfo-mono*:
assumes *s-t*: *s* ≤ *t*
shows *export-uinfo s* ≤ *export-uinfo t*
⟨*proof*⟩

lemma *descriptor-not-in-self*: (*TypDesc algn st nm*, *n*) ∉ *td-set-struct st m*
⟨*proof*⟩

lemma *field-names-struct-descriptor-empty*: *field-names-struct-u st* (*TypDesc algn st nm*) = []
⟨*proof*⟩

lemma *all-field-names-exists-field-names-u*:
fixes *t* :: *typ-uinfo* **and**
st :: *typ-uinfo-struct* **and**
ts :: *typ-uinfo-tuple list* **and**
x :: *typ-uinfo-tuple*
shows
f ∈ *set* (*all-field-names t*) ⟹ ∃ *s*. *f* ∈ *set* (*field-names-u t s*)
f ∈ *set* (*all-field-names-struct st*) ⟹ ∃ *s*. *f* ∈ *set* (*field-names-struct-u st s*)
f ∈ *set* (*all-field-names-list ts*) ⟹ ∃ *s*. *f* ∈ *set* (*field-names-list-u ts s*)
f ∈ *set* (*all-field-names-tuple x*) ⟹ ∃ *s*. *f* ∈ *set* (*field-names-tuple-u x s*)
⟨*proof*⟩

theorem *all-field-names-union-field-names-u-conv*: *set* (*all-field-names t*) = (⋃ *s*. *set* (*field-names-u t s*))
⟨*proof*⟩

corollary *all-field-names-union-field-names-export-uinfo-conv*:
set (*all-field-names* (*export-uinfo t*)) = (⋃ *s*. *set* (*field-names t s*))
⟨*proof*⟩

lemma *filter-same-eq*: (⋀*x*. *x* ∈ *set xs* ⟹ *P x* = *Q x*) ⟹ *filter P xs* = *filter Q xs*
⟨*proof*⟩

lemma *field-lookup-tuple-hd-notin*: *n* ≠ *dt-snd x* ⟹ *field-lookup-tuple x* (*n* # *ns*) *m* = *None*
⟨*proof*⟩

lemma *field-lookup-list-hd-notin*: *n* ∉ *dt-snd* ‘ *set xs* ⟹ *field-lookup-list xs* (*n* # *ns*) *m* = *None*
⟨*proof*⟩

lemma *list-append-eq-split*: *xs1* = *xs2* ⟹ *ys1* = *ys2* ⟹ (*xs1* @ *ys1*) = (*xs2* @ *ys2*)
⟨*proof*⟩

lemma *field-names-u-filter-all-field-names-conv*:
fixes *t* :: *typ-uinfo* **and**
st :: *typ-uinfo-struct* **and**
ts :: *typ-uinfo-tuple list* **and**
x :: *typ-uinfo-tuple*
shows
wf-desc t ⟹
(*field-names-u t s*) = *filter* (λ*f*. ∃ *n*. *field-lookup t f 0* = *Some* (*s*, *n*)) (*all-field-names*

t)
wf-desc-struct st ⟹
field-names-struct-u st s = *filter* (λ*f*. ∃ *n*. *field-lookup-struct st f 0* = *Some* (*s*, *n*)) (*all-field-names-struct st*)
wf-desc-list ts ⟹
field-names-list-u ts s = *filter* (λ*f*. ∃ *n*. *field-lookup-list ts f 0* = *Some* (*s*, *n*)) (*all-field-names-list ts*)
wf-desc-tuple x ⟹
field-names-tuple-u x s = *filter* (λ*f*. ∃ *n*. *field-lookup-tuple x f 0* = *Some* (*s*, *n*)) (*all-field-names-tuple x*)
⟨*proof*⟩

lemma *all-field-names-export-uinfo'*:
fixes *t*:: ('*a*, '*b*) *typ-info*
and *st*:: ('*a*, '*b*) *typ-info-struct*
and *ts*::('*a*, '*b*) *typ-info-tuple list*
and *x*::('*a*, '*b*) *typ-info-tuple*
shows
all-field-names (*map-td field-norm* (λ-. ()) *t*) = *all-field-names t*
all-field-names-struct (*map-td-struct field-norm* (λ-. ()) *st*) = *all-field-names-struct st*
all-field-names-list (*map-td-list field-norm* (λ-. ()) *ts*) = *all-field-names-list ts*
all-field-names-tuple (*map-td-tuple field-norm* (λ-. ()) *x*) = *all-field-names-tuple x*
⟨*proof*⟩

lemma *all-field-names-export-uinfo*:
all-field-names (*export-uinfo t*) = *all-field-names t*
⟨*proof*⟩

lemma *inj* (#)
⟨*proof*⟩

lemma *distinct-map-cons*: *distinct xs* ⟹ *distinct* (*map* ((#) *ys*) *xs*)
⟨*proof*⟩

lemma *all-field-names-list-conv*: *all-field-names-list xs* =
concat (*map* (λ*x*. *map* ((#) (*dt-snd x*)) ((*all-field-names o dt-fst*) *x*)) *xs*)
⟨*proof*⟩

lemma *distinct-all-field-names*:
fixes *t* :: ('*a*, '*b*) *typ-info* **and**
st :: ('*a*, '*b*) *typ-info-struct* **and**
ts :: ('*a*, '*b*) *typ-info-tuple list* **and**
x :: ('*a*, '*b*) *typ-info-tuple*
shows
wf-desc t ⟹ *distinct* (*all-field-names t*) **and**
wf-desc-struct st ⟹ *distinct* (*all-field-names-struct st*) **and**

wf-desc-list ts $\Longrightarrow$ *distinct* (*all-field-names-list ts*) **and**
wf-desc-tuple x $\Longrightarrow$ *distinct* (*all-field-names-tuple x*)
⟨*proof*⟩

lemma *td-set-field-names-u-nonempty*:
(*s*, *n*) ∈ *td-set t m* $\Longrightarrow$ *field-names-u t s* ≠ []
(*s*, *n*) ∈ *td-set-struct st m* $\Longrightarrow$ *field-names-struct-u st s* ≠ []
(*s*, *n*) ∈ *td-set-list ts m* $\Longrightarrow$ *field-names-list-u ts s* ≠ []
(*s*, *n*) ∈ *td-set-tuple x m* $\Longrightarrow$ *field-names-tuple-u x s* ≠ []
⟨*proof*⟩

lemma *field-lookup-export-uinfo-Some-rev*:
field-lookup (*export-uinfo t*) *f n* = *Some* (*s*, *k*) $\Longrightarrow$ ∃ *s'*. *field-lookup t f n* = *Some* (*s'*, *k*) ∧ *s* = *export-uinfo s'*
⟨*proof*⟩

lemma *wf-desc-export-uinfo-pres*:
fixes *t* :: ('*a*, '*b*) *typ-info* **and**
st :: ('*a*, '*b*) *typ-info-struct* **and**
ts :: ('*a*, '*b*) *typ-info-tuple list* **and**
x :: ('*a*, '*b*) *typ-info-tuple*
shows
wf-desc t $\Longrightarrow$ *wf-desc* (*export-uinfo t*)
wf-desc-struct st $\Longrightarrow$ *wf-desc-struct* (*map-td-struct field-norm* (λ-. ()) *st*)
wf-desc-list ts $\Longrightarrow$ *wf-desc-list* (*map-td-list field-norm* (λ-. ()) *ts*)
wf-desc-tuple x $\Longrightarrow$ *wf-desc-tuple* (*map-td-tuple field-norm* (λ-. ()) *x*)
⟨*proof*⟩

primrec
toplevel-field-names :: ('*a*, '*b*) *typ-desc* ⇒
(*field-name*) *list* **and**
toplevel-field-names-struct :: ('*a*, '*b*) *typ-struct* ⇒
(*field-name*) *list* **and**
toplevel-field-names-list :: ('*a*, '*b*) *typ-tuple list* ⇒
(*field-name*) *list* **and**
toplevel-field-names-tuple :: ('*a*, '*b*) *typ-tuple* ⇒
(*field-name*) *list*
where
toplevel-field-names (*TypDesc align st nm*) = *toplevel-field-names-struct st*

| *toplevel-field-names-struct* (*TypScalar m align d*) = []
| *toplevel-field-names-struct* (*TypAggregate xs*) = *toplevel-field-names-list xs*

| *toplevel-field-names-list* [] = []
| *toplevel-field-names-list* (*x*#*xs*) = *toplevel-field-names-tuple x* @ *toplevel-field-names-list xs*

| *toplevel-field-names-tuple* (*DTuple s f d*) = [*f*]

lemma *all-field-names-root*: $\exists\, xs.\ all\text{-}field\text{-}names\ t = [[]]\ @\ xs$
⟨*proof*⟩

lemma *toplevel-field-names-all-field-names*:
fixes *t*:: ('*a*, '*b*) *typ-desc*
and *st*:: ('*a*, '*b*) *typ-struct*
and *ts*::('*a*, '*b*) *typ-tuple list*
and *x*::('*a*, '*b*) *typ-tuple*
shows
$f \in set\ (toplevel\text{-}field\text{-}names\ t) \Longrightarrow [f] \in set\ (all\text{-}field\text{-}names\ t)$
$f \in set\ (toplevel\text{-}field\text{-}names\text{-}struct\ st) \Longrightarrow [f] \in set\ (all\text{-}field\text{-}names\text{-}struct\ st)$
$f \in set\ (toplevel\text{-}field\text{-}names\text{-}list\ ts) \Longrightarrow [f] \in set\ (all\text{-}field\text{-}names\text{-}list\ ts)$
$f \in set\ (toplevel\text{-}field\text{-}names\text{-}tuple\ x) \Longrightarrow [f] \in set\ (all\text{-}field\text{-}names\text{-}tuple\ x)$
⟨*proof*⟩

lemma *append-eq-same-prefixI*: $ys = zs \Longrightarrow xs\ @\ ys = xs\ @\ zs$
⟨*proof*⟩

lemma *toplevel-field-names-field-lookup*:
fixes *t*:: ('*a*, '*b*) *typ-info*
and *st*:: ('*a*, '*b*) *typ-info-struct*
and *ts*::('*a*, '*b*) *typ-info-tuple list*
and *x*::('*a*, '*b*) *typ-info-tuple*
shows
$f \in set\ (toplevel\text{-}field\text{-}names\ t) \Longrightarrow wf\text{-}desc\ t \Longrightarrow$
$\exists\, s\ n.\ field\text{-}lookup\ t\ [f]\ m = Some\ (s,\ m + n)$

$f \in set\ (toplevel\text{-}field\text{-}names\text{-}struct\ st) \Longrightarrow wf\text{-}desc\text{-}struct\ st \Longrightarrow$
$\exists\, s\ n.\ field\text{-}lookup\text{-}struct\ st\ [f]\ m = Some\ (s,\ m + n)$

$f \in set\ (toplevel\text{-}field\text{-}names\text{-}list\ ts) \Longrightarrow wf\text{-}desc\text{-}list\ ts \Longrightarrow$
$\exists\, s\ n.\ field\text{-}lookup\text{-}list\ ts\ [f]\ m = Some\ (s,\ m + n)$

$f \in set\ (toplevel\text{-}field\text{-}names\text{-}tuple\ x) \Longrightarrow wf\text{-}desc\text{-}tuple\ x \Longrightarrow$
$\exists\, s\ n.\ field\text{-}lookup\text{-}tuple\ x\ [f]\ m = Some\ (s,\ m + n)$
⟨*proof*⟩

lemma *partition-toplevel-field-names*:
fixes *t*:: ('*a*, '*b*) *typ-info*
and *st*:: ('*a*, '*b*) *typ-info-struct*
and *ts*::('*a*, '*b*) *typ-info-tuple list*
and *x*::('*a*, '*b*) *typ-info-tuple*
shows
$length\ bs = size\text{-}td\ t \Longrightarrow aggregate\ t \Longrightarrow wf\text{-}desc\ t \Longrightarrow$
$concat\ (map\ (\lambda f.\ take\ (size\text{-}td\ (fst\ (the\ (field\text{-}lookup\ t\ [f]\ n))))$
$(drop\ (snd\ (the\ (field\text{-}lookup\ t\ [f]\ n))) - n)\ bs))$
$(toplevel\text{-}field\text{-}names\ t)) = bs$

length bs = size-td-struct st ⟹ aggregate-struct st ⟹ wf-desc-struct st ⟹
concat (map (λf. take (size-td (fst (the (field-lookup-struct st [f] n))))
(drop (snd (the (field-lookup-struct st [f] n)) − n) bs))
(toplevel-field-names-struct st)) = bs

length bs = size-td-list ts ⟹ wf-desc-list ts ⟹
concat (map (λf. take (size-td (fst (the (field-lookup-list ts [f] n))))
(drop (snd (the (field-lookup-list ts [f] n)) − n) bs))
(toplevel-field-names-list ts)) = bs

length bs = size-td-tuple x ⟹ wf-desc-tuple x ⟹
concat (map (λf. take (size-td (fst (the (field-lookup-tuple x [f] n))))
(drop (snd (the (field-lookup-tuple x [f] n)) − n) bs))
(toplevel-field-names-tuple x)) = bs
⟨*proof*⟩

lemma *toplevel-field-names-export-uinfo'*:
fixes *t*:: (*'a*, *'b*) *typ-info*
and *st*:: (*'a*, *'b*) *typ-info-struct*
and *ts*::(*'a*, *'b*) *typ-info-tuple list*
and *x*::(*'a*, *'b*) *typ-info-tuple*
shows
toplevel-field-names (*map-td field-norm* (λ-. ()) *t*) = *toplevel-field-names t*
toplevel-field-names-struct (*map-td-struct field-norm* (λ-. ()) *st*) = *toplevel-field-names-struct st*
toplevel-field-names-list (*map-td-list field-norm* (λ-. ()) *ts*) = *toplevel-field-names-list ts*
toplevel-field-names-tuple (*map-td-tuple field-norm* (λ-. ()) *x*) = *toplevel-field-names-tuple x*
⟨*proof*⟩

lemma *toplevel-field-names-export-uinfo*:
toplevel-field-names (*export-uinfo t*) = *toplevel-field-names t*
⟨*proof*⟩

lemma (**in** *xmem-type*) *xmem-type-partition-toplevel-field-names*:
assumes *aggregate*: *aggregate* (*typ-info-t TYPE*(*'a*))
assumes *lbs*: *length bs* = *size-of* (*TYPE*(*'a*))
shows *concat* (*map* (λ*f*. *take* (*size-td* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*(*'a*))
[*f*] *0*))))
(*drop* (*snd* (*the* (*field-lookup* (*typ-info-t TYPE*(*'a*)) [*f*] *0*))) *bs*))
(*toplevel-field-names* (*typ-info-t TYPE*(*'a*)))) = *bs*
⟨*proof*⟩

lemma *toplevel-field-names-sum-list-size*:
fixes *t*:: (*'a*, *'b*) *typ-info*
and *st*:: (*'a*, *'b*) *typ-info-struct*

and *ts*::(*'a*, *'b*) *typ-info-tuple list*
and *x*::(*'a*, *'b*) *typ-info-tuple*
shows
aggregate t ⟹ *wf-desc t* ⟹
sum-list (*map* (λ*f*. *size-td* (*fst* (*the* (*field-lookup t* [*f*] *n*)))) (*toplevel-field-names t*)) =
size-td t

aggregate-struct st ⟹ *wf-desc-struct st* ⟹
sum-list (*map* (λ*f*. *size-td* (*fst* (*the* (*field-lookup-struct st* [*f*] *n*)))) (*toplevel-field-names-struct st*)) =
size-td-struct st

wf-desc-list ts ⟹
sum-list (*map* (λ*f*. *size-td* (*fst* (*the* (*field-lookup-list ts* [*f*] *n*)))) (*toplevel-field-names-list ts*)) =
size-td-list ts

wf-desc t ⟹
sum-list (*map* (λ*f*. *size-td* (*fst* (*the* (*field-lookup-tuple x* [*f*] *n*)))) (*toplevel-field-names-tuple x*)) =
size-td-tuple x
⟨*proof*⟩

lemma *toplevel-field-names-sum-list-offset*:
fixes *t*:: (*'a*, *'b*) *typ-info*
and *st*:: (*'a*, *'b*) *typ-info-struct*
and *ts*::(*'a*, *'b*) *typ-info-tuple list*
and *x*::(*'a*, *'b*) *typ-info-tuple*
shows
aggregate t ⟹ *wf-desc t* ⟹ *i* < *length* (*toplevel-field-names t*) ⟹
sum-list (*map* (λ*i*. *size-td* (*fst* (*the* (*field-lookup t* [(*toplevel-field-names t*) ! *i*] *n*)))) [*0*..<*i*]) =
(*snd* (*the* (*field-lookup t* [(*toplevel-field-names t*) ! *i*] *n*)) − *n*)

aggregate-struct st ⟹ *wf-desc-struct st* ⟹ *i* < *length* (*toplevel-field-names-struct st*) ⟹
sum-list (*map* (λ*i*. *size-td* (*fst* (*the* (*field-lookup-struct st* [(*toplevel-field-names-struct st*) ! *i*] *n*)))) [*0*..<*i*]) =
(*snd* (*the* (*field-lookup-struct st* [(*toplevel-field-names-struct st*) ! *i*] *n*)) − *n*)

wf-desc-list ts ⟹ *i* < *length* (*toplevel-field-names-list ts*) ⟹
sum-list (*map* (λ*i*. *size-td* (*fst* (*the* (*field-lookup-list ts* [(*toplevel-field-names-list ts*) ! *i*] *n*)))) [*0*..<*i*]) =
(*snd* (*the* (*field-lookup-list ts* [(*toplevel-field-names-list ts*) ! *i*] *n*)) − *n*)

wf-desc-tuple x ⟹ *i* < *length* (*toplevel-field-names-tuple x*) ⟹
sum-list (*map* (λ*i*. *size-td* (*fst* (*the* (*field-lookup-tuple x* [(*toplevel-field-names-tuple*

$x)\ !\ i]\ n)))))\ [0..<i]) =$
$(snd\ (the\ (field\text{-}lookup\text{-}tuple\ x\ [(toplevel\text{-}field\text{-}names\text{-}tuple\ x)\ !\ i]\ n))\ -\ n)$

⟨*proof*⟩

lemma *sum-list-upt-map-nth-conv*: *sum-list* (*map* (λ*i*. *g* (*xs* ! *i*)) [*0..<length xs*])
= *sum-list* (*map g xs*)
⟨*proof*⟩

lemma *toplevel-field-names-empty-typ-info*: *toplevel-field-names* (*empty-typ-info algn tn*) = []
⟨*proof*⟩

lemma *toplevel-field-names-no-padding-empty-typ-info*:
filter (*Not o padding-field-name*) (*toplevel-field-names* (*empty-typ-info algn tn*))
= []
⟨*proof*⟩

lemma *toplevel-field-names-list-append* [*simp*]:
toplevel-field-names-list (*xs* @ *ys*) = *toplevel-field-names-list xs* @ *toplevel-field-names-list ys*
⟨*proof*⟩

lemma *toplevel-field-names-extend-ti*:
fixes
t :: *'a xtyp-info* **and**
st :: *'a xtyp-info-struct* **and**
ts :: *'a xtyp-info-tuple list* **and**
x :: *'a xtyp-info-tuple*
shows
toplevel-field-names (*extend-ti t s n fn d*) = *toplevel-field-names t* @ [*fn*]
toplevel-field-names-struct (*extend-ti-struct st s fn d*) = *toplevel-field-names-struct st* @ [*fn*]
toplevel-field-names-list ts = *toplevel-field-names-list ts*
toplevel-field-names-tuple x = *toplevel-field-names-tuple x*
⟨*proof*⟩

lemma *toplevel-field-names-adjust-ti'*:
fixes
t :: *'a xtyp-info* **and**
st :: *'a xtyp-info-struct* **and**
ts :: *'a xtyp-info-tuple list* **and**
x :: *'a xtyp-info-tuple*
shows
toplevel-field-names (*map-td* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *t*) =
toplevel-field-names t
toplevel-field-names-struct (*map-td-struct* (λ*n algn*. *update-desc acc upd*) (*update-desc*

acc upd) *st*) =
  toplevel-field-names-struct st
toplevel-field-names-list (*map-td-list* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *ts*) =
  toplevel-field-names-list ts
toplevel-field-names-tuple (*map-td-tuple* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *x*) =
  toplevel-field-names-tuple x
⟨*proof*⟩

lemma *toplevel-field-names-adjust-ti*:
  toplevel-field-names (*adjust-ti t acc upd*) = *toplevel-field-names t*
  ⟨*proof*⟩

lemma *padding-field-name-pad*: *padding-field-name* (*foldl* (@) ''!*pad-*'' *xs*)
  ⟨*proof*⟩

lemma *toplevel-field-names-no-padding-ti-pad-combine*:
  filter (*Not o padding-field-name*) (*toplevel-field-names* (*ti-pad-combine n t*)) =
      filter (*Not o padding-field-name*) (*toplevel-field-names t*)
  ⟨*proof*⟩

lemma *toplevel-field-names-ti-pad-combine*:
  (*toplevel-field-names* (*ti-pad-combine n t*)) =
     toplevel-field-names t @ [*foldl* (@) ''!*pad-*'' (*CompoundCTypes.field-names-list t*)]
  ⟨*proof*⟩

lemma *toplevel-field-names-ti-typ-combine*:
  toplevel-field-names (*ti-typ-combine t-b acc upd algn fn t*) = *toplevel-field-names t* @ [*fn*]
  ⟨*proof*⟩

lemma *toplevel-field-names-no-padding-ti-typ-combine*:
  ¬ *padding-field-name fn* ⟹
  filter (*Not o padding-field-name*) (*toplevel-field-names* (*ti-typ-combine t-b acc upd algn fn t*)) =
    filter (*Not o padding-field-name*) (*toplevel-field-names t*) @ [*fn*]
  ⟨*proof*⟩

lemma *toplevel-field-names-no-padding-ti-typ-pad-combine*:
  ¬ *padding-field-name fn* ⟹
  filter (*Not o padding-field-name*) (*toplevel-field-names* (*ti-typ-pad-combine t-b acc upd algn fn t*)) =
      filter (*Not o padding-field-name*) (*toplevel-field-names t*) @ [*fn*]
  ⟨*proof*⟩

lemma *toplevel-field-names-ti-typ-pad-combine*:
toplevel-field-names (*ti-typ-pad-combine* (*t-b*:: *'b itself*) *acc upd algn fn t*) =
toplevel-field-names t @ (
if 0 < padup (*max* (*2* ^ *algn*) (*align-of TYPE*('*b*::*c-type*))) (*size-td t*) *then*
[*foldl* (@) *''!pad-''* (*CompoundCTypes.field-names-list t*), *fn*]
else
[*fn*])
⟨*proof*⟩

lemma *toplevel-field-names-map-align*: *toplevel-field-names* (*map-align n t*) = *toplevel-field-names t*
⟨*proof*⟩

lemma *toplevel-field-names-no-padding-final-pad*:
filter (*Not o padding-field-name*) (*toplevel-field-names* (*final-pad n t*))
= *filter* (*Not o padding-field-name*) (*toplevel-field-names t*)
⟨*proof*⟩

lemma *toplevel-field-names-final-pad*:
(*toplevel-field-names* (*final-pad n t*))
=
toplevel-field-names t @ (
if 0 < padup (*2* ^ *max n* (*align-td t*)) (*size-td t*) *then*
[*foldl* (@) *''!pad-''* (*CompoundCTypes.field-names-list t*)]
else [])
⟨*proof*⟩

lemmas *toplevel-field-names-no-padding-combinator-simps* =
toplevel-field-names-no-padding-empty-typ-info
toplevel-field-names-no-padding-final-pad
toplevel-field-names-no-padding-ti-typ-pad-combine
toplevel-field-names-no-padding-ti-typ-combine

lemmas *toplevel-field-names-combinator-simps* =
toplevel-field-names-empty-typ-info
toplevel-field-names-final-pad
toplevel-field-names-ti-typ-pad-combine
toplevel-field-names-ti-typ-combine

lemma *fold-filter-out-id*:
assumes *filter-out-id*: ⋀*i v*. *i < length xs* ⟹ ¬ *P* (*xs* ! *i*) ⟹ *f* (*xs*!*i*) *v* = *v*
shows*fold f xs* = *fold f* (*filter P xs*)
⟨*proof*⟩

context *xmem-type*
begin

lemma *xmem-type-toplevel-field-names-sum-list-size*:

assumes *aggregate*: *aggregate* (*typ-info-t TYPE*('*a*))
shows *sum-list*
(*map* (λ*f*. *size-td* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) [*f*] *n*))))
(*toplevel-field-names* (*typ-info-t TYPE*('*a*)))) =
size-of TYPE('*a*)
⟨*proof*⟩

lemma *xmem-type-toplevel-field-names-sum-list-offset*:
assumes *aggregate*: *aggregate* (*typ-info-t TYPE*('*a*))
assumes *i-bound*: *i* < *length* (*toplevel-field-names* (*typ-info-t TYPE*('*a*)))
shows
sum-list (*map* (λ*i*. *size-td* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*))
[(*toplevel-field-names* (*typ-info-t TYPE*('*a*))) ! *i*] *0*))))
[*0*..<*i*]) =
(*snd* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) [(*toplevel-field-names* (*typ-info-t* *TYPE*('*a*))) ! *i*] *0*)))
⟨*proof*⟩

lemma *xmem-type-toplevel-field-names-field-lookup*:
assumes *f*: *f* ∈ *set* (*toplevel-field-names* (*typ-info-t TYPE*('*a*)))
shows ∃ *s n*. *field-lookup* (*typ-info-t TYPE*('*a*)) [*f*] *0* = *Some* (*s*, *n*)
⟨*proof*⟩

lemma (**in** *c-type*) *field-lookup-typ-uinfo-t-Some*:
field-lookup (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*s*, *n*) ⟹
field-lookup (*typ-uinfo-t TYPE*('*a*)) *f m* = *Some* (*export-uinfo s*, *n*)
⟨*proof*⟩

lemma *toplevel-field-names-field-lookup-offset-conv*:
assumes *f*: *f* ∈ *set* (*toplevel-field-names* (*typ-info-t TYPE*('*a*)))
shows *snd* (*the* (*field-lookup* (*typ-uinfo-t TYPE*('*a*)) [*f*] *0*)) =
snd (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) [*f*] *0*))
⟨*proof*⟩

lemma *heap-update-list-fold-toplevel-field-names*:
fixes *p*::'*a ptr*
assumes *aggregate*: *aggregate* (*typ-info-t TYPE*('*a*))
assumes *cgrd*: *c-guard p*
shows
heap-update-list (*ptr-val p*) (*to-bytes x* (*heap-list h* (*size-of TYPE*('*a*)) (*ptr-val p*)))
h =
fold
(λ*f h*. *heap-update-list* (&(*p*→[*f*]))
(*access-ti*
(*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) [*f*] *0*)))
x
(*heap-list h* (*size-td* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) [*f*]

0)))) (&(*p*→[*f*]))))
        h)
  (*toplevel-field-names* (*typ-info-t TYPE*('*a*))) *h* (**is** *?LHS* = *fold ?F ?fs h*)
⟨*proof*⟩

lemma *heap-update-list-padding-fold-toplevel-field-names*:
  fixes *p*::'*a ptr*
  assumes *aggregate*: *aggregate* (*typ-info-t TYPE*('*a*))
  assumes *cgrd*: *c-guard p*
  assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
  shows
heap-update-list (*ptr-val p*) (*to-bytes x bs*) *h* =
  fold
   (λ*f h*. *heap-update-list* (&(*p*→[*f*]))
         (*access-ti*
           (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) [*f*] *0*)))
           x
           (*take* (*size-td* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) [*f*] *0*))))
             (*drop* ((*snd* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) [*f*] *0*)))) *bs*)))
         h)
   (*toplevel-field-names* (*typ-info-t TYPE*('*a*))) *h* (**is** *?LHS* = *fold ?F ?fs h*)
⟨*proof*⟩

lemma *heap-update-fold-toplevel-field-names*:
  fixes *p*::'*a ptr*
  assumes *aggregate*: *aggregate* (*typ-info-t TYPE*('*a*))
  assumes *cgrd*: *c-guard p*
  shows
heap-update p x h =
  fold
   (λ*f h*. *heap-update-list* (&(*p*→[*f*]))
         (*access-ti*
           (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) [*f*] *0*)))
           x
             (*heap-list h* (*size-td* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) [*f*]
0)))) (&(*p*→[*f*]))))
         h)
   (*toplevel-field-names* (*typ-info-t TYPE*('*a*))) *h* (**is** *?LHS* = *fold ?F ?fs h*)
  ⟨*proof*⟩

lemma *heap-update-padding-fold-toplevel-field-names*:
  fixes *p*::'*a ptr*
  assumes *aggregate*: *aggregate* (*typ-info-t TYPE*('*a*))
  assumes *cgrd*: *c-guard p*
  assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
  shows
heap-update-padding p x bs h =
  fold
   (λ*f h*. *heap-update-list* (&(*p*→[*f*]))

```
          (access-ti
            (fst (the (field-lookup (typ-info-t TYPE('a)) [f] 0)))
            x
            (take (size-td (fst (the (field-lookup (typ-info-t TYPE('a)) [f] 0))))
              (drop ((snd (the (field-lookup (typ-info-t TYPE('a)) [f] 0)))) bs)))
           h)
  (toplevel-field-names (typ-info-t TYPE('a))) h (is ?LHS = fold ?F ?fs h)
 ⟨proof⟩

lemma heap-update-fold-toplevel-field-names-no-padding:
  fixes p::'a ptr
  assumes aggregate: aggregate (typ-info-t TYPE('a))
  assumes cgrd: c-guard p
  shows
heap-update p x h =
  fold
   (λf h. heap-update-list (&(p→[f]))
          (access-ti
            (fst (the (field-lookup (typ-info-t TYPE('a)) [f] 0)))
            x
              (heap-list h (size-td (fst (the (field-lookup (typ-info-t TYPE('a)) [f]
0)))) (&(p→[f]))))
           h)
   (filter (Not o padding-field-name) (toplevel-field-names (typ-info-t TYPE('a))))
h (is ?LHS = fold ?F ?filter-fs h)
⟨proof⟩

lemma heap-list-concat-toplevel-field-names:
  fixes p::'a ptr
  assumes aggregate: aggregate (typ-info-t TYPE('a))
  assumes cgrd: c-guard p
  shows
   heap-list h (size-of TYPE('a)) (ptr-val p) =
    concat (map (λf. heap-list h (size-td (fst (the (field-lookup (typ-info-t TYPE('a))
[f] 0)))) (&(p→[f])))
         (toplevel-field-names (typ-info-t TYPE('a)))) (is ?LHS = concat (map ?F
?fs))
⟨proof⟩


lemma h-val-concat-toplevel-field-names:
  fixes p::'a ptr
  assumes aggregate: aggregate (typ-info-t TYPE('a))
  assumes cgrd: c-guard p
  shows
   h-val h p =
     from-bytes
          (concat (map (λf. heap-list h (size-td (fst (the (field-lookup (typ-info-t
TYPE('a)) [f] 0)))) (&(p→[f])))
```

(*toplevel-field-names* (*typ-info-t* *TYPE*('*a*)))))
⟨*proof*⟩

end

lemma *set-field-names-u-all-field-names-conv*:
set (*field-names-u* (*typ-uinfo-t* *TYPE*('*a*::*mem-type*)) *t*) =
{*f*. *f* ∈ *set* (*all-field-names* (*typ-info-t* *TYPE*('*a*))) ∧
(∃ *s*. *field-ti* *TYPE*('*a*) *f* = *Some* *s* ∧ *export-uinfo* *s* = *t*)}
⟨*proof*⟩

lemma *field-names-u-all-field-names-conv*:
field-names-u (*typ-uinfo-t* *TYPE*('*a*::*mem-type*)) *t* =
filter (λ*f*. (∃ *s*. *field-ti* *TYPE*('*a*) *f* = *Some* *s* ∧ *export-uinfo* *s* = *t*))
(*all-field-names* (*typ-info-t* *TYPE*('*a*)))
⟨*proof*⟩

lemma *set-field-names-all-field-names-conv*:
set (*field-names* (*typ-info-t* *TYPE*('*a*::*mem-type*)) *t*) =
{*f*. *f* ∈ *set* (*all-field-names* (*typ-info-t* *TYPE*('*a*))) ∧
(∃ *s*. *field-ti* *TYPE*('*a*) *f* = *Some* *s* ∧ *export-uinfo* *s* = *t*)}
⟨*proof*⟩

lemma *field-names-all-field-names-conv*:
field-names (*typ-info-t* *TYPE*('*a*::*mem-type*)) *t* =
filter (λ*f*. ∃ *s*. *field-ti* *TYPE*('*a*) *f* = *Some* *s* ∧ *export-uinfo* *s* = *t*) (*all-field-names*
(*typ-info-t* *TYPE*('*a*)))
⟨*proof*⟩

lemma *field-lookup-qualified-padding-field-name*:
fixes
t:: ('*a*, '*b*) *typ-info* **and**
st :: ('*a*, '*b*) *typ-info-struct* **and**
ts :: ('*a*, '*b*) *typ-info-tuple list* **and**
x :: ('*a*, '*b*) *typ-info-tuple*
shows
field-lookup *t* *f* *n* = *Some* (*s*, *m*) ⟹ *qualified-padding-field-name* *f* ⟹ *wf-padding*
t ⟹
is-padding-tag *s*
field-lookup-struct *st* *f* *n* = *Some* (*s*, *m*) ⟹ *qualified-padding-field-name* *f* ⟹
wf-padding-struct *st* ⟹
is-padding-tag *s*
field-lookup-list *ts* *f* *n* = *Some* (*s*, *m*) ⟹ *qualified-padding-field-name* *f* ⟹ *wf-padding-list*
ts ⟹
is-padding-tag *s*
field-lookup-tuple *x* *f* *n* = *Some* (*s*, *m*) ⟹ *qualified-padding-field-name* *f* ⟹
wf-padding-tuple *x* ⟹

is-padding-tag s
⟨*proof*⟩

lemma *all-field-names-empty-typ-info* [*simp*]: *all-field-names* (*empty-typ-info algn n*) = [[]]
⟨*proof*⟩

lemma *all-field-names-no-padding-empty-typ-info* [*simp*]:
filter (*Not o qualified-padding-field-name*) (*all-field-names* (*empty-typ-info algn n*)) = [[]]
⟨*proof*⟩

lemma *all-field-names-list-append* [*simp*]:
all-field-names-list (*xs* @ *ys*) = *all-field-names-list xs* @ *all-field-names-list ys*
⟨*proof*⟩

lemma *all-field-names-extend-ti*:
fixes
t :: *'a xtyp-info* **and**
st :: *'a xtyp-info-struct* **and**
ts :: *'a xtyp-info-tuple list* **and**
x :: *'a xtyp-info-tuple*
shows
all-field-names (*extend-ti t s n fn d*) = *all-field-names t* @ (*map* ((#) *fn*) (*all-field-names s*))
all-field-names-struct (*extend-ti-struct st s fn d*) = *all-field-names-struct st* @ (*map* ((#) *fn*) (*all-field-names s*))
all-field-names-list ts = *all-field-names-list ts*
all-field-names-tuple x = *all-field-names-tuple x*
⟨*proof*⟩

lemma *all-field-names-adjust-ti'*:
fixes
t :: *'a xtyp-info* **and**
st :: *'a xtyp-info-struct* **and**
ts :: *'a xtyp-info-tuple list* **and**
x :: *'a xtyp-info-tuple*
shows
all-field-names (*map-td* (*λn algn. update-desc acc upd*) (*update-desc acc upd*) *t*) =
all-field-names t
all-field-names-struct (*map-td-struct* (*λn algn. update-desc acc upd*) (*update-desc acc upd*) *st*) =
all-field-names-struct st
all-field-names-list (*map-td-list* (*λn algn. update-desc acc upd*) (*update-desc acc upd*) *ts*) =
all-field-names-list ts
all-field-names-tuple (*map-td-tuple* (*λn algn. update-desc acc upd*) (*update-desc acc upd*) *x*) =

all-field-names-tuple x
⟨*proof*⟩

lemma *all-field-names-adjust-ti*[*simp*]:
all-field-names (*adjust-ti t acc upd*) = *all-field-names t*
⟨*proof*⟩

lemma *all-field-names-no-padding-ti-pad-combine*:
filter (*Not o qualified-padding-field-name*) (*all-field-names* (*ti-pad-combine n t*))
=
filter (*Not o qualified-padding-field-name*) (*all-field-names t*)
⟨*proof*⟩

lemma *all-field-names-ti-typ-combine*:
all-field-names (*ti-typ-combine* (*t-b*::'*b*::*c-type itself*) *acc upd algn fn t*) =
all-field-names t @ (*map* ((#) *fn*) (*all-field-names* (*typ-info-t TYPE*('*b*))))
⟨*proof*⟩

lemma *all-field-names-no-padding-ti-typ-combine*:
assumes *not-padding*: ¬ *padding-field-name fn*
shows *filter* (*Not o qualified-padding-field-name*) (*all-field-names* (*ti-typ-combine* (*t-b*::'*b*::*c-type itself*) *acc upd algn fn t*)) =
filter (*Not o qualified-padding-field-name*) (*all-field-names t*) @
(*map* ((#) *fn*) (*filter* (*Not o qualified-padding-field-name*) (*all-field-names* (*typ-info-t TYPE*('*b*)))))
⟨*proof*⟩

lemma *all-field-names-no-padding-ti-typ-pad-combine*:
assumes *not-padding*: ¬ *padding-field-name fn*
shows *filter* (*Not o qualified-padding-field-name*) (*all-field-names* (*ti-typ-pad-combine* (*t-b*::'*b*::*c-type itself*) *acc upd algn fn t*)) =
filter (*Not o qualified-padding-field-name*) (*all-field-names t*) @
(*map* ((#) *fn*) (*filter* (*Not o qualified-padding-field-name*) (*all-field-names* (*typ-info-t TYPE*('*b*)))))
⟨*proof*⟩

lemma *all-field-names-map-align*[*simp*]: *all-field-names* (*map-align n t*) = *all-field-names t*
⟨*proof*⟩

lemma *all-field-names-no-padding-final-pad*:
filter (*Not o qualified-padding-field-name*) (*all-field-names* (*final-pad n t*))
= *filter* (*Not o qualified-padding-field-name*) (*all-field-names t*)
⟨*proof*⟩

lemmas *all-field-names-filter-no-padding-combinator-simps* =
all-field-names-no-padding-empty-typ-info
all-field-names-no-padding-final-pad
all-field-names-no-padding-ti-typ-pad-combine

all-field-names-no-padding-ti-typ-combine

lemma *all-field-names-array-tag-n*: *all-field-names* ((*array-tag-n n*)::(*'a*::*c-type*,*'b*::*finite*) *array xtyp-info*) =
[] #
concat (*map* (λ*i*. (*map* ((#) (*replicate i CHR ''1''*)) (*all-field-names* (*typ-info-t TYPE*(*'a*))))) [*0*..<*n*])
⟨*proof*⟩

lemma *all-field-names-array*:
all-field-names (*typ-info-t TYPE*(*'a*::*c-type*[*'b*::*finite*])) =
[] #
concat (*map* (λ*i*. (*map* ((#) (*replicate i CHR ''1''*)) (*all-field-names* (*typ-info-t TYPE*(*'a*))))) [*0*..<*CARD*(*'b*)])
⟨*proof*⟩

lemma *not-padding-field-name-replicate-1*[*simp*]: *padding-field-name* (*replicate n CHR ''1''*) = *False*
⟨*proof*⟩

lemma *non-empty-not-padding-field-conv*: (*x* ≠ [] ⟶ ¬ *padding-field-name* (*last x*)) ⟷ ¬ *qualified-padding-field-name x*
⟨*proof*⟩

named-theorems *all-field-names-no-padding* **and** *set-all-field-names-no-padding*

definition *all-field-names-no-padding* :: (*'a*, *'b*) *typ-desc* ⇒ *qualified-field-name list*
where
all-field-names-no-padding t = *filter* (*Not o qualified-padding-field-name*) (*all-field-names t*)

lemma *all-field-names-no-padding-combinator-simps*:
all-field-names-no-padding (*empty-typ-info algn nm*) = [[]]
all-field-names-no-padding (*final-pad n t*) = *all-field-names-no-padding t*
¬ *padding-field-name fn* ⟹
all-field-names-no-padding (*ti-typ-pad-combine* (*t-b*::*'b*::*c-type itself*) *acc upd algn fn t*) =
all-field-names-no-padding t @
map ((#) *fn*)
(*all-field-names-no-padding* (*typ-info-t TYPE*(*'b*::*c-type*)))
¬ *padding-field-name fn* ⟹
all-field-names-no-padding (*ti-typ-combine* (*t-b*::*'b*::*c-type itself*) *acc upd algn fn t*) =
all-field-names-no-padding *t* @
map ((#) *fn*)
(*all-field-names-no-padding* (*typ-info-t TYPE*(*'b*::*c-type*)))
⟨*proof*⟩

lemma *all-field-names-filter-no-padding-array*:
filter (*Not o qualified-padding-field-name*) (*all-field-names* (*typ-info-t TYPE*('*a*::*c-type*['*b*::*finite*])))
=
[] #
concat (*map* (λ*i*. (*map* ((#) (*replicate i CHR* ''*1*''))
(*filter* (*Not o qualified-padding-field-name*) (*all-field-names* (*typ-info-t TYPE*('*a*)))))))
[*0*..<*CARD*('*b*)])
⟨*proof*⟩

lemma *all-field-names-no-padding-array*[*all-field-names-no-padding*]:
all-field-names-no-padding (*typ-info-t TYPE*('*a*::*c-type*['*b*::*finite*])) =
[] #
concat (*map* (λ*i*. (*map* ((#) (*replicate i CHR* ''*1*''))
(*all-field-names-no-padding* (*typ-info-t TYPE*('*a*))))) [*0*..<*CARD*('*b*)])
⟨*proof*⟩

lemma *set-all-field-names-no-padding-array*[*set-all-field-names-no-padding*]:
set (*all-field-names-no-padding* (*typ-info-t TYPE*('*a*::*c-type*['*b*::*finite*]))) =
insert []
(⋃ *x*∈{*0*..<*CARD*('*b*)}.
(#) (*replicate x CHR* ''*1*'') ‘
set (*all-field-names-no-padding* (*typ-info-t TYPE*('*a*))))

⟨*proof*⟩

lemma *sub-typ-trans*: $t \leq_\tau s \Longrightarrow s \leq_\tau w \Longrightarrow t \leq_\tau w$
⟨*proof*⟩

lemma *sub-typ-trans-rev*: $s \leq_\tau w \Longrightarrow t \leq_\tau s \Longrightarrow t \leq_\tau w$
⟨*proof*⟩

lemma *element-typ-le-array-typ*: *typ-uinfo-t TYPE*('*a*::*mem-type*) ≤ *typ-uinfo-t TYPE*('*a*['*b*::*finite*])
⟨*proof*⟩

lemma *element-typ-subtyp-array-typ*: *TYPE* ('*a*::*mem-type*) ≤$_\tau$ *TYPE*('*a*['*b*::*finite*])
⟨*proof*⟩

lemma *field-lookup-sub-typ*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*::*c-type*)) *f 0* = *Some* (*s*, *m*)
assumes *match*: *export-uinfo s* = *export-uinfo* (*typ-info-t TYPE*('*b*::*c-type*))
shows *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)
⟨*proof*⟩

lemma *field-lookup-sub-typ'*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*::*c-type*)) *f 0* ≡ *Some* (*adjust-ti*

(*typ-info-t TYPE('b::mem-type)) acc upd, n*)
assumes *fg-cons*: *fg-cons acc upd*
shows *TYPE('b)* $\leq_\tau$ *TYPE('a)*
⟨*proof*⟩

lemma *all-field-names-no-padding-word*[*all-field-names-no-padding*]:
all-field-names-no-padding (typ-info-t (TYPE('a::len8 word))) = [[]]
⟨*proof*⟩

lemma *set-all-field-names-no-padding-word*[*set-all-field-names-no-padding*]:
set (all-field-names-no-padding (typ-info-t (TYPE('a::len8 word)))) = {[]}
⟨*proof*⟩

lemma *all-field-names-no-padding-ptr*[*all-field-names-no-padding*]:
all-field-names-no-padding (typ-info-t (TYPE('a::c-type ptr))) = [[]]
⟨*proof*⟩

lemma *set-all-field-names-no-padding-ptr*[*set-all-field-names-no-padding*]:
set (all-field-names-no-padding (typ-info-t (TYPE('a::c-type ptr)))) = {[]}
⟨*proof*⟩

definition *field-names-no-padding*::('*a*, '*b*) *typ-info* ⇒ *typ-uinfo* ⇒ *qualified-field-name list*
where *field-names-no-padding t s = filter (Not o qualified-padding-field-name) (field-names t s)*

lemma *set-field-names-no-padding-all-field-names-no-padding-conv*:
set (field-names-no-padding (typ-info-t TYPE('a::mem-type)) t) =
{f ∈ set (all-field-names-no-padding (typ-info-t TYPE('a))).
∃ s n. field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n) ∧ export-uinfo s = t}
⟨*proof*⟩

lemma *field-names-no-padding-all-field-names-no-padding-conv*:
field-names-no-padding (typ-info-t TYPE('a::mem-type)) t =
filter (λf. ∃ s n. field-lookup (typ-info-t TYPE('a)) f 0 = Some (s, n) ∧ export-uinfo s = t)
(all-field-names-no-padding (typ-info-t TYPE('a)))
⟨*proof*⟩

lemma *subset-all-field-names-no-padding-all-field-names*:
set (all-field-names-no-padding t) ⊆ set (all-field-names t)
⟨*proof*⟩

lemma *all-field-names-typ-uinfo-t-conv*:
all-field-names (typ-info-t (TYPE('a::c-type))) = all-field-names (typ-uinfo-t (TYPE('a::c-type)))
⟨*proof*⟩

lemma *all-field-names-no-padding-typ-uinfo-t-conv*:
all-field-names-no-padding (*typ-info-t* (*TYPE*('*a*::*c-type*))) = *all-field-names-no-padding* (*typ-uinfo-t* (*TYPE*('*a*::*c-type*)))
⟨*proof*⟩

lemma *update-ti-to-bytes-p*[*simp*]:
update-ti (*typ-info-t TYPE*('*a*::*xmem-type*)) (*to-bytes-p* (*v*::'*a*)) *w* = *v*
⟨*proof*⟩

context *mem-type*
begin

lemma *mem-type-access-ti-super-update-bs*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f* 0 = *Some* (*s*, *n*)
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *lbs'*: *length bs'* = *size-td s*
shows *access-ti* (*typ-info-t TYPE*('*a*))
(*update-ti s* (*access-ti*$_0$ *s w*) *v*) (*super-update-bs bs' bs n*) =
super-update-bs (*access-ti s w bs'*) (*access-ti* (*typ-info-t TYPE*('*a*)) *v bs*) *n*
⟨*proof*⟩

end

lemma *update-ti-undefined*[*simp*]:
assumes *NO-MATCH undefined w* **assumes** *bs*: *length bs* = *size-of TYPE*('*a*)
shows *update-ti* (*typ-info-t TYPE*('*a*::*xmem-type*)) *bs w* =
update-ti (*typ-info-t TYPE*('*a*)) *bs undefined*
⟨*proof*⟩

11.28 *heap-upd* and *heap-upd-list*

11.29 *heap-upd*

lemma *heap-upd-id*: *heap-upd* (*p*::'*a*::*xmem-type ptr*) *id* = *id*
⟨*proof*⟩

lemma *heap-upd-const*: *heap-upd p* (λ-. *x*) = *heap-update p x*
⟨*proof*⟩

lemma *heap-upd-comp*: *heap-upd* (*p*::'*a*::*xmem-type ptr*) (*f* ∘ *g*) = *heap-upd p f* ∘ *heap-upd p g*
⟨*proof*⟩

lemma *hrs-mem-update-heap-upd*:
hrs-mem-update (*heap-upd p g*) *h* = *hrs-mem-update* (*heap-update p* (*g* (*h-val* (*hrs-mem h*) *p*))) *h*
⟨*proof*⟩

lemma *heap-update-eq-heap-upd-list*:
fixes p :: $'a$::*mem-type ptr*
shows *heap-update* p x =
heap-upd-list (*size-of* *TYPE*($'a$)) (*ptr-val* p) (*access-ti* (*typ-info-t* *TYPE*($'a$)) x)
⟨*proof*⟩

lemma *heap-upd-list-id*[*simp*]: *heap-upd-list* n p *id* = *id*
⟨*proof*⟩

lemma *heap-upd-list-access-ti-typ-info-t*[*simp*]:
sz = *size-of* *TYPE*($'a$) ⟹
heap-upd-list sz p (*access-ti* (*typ-info-t* *TYPE*($'a$::*xmem-type*)) v) =
heap-update (*PTR*($'a$) p) v
⟨*proof*⟩

lemma *heap-list-heap-upd-list*:
n ≤ *addr-card* ⟹ *length* xs = n ⟹ (⋀xs. *length* xs = n ⟹ *length* (f xs) = n)
⟹
heap-list (*heap-upd-list* n p f h) n p = f (*heap-list* h n p)
⟨*proof*⟩

lemma *heap-upd-list-comp*:
assumes n ≤ *addr-card length* xs = n
assumes f: ⋀xs. *length* xs = n ⟹ *length* (f xs) = n
assumes g: ⋀xs. *length* xs = n ⟹ *length* (g xs) = n
shows *heap-upd-list* n p (f ∘ g) = *heap-upd-list* n p f ∘ *heap-upd-list* n p g
⟨*proof*⟩

lemma *heap-update-list-append*:
fixes v :: *word8*
shows *heap-update-list* s (xs @ ys) *hp* =
heap-update-list (s + *of-nat* (*length* xs)) ys (*heap-update-list* s xs *hp*)
⟨*proof*⟩

lemma *heap-update-list-super-update-bs*:
length *bs* + n ≤ *length* *bs*′ ⟹ *length* *bs*′ ≤ *addr-card* ⟹
heap-update-list (p + *of-nat* n) *bs* (*heap-update-list* p *bs*′ h) =
heap-update-list p (*super-update-bs* *bs* *bs*′ n) h
⟨*proof*⟩

lemma *update-ti-adjust-ti*:
fixes t::$'a$ *xtyp-info*
and st::$'a$ *xtyp-info-struct*
and ts::$'a$ *xtyp-info-tuple list*
and x::$'a$ *xtyp-info-tuple*
assumes *fg-cons*: *fg-cons* f g
shows
update-ti (*adjust-ti* t (f::$'b$ ⇒ $'a$) (g::$'a$ ⇒ $'b$ ⇒ $'b$)) *bs* v = g (*update-ti* t *bs* (f

v)) *v*
update-ti-struct (*map-td-struct* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*) *st*) *bs v* = *g* (*update-ti-struct st bs* (*f v*)) *v*
update-ti-list (*map-td-list* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*) *ts*) *bs v* = *g* (*update-ti-list ts bs* (*f v*)) *v*
update-ti-tuple (*map-td-tuple* (λ*n algn d*. *update-desc f g d*) (*update-desc f g*) *x*) *bs v* = *g* (*update-ti-tuple x bs* (*f v*)) *v*
⟨*proof*⟩

lemma *field-ti-field-lookupE*:
⟦ *field-ti TYPE*('*a* :: *c-type*) *f* = *Some t*; ⋀*n*. ⟦*field-lookup* (*typ-info-t TYPE*('*a*)) *f* 0 = *Some* (*t*, *n*) ⟧ ⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

lemma *field-ti-append-field-lookup*:
field-ti TYPE('*a*::*wf-type*) *f* = *Some u* ⟹ *field-lookup u g l* = *Some* (*v*, *k*) ⟹
field-ti TYPE('*a*) (*f* @ *g*) = *Some v*
⟨*proof*⟩

lemma *field-tiD*:
field-ti TYPE('*a*::*mem-type*) *f* = *Some t* ⟹
field-lookup (*typ-info-t TYPE*('*a*)) *f* 0 = *Some* (*t*, *field-offset TYPE*('*a*) *f*)
⟨*proof*⟩

lemma *wf-fd-field-lookup-mem-type*: *field-lookup* (*typ-info-t*(*TYPE*('*a*::*mem-type*))) *f m* = *Some* (*s*, *n*) ⟹ *wf-fd s*
⟨*proof*⟩

lemma *wf-fd-field-ti-mem-type*: *field-ti TYPE*('*a*::*mem-type*) *f* = *Some s* ⟹ *wf-fd s*
⟨*proof*⟩

lemma *field-lookup-offset-non-zero*:
NO-MATCH 0 *m* ⟹ *field-lookup t f* 0 = *Some* (*t*′, *n*) ⟹ *field-lookup t f m* = *Some* (*t*′, *m* + *n*)
⟨*proof*⟩

lemma *field-lookup-append-Some*:
assumes *wf*: *wf-desc t*
shows *field-lookup t* (*f*@*g*) *n* = *Some* (*s*, *m*) ⟹
∃ *w k*. *field-lookup t f n* = *Some* (*w*, *k*) ∧ *field-lookup w g k* = *Some* (*s*, *m*)
⟨*proof*⟩

11.30 *merge-ti*

definition *merge-ti* :: ('*a field-desc*, '*b*) *typ-desc* ⇒ '*a* ⇒ '*a* ⇒ '*a* **where**
merge-ti t a b = *update-ti t* (*access-ti*$_0$ *t a*) *b*

lemma *merge-ti-adjust-ti*[*simp*]:

fg-cons g s ⟹ *merge-ti* (*adjust-ti* (*typ-info-t TYPE*('*a*::*xmem-type*)) *g s*) = (λ*a*. *s* (*g a*))
⟨*proof*⟩

lemma *is-scene-merge-ti*:
assumes *t*: *field-ti TYPE*('*a*::*xmem-type*) *f* = *Some t* **shows** *is-scene* (*merge-ti t*)
⟨*proof*⟩

lemma *merge-ti-update-ti-disj*:
assumes ∗: *field-ti TYPE*('*a*::*xmem-type*) *f* = *Some t field-ti TYPE*('*a*::*xmem-type*) *g* = *Some u*
and *f-g*: *disj-fn f g*
assumes *bs*: *length bs* = *size-td u*
shows *merge-ti t x* (*update-ti u bs y*) = *update-ti u bs* (*merge-ti t x y*)
⟨*proof*⟩

lemma *disjnt-scene-merge-ti*:
assumes ∗: *field-ti TYPE*('*a*) *f* = *Some t field-ti TYPE*('*a*::*xmem-type*) *g* = *Some u*
and *f-g*: *disj-fn f g*
shows *disjnt-scene* (*merge-ti t*) (*merge-ti u*)
⟨*proof*⟩

lemma *access-ti-merge-ti-sub*:
assumes *r*: *field-ti TYPE*('*a*::*xmem-type*) *r* = *Some t* **and** *t*: *field-lookup t f 0* = *Some* (*v*, *n*)
and *bs*: *length bs* = *size-td v*
shows *access-ti v* (*merge-ti t x y*) *bs* = *access-ti v x bs*
⟨*proof*⟩

lemma *access-ti-merge-ti-disj*:
assumes *f*: *field-ti TYPE*('*a*::*xmem-type*) *f* = *Some t*
assumes *g*: *field-ti TYPE*('*a*::*xmem-type*) *g* = *Some u*
and *f-g*: *disj-fn f g*
and *bs*: *length bs* = *size-td u*
shows *access-ti u* (*merge-ti t x y*) *bs* = *access-ti u y bs*
⟨*proof*⟩

lemma *merge-ti-update-ti-sub*:
assumes *r*: *field-ti TYPE*('*a*::*xmem-type*) *r* = *Some t*
and *t*: *field-lookup t f 0* = *Some* (*v*, *n*)
and *bs*: *length bs* = *size-td v*
shows *merge-ti t x* (*update-ti v bs y*) = *merge-ti t x y*
⟨*proof*⟩

lemma *merge-ti-merge-ti-sub1*:
assumes *t*: *field-ti TYPE*('*a*::*xmem-type*) *f* = *Some t*
assumes *u*: *field-ti TYPE*('*a*) (*f* @ *g*) = *Some u*

shows *merge-ti t a* (*merge-ti u a b*) = *merge-ti t a b*
⟨*proof*⟩

lemma *merge-ti-merge-ti-sub2*:
assumes *t*: *field-ti TYPE*('*a*::*xmem-type*) *f* = *Some t*
assumes *u*: *field-ti TYPE*('*a*) (*f* @ *g*) = *Some u*
shows *merge-ti u a* (*merge-ti t a b*) = *merge-ti t a b*
⟨*proof*⟩

lemma *comm-scene-merge-ti-sub*:
assumes ∗: *field-ti TYPE*('*a*::*xmem-type*) *f* = *Some t field-ti TYPE*('*a*) (*f* @ *g*) = *Some u*
shows *comm-scene* (*merge-ti t*) (*merge-ti u*)
⟨*proof*⟩

lemma *comm-scene-merge-ti*:
assumes ∗: *field-ti TYPE*('*a*::*xmem-type*) *f* = *Some t field-ti TYPE*('*a*) *g* = *Some u*
shows *comm-scene* (*merge-ti t*) (*merge-ti u*)
⟨*proof*⟩

11.30.1 *merge-ti-list*

definition *merge-ti-list* **where**
merge-ti-list ts a = *fold* (λ*t*. *merge-ti t a*) *ts*

lemma *is-scene-merge-ti-list*:
list-all (λ*u*. ∃*f*. *field-ti TYPE*('*a*::*xmem-type*) *f* = *Some u*) *ts* ⟹
is-scene (*merge-ti-list ts*)
⟨*proof*⟩

lemma *merge-ti-list-nil*[*simp*]: *merge-ti-list* [] = (λ*a*. *id*)
⟨*proof*⟩

lemma *merge-ti-list-cons*[*simp*]:
merge-ti-list (*t* # *ts*) *a* = *merge-ti-list ts a* ∘ *merge-ti t a*
⟨*proof*⟩

lemma *merge-ti-list-append*[*simp*]:
merge-ti-list (*ts* @ *ts*′) *x y* = *merge-ti-list ts*′ *x* (*merge-ti-list ts x y*)
⟨*proof*⟩

lemma *access-ti-merge-ti-list*:
assumes *ts*: *list-all* (λ(*f*, *u*). *field-ti TYPE*('*a*::*xmem-type*) *f* = *Some u*) *ts*
distinct-prop disj-fn (*map fst ts*)
and *r-t*: (*r*, *t*) ∈ *set ts* **and** *v*: *field-lookup t f 0* = *Some* (*v*, *n*)
and *bs*: *length bs* = *size-td v*
shows *access-ti v* (*merge-ti-list* (*map snd ts*) *x y*) *bs* = *access-ti v x bs*
⟨*proof*⟩

lemma *merge-ti-list-update-ti*:
assumes *ts*: *list-all* (λ(*f*, *u*). *field-ti TYPE*('*a*::*xmem-type*) *f* = *Some u*) *ts* (*f*-*t*, *t*) ∈ *set ts*
and *disj*: *distinct-prop disj-fn* (*map fst ts*)
and *t*: *field-lookup t f 0* = *Some* (*v*, *n*)
and *bs*: *length bs* = *size-td v*
shows *merge-ti-list* (*map snd ts*) *x* (*update-ti v bs y*) = *merge-ti-list* (*map snd ts*) *x y*
⟨*proof*⟩

lemma *heap-update-eq-fold-subfields*:
assumes *ts*: *list-all* (λ(*f*, *u*). *field-ti TYPE*('*a*::*xmem-type*) *f* = *Some u*) *ts*
shows *heap-update p x* =
fold (λ(*f*, *u*). *heap-upd-list* (*size-td u*) &(*p*→*f*) (*access-ti u x*)) *ts* ∘
heap-upd-list (*size-of TYPE*('*a*)) (*ptr-val p*)
(*access-ti* (*typ-info-t TYPE*('*a*)) (*merge-ti-list* (*map snd ts*) *y x*))
⟨*proof*⟩

end

Chapter 12

More Building Blocks for our C-Language Model

theory *CLanguage*
imports
CProof
Lens
begin

12.1 addr bounds

lemma *addr-card-eq*: *addr-card* = *2^LENGTH*(*addr-bitsize*)
⟨*proof*⟩

lemma *size-of-bnd*: *size-of TYPE*('*a*::*mem-type*) < *2^LENGTH*(*addr-bitsize*)
⟨*proof*⟩

lemma *size-of-mem-type*[*simp*]: *size-of TYPE*('*c*::*mem-type*) ≠ *0*
⟨*proof*⟩

lemma *addr-card-len-of-conv*: *addr-card* = *2* ^ *len-of TYPE*(*addr-bitsize*)
⟨*proof*⟩

lemma *intvl-split*:
⟦ *n* ≥ *a* ⟧ ⟹ { *p* :: ('*a* :: *len*) *word* ..+ *n* } = { *p* ..+ *a* } ∪ { *p* + *of-nat a* ..+ (*n* − *a*)}

⟨*proof*⟩

12.2 More Heap Typing

primrec
htd-upd :: *addr* ⇒ *typ-slice list* ⇒ *heap-typ-desc* ⇒ *heap-typ-desc*
where

htd-upd p [] d = d
| *htd-upd p (x#xs) d = htd-upd (p+1) xs (d(p := (True, x)))*

definition (**in** *c-type*) *ptr-force-type* :: *'a ptr* ⇒ *heap-typ-desc* ⇒ *heap-typ-desc* **where**
ptr-force-type p ≡ *htd-upd* (*ptr-val p*) (*typ-slices TYPE('a)*)

definition *ptr-force-types* :: *'a*::*c-type ptr list* ⇒ *heap-typ-desc* ⇒ *heap-typ-desc* **where**
ptr-force-types = *fold ptr-force-type*

definition *ptr-force-free* :: *addr* ⇒ *nat* ⇒ *heap-typ-desc* ⇒ *heap-typ-desc* **where**
ptr-force-free p b = *ptr-force-types* (*map* (λ*n*. *PTR*(*8 word*) *p* +$_p$ *n*) (*map of-nat* [*0*..<*2*^*b*]))

definition *ptr-u* :: *'a*::*c-type ptr* ⇒ (*addr* × *typ-uinfo*) **where**
ptr-u p = (*ptr-val p*, *typ-uinfo-t TYPE('a)*)

abbreviation *ptr-span-u* ≡ (λ(*a*, *t*). {*a* ..+ *size-td t*})

definition *typ-slices-u* :: *typ-uinfo* ⇒ *typ-slice list* **where**
typ-slices-u t = *map* (λ*n*. *list-map* (*typ-slice-t t n*)) [*0*..<*size-td t*]

definition *ptr-force-type-u* :: *typ-uinfo* ⇒ *addr* ⇒ *heap-typ-desc* ⇒ *heap-typ-desc* **where**
ptr-force-type-u t a ≡ *htd-upd a* (*typ-slices-u t*)

lemma *heap-update-list-id*: *heap-update-list x* [] = (λ*x*. *x*)
⟨*proof*⟩

lemma *to-bytes-word8*:
to-bytes (*v* :: *word8*) *xs* = [*v*]
⟨*proof*⟩

lemma *heap-update-heap-update-list*:
⟦ *ptr-val p* = *q* + (*of-nat* (*length l*)); *Suc* (*length l*) < *addr-card* ⟧ ⟹
heap-update (*p* :: *word8 ptr*) *v* (*heap-update-list q l s*) = (*heap-update-list q* (*l* @ [*v*]) *s*)
⟨*proof*⟩

lemma *htd-upd-empty*[*simp*]: *htd-upd p* [] = *id*
⟨*proof*⟩

lemma *htd-upd-append*:
htd-upd p (*xs* @ *ys*) = *htd-upd* (*p* + *of-nat* (*length xs*)) *ys* ∘ *htd-upd p xs*
⟨*proof*⟩

lemma *htd-upd-singleton*[*simp*]: *htd-upd p* [*x*] = *upd-fun p* (λ*h*. (*True*, *x*))

⟨*proof*⟩

lemma *intvl-Suc-eq*: $\{p\ ..+\ Suc\ n\} = insert\ p\ \{p + 1\ ..+\ n\}$
⟨*proof*⟩

lemma *htd-upd-disj*: $p \notin \{p'\ ..+\ length\ v\} \Longrightarrow htd\text{-}upd\ p'\ v\ h\ p = h\ p$
⟨*proof*⟩

lemma *htd-upd-head*:
$xs \neq [] \Longrightarrow length\ xs \leq addr\text{-}card \Longrightarrow htd\text{-}upd\ p\ xs\ s\ p = (True,\ hd\ xs)$
⟨*proof*⟩

lemma *htd-upd-at*:
$i < length\ xs \Longrightarrow length\ xs \leq addr\text{-}card \Longrightarrow htd\text{-}upd\ p\ xs\ s\ (p\ +\ of\text{-}nat\ i) = (True,\ xs\ !\ i)$
⟨*proof*⟩

lemma *ptr-force-type-disj*:
$p \notin ptr\text{-}span\ (p' :: 'a{::}mem\text{-}type\ ptr) \Longrightarrow ptr\text{-}force\text{-}type\ p'\ h\ p = h\ p$
⟨*proof*⟩

lemma *ptr-force-types-disj*:
fixes $xs :: 'a{::}mem\text{-}type\ ptr\ list$
assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow i \notin ptr\text{-}span\ x$
shows $ptr\text{-}force\text{-}types\ xs\ h\ i = h\ i$
⟨*proof*⟩

12.2.1 Heap type tag and valid simple footprint

datatype *heap-typ-contents* =
HeapType typ-uinfo
| *HeapFootprint*
| *HeapEmpty*

definition
heap-type-tag :: *heap-typ-desc* ⇒ *addr* ⇒ *heap-typ-contents*
where
heap-type-tag d a ≡
(*if fst* (*d a*) = *False* ∨ (∀ *x*. (*snd* (*d a*)) *x* = *None*) ∨ (∀ *x*. (*snd* (*d a*)) *x* ≠ *None*) *then*
HeapEmpty
else
case (*snd* (*d a*)) (*GREATEST x*. *snd* (*d a*) *x* ≠ *None*) *of*
Some (-, *False*) ⇒ *HeapFootprint*
| *Some* (*x*, *True*) ⇒ *HeapType x*
| *None* ⇒ *HeapEmpty*)

definition
valid-simple-footprint :: *heap-typ-desc* ⇒ *addr* ⇒ *typ-uinfo* ⇒ *bool*
where
valid-simple-footprint d x t ≡
heap-type-tag d x = *HeapType t* ∧
(∀ *y*. *y* ∈ {*x* + *1*..+ (*size-td t*)− *Suc 0*} ⟶ *heap-type-tag d y* = *HeapFootprint*)

lemma *valid-simple-footprint-size-td*:
assumes *valid*: *valid-simple-footprint d x t*
shows *size-td t* ≤ *addr-card*
⟨*proof*⟩

lemma *valid-simple-footprintI*:
⟦ *heap-type-tag d x* = *HeapType t*; ⋀*y*. *y* ∈ {*x* + *1*..+(*size-td t*) − *Suc 0*} ⟹ *heap-type-tag d y* = *HeapFootprint* ⟧
⟹ *valid-simple-footprint d x t*
⟨*proof*⟩

lemma *valid-simple-footprintD*:
valid-simple-footprint d x t ⟹ *heap-type-tag d x* = *HeapType t*
⟨*proof*⟩

lemma *valid-simple-footprintD2*:
⟦ *valid-simple-footprint d x t*; *y* ∈ {*x* + *1*..+(*size-td t*) − *Suc 0*} ⟧ ⟹ *heap-type-tag d y* = *HeapFootprint*
⟨*proof*⟩

lemma *typ-slices-not-empty*:
typ-slices (*x*::('*a*::{*mem-type*} *itself*)) ≠ []
⟨*proof*⟩

lemma *last-typ-slice-t*:
(*last* (*typ-slice-t t 0*)) = (*t*, *True*)
⟨*proof*⟩

lemma *last-typ-slice-t-non-zero*:
k ≠ *0* ⟹ (*last* (*typ-slice-t t k*)) = (*t*, *False*)
⟨*proof*⟩

lemma *if-eqI*:
⟦ *a* ⟹ *x* = *z*; ¬ *a* ⟹ *y* = *z* ⟧ ⟹ (*if a then x else y*) = *z*
⟨*proof*⟩

lemma *heap-type-tag-ptr-retyp*:
snd (*s* (*ptr-val t*)) = *Map.empty* ⟹
heap-type-tag (*ptr-retyp* (*t* :: '*a*::*mem-type ptr*) *s*) (*ptr-val t*) = *HeapType* (*typ-uinfo-t TYPE*('*a*))

⟨*proof*⟩

lemma *not-snd-last-typ-slice-t*:
$k \neq 0 \Longrightarrow \neg$ *snd* (*last* (*typ-slice-t z k*))
⟨*proof*⟩

lemma *heap-type-tag-ptr-retyp-rest*:
⟦ *snd* (*s* (*ptr-val t* + *k*)) = *Map.empty*; 0 < *k*; *unat k* < *size-td* (*typ-uinfo-t TYPE*('*a*)) ⟧ ⟹
heap-type-tag (*ptr-retyp* (*t* :: '*a*::*mem-type ptr*) *s*) (*ptr-val t* + *k*) = *HeapFootprint*
⟨*proof*⟩

lemma *typ-slices-addr-card* [*simp*]:
length (*typ-slices* (*x*::('*a*::{*mem-type*} *itself*))) < *addr-card*
⟨*proof*⟩

lemma *unat-less-impl-less*:
unat a < *unat b* ⟹ *a* < *b*
⟨*proof*⟩

lemma *valid-simple-footprint-ptr-retyp*:
⟦ ∀ *k* < *size-td* (*typ-uinfo-t TYPE*('*a*)). *snd* (*s* (*ptr-val t* + *of-nat k*)) = *Map.empty*;
1 ≤ *size-td* (*typ-uinfo-t TYPE*('*a*));
size-td (*typ-uinfo-t TYPE*('*a*)) < *addr-card* ⟧
⟹ *valid-simple-footprint* (*ptr-retyp* (*t* :: '*a*::*mem-type ptr*) *s*) (*ptr-val t*) (*typ-uinfo-t TYPE*('*a*))
⟨*proof*⟩

lemma *heap-type-tag-cong*: *s p* = *s'* *p* ⟹ *heap-type-tag s p* = *heap-type-tag s'* *p*
⟨*proof*⟩

lemma *heap-type-tag*:
assumes *eq*: *h p* = (*f*, *list-map l*)
shows *heap-type-tag h p* =
(*if* ¬ *f* ∨ *l* = [] *then HeapEmpty else*
case last l of (*x*, *b*) ⇒ *if b then HeapType x else HeapFootprint*)
⟨*proof*⟩

lemma *valid-simple-footprint-cong-state*:
assumes *t*: *wf-size-desc t*
assumes *eq*: ⋀*p'*. *p'* ∈ {*p* ..+*size-td t*} ⟹ *s p'* = *s' p'*
shows *valid-simple-footprint s p t* ⟷ *valid-simple-footprint s' p t*
⟨*proof*⟩

lemma *heap-type-tag-ptr-force-type-HeapType*:
fixes *x* :: '*a*::*mem-type ptr*
shows *heap-type-tag* (*ptr-force-type x s*) (*ptr-val x*) = *HeapType* (*typ-uinfo-t*

$TYPE('a)$)
⟨*proof*⟩

lemma *heap-type-tag-ptr-force-type-HeapFootprint*:
fixes p :: $'a$::*mem-type ptr*
shows $p' \in \{ptr\text{-}val\ p + 1\ ..+\ size\text{-}of\ TYPE('a) - Suc\ 0\} \Longrightarrow$
heap-type-tag (*ptr-force-type p s*) p' = *HeapFootprint*
⟨*proof*⟩

lemma *valid-simple-footprint-ptr-force-type-iff*:
fixes p :: $'a$::*mem-type ptr*
assumes t: *wf-size-desc t*
shows *valid-simple-footprint* (*ptr-force-type p s*) *a t* $\longleftrightarrow$
(*valid-simple-footprint s a t* $\wedge$ *disjnt* $\{a\ ..+\ size\text{-}td\ t\}$ (*ptr-span p*)) $\vee$
(t = *typ-uinfo-t* $TYPE('a) \wedge p$ = *Ptr a*)
⟨*proof*⟩

lemma *valid-simple-footprint-fold-ptr-force-type-iff*:
fixes *ps* :: $'a$::*mem-type ptr list*
assumes [*simp*]: *wf-size-desc t*
shows *distinct-prop* (λ*p1 p2*. *disjnt* (*ptr-span p1*) (*ptr-span p2*)) *ps* $\Longrightarrow$
valid-simple-footprint (*fold ptr-force-type ps s*) *a t* $\longleftrightarrow$
(*valid-simple-footprint s a t* $\wedge$ *disjnt* $\{a\ ..+\ size\text{-}td\ t\}$ ($\bigcup p \in set\ ps$. *ptr-span p*)) $\vee$
(t = *typ-uinfo-t* $TYPE('a) \wedge$ *Ptr a* $\in$ *set ps*)
⟨*proof*⟩

12.3 Pointers to local (stack) variables

definition *stack-typ-info t* = (*stack-byte-name* $\notin$ *td-names t*)

lemma *stack-typ-info-export-uinfo*[*simp*]: *stack-typ-info* (*export-uinfo t*) = *stack-typ-info t*
⟨*proof*⟩

lemma *stack-typ-info-td-set*:
assumes *stack-typ*: *stack-typ-info t*
assumes t': $(t', n) \in$ *td-set t 0*
shows *typ-name* $t' \neq$ *stack-byte-name*
⟨*proof*⟩

lemma *stack-typ-info-td-set-stack-byte*:
assumes *stack-typ*: *stack-typ-info t*
assumes t': $(t', n) \in$ *td-set t 0*
shows $t' \neq$ *typ-uinfo-t* $TYPE(stack\text{-}byte)$
⟨*proof*⟩

class *stack-type* = *c-type* +

assumes *stack-typ-info*: *stack-typ-info* (*typ-info-t TYPE*('*a*))
begin
lemma *stack-typ-uinfo*: *stack-typ-info* (*typ-uinfo-t TYPE*('*a*))
⟨*proof*⟩

lemma *no-stack-byte* [*simp*]: *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
⟨*proof*⟩

end

lemma *stack-typ-info-no-stack-byte*:
stack-typ-info t ⟹ *t* ≠ *typ-uinfo-t TYPE*(*stack-byte*)
⟨*proof*⟩

lemma *stack-typ-info-empty-typ-info*:
nm ≠ *stack-byte-name* ⟹ *stack-typ-info* (*empty-typ-info algn nm*)
⟨*proof*⟩

lemma *td-set-list-to-set*:
(*t*, *m*) ∈ *td-set-list xs n* ⟹ (∃ *x k*. *x* ∈ *set xs* ∧ (*t*, *k*) ∈ *td-set* (*dt-fst x*) *0*)
⟨*proof*⟩

lemma *td-names-list-to-set*:
nm ∈ *td-names-list xs* ⟹ (∃ *x*. *x* ∈ *set xs* ∧ *nm* ∈ *td-names* (*dt-fst x*))
⟨*proof*⟩

lemma *set-to-td-names-list*:
x ∈ *set xs* ⟹ *nm* ∈ *td-names* (*dt-fst x*) ⟹ *nm* ∈ *td-names-list xs*
⟨*proof*⟩

lemma *stack-typ-info-TypAggregateI*:
assumes *xs*: ⋀*x*. *x* ∈ *set xs* ⟹ *stack-typ-info* (*dt-fst x*)
assumes *nm*: *nm* ≠ *stack-byte-name*
shows *stack-typ-info* (*TypDesc algn* (*TypAggregate* (*xs*)) *nm*)
⟨*proof*⟩

lemma *TypAggregate-not-stack-byte*:
TypDesc algn (*TypAggregate xs*) *nm* ≠ *typ-uinfo-t TYPE*(*stack-byte*)
⟨*proof*⟩

lemma *stack-typ-info-TypAggregateD*:
assumes *aggr*: *stack-typ-info* (*TypDesc algn* (*TypAggregate* (*xs*)) *nm*)
assumes *x*: *x* ∈ *set xs*
shows *stack-typ-info* (*dt-fst x*)
⟨*proof*⟩

lemma *stack-typ-info-extend-ti*:
stack-typ-info s $\Longrightarrow$ *stack-typ-info t* $\Longrightarrow$
stack-typ-info (*extend-ti s t algn fn d*)
⟨*proof*⟩

lemma *stack-typ-info-ti-pad-combine*:
stack-typ-info t $\Longrightarrow$ *stack-typ-info* (*ti-pad-combine n t*)
⟨*proof*⟩

lemma *stack-typ-info-export-uinfo-adjust-ti'*:
shows *stack-typ-info* (*adjust-ti* (*typ-info-t TYPE*('*b*::*stack-type*)) *acc upd*)
⟨*proof*⟩

lemma *stack-typ-info-export-uinfo-adjust-ti*:
shows *stack-typ-info* (*typ-info-t* (*TYPE*('*b*))) $\Longrightarrow$ *stack-typ-info* (*adjust-ti* (*typ-info-t TYPE*('*b*::*c-type*)) *acc upd*)
⟨*proof*⟩

lemma *stack-typ-info-ti-typ-combine'*:
stack-typ-info t $\Longrightarrow$
stack-typ-info (*ti-typ-combine TYPE*('*b*::*stack-type*) *acc upd algn nm t*)
⟨*proof*⟩

lemma *stack-typ-info-ti-typ-combine*:
stack-typ-info (*typ-info-t* (*TYPE*('*b*))) $\Longrightarrow$ *stack-typ-info t* $\Longrightarrow$
stack-typ-info (*ti-typ-combine TYPE*('*b*::*c-type*) *acc upd algn nm t*)
⟨*proof*⟩

lemma *stack-typ-info-ti-typ-pad-combine'*:
stack-typ-info t $\Longrightarrow$
stack-typ-info (*ti-typ-pad-combine TYPE*('*b*::*stack-type*) *acc upd algn nm t*)
⟨*proof*⟩

lemma *stack-typ-info-ti-typ-pad-combine*:
stack-typ-info (*typ-info-t* (*TYPE*('*b*))) $\Longrightarrow$ *stack-typ-info t* $\Longrightarrow$
stack-typ-info (*ti-typ-pad-combine TYPE*('*b*::*c-type*) *acc upd algn nm t*)
⟨*proof*⟩

lemma *stack-typ-info-map-align*:
stack-typ-info t $\Longrightarrow$ *stack-typ-info* (*map-align algn t*)
⟨*proof*⟩

lemma *stack-typ-info-final-pad*: *stack-typ-info t* $\Longrightarrow$
stack-typ-info (*final-pad algn t*)
⟨*proof*⟩

lemmas *stack-typ-info-intros* =

stack-typ-info-empty-typ-info
stack-typ-info-ti-typ-pad-combine
stack-typ-info-final-pad
stack-typ-info

named-theorems *stack-typ-info*

definition *valid-root-footprint* :: *heap-typ-desc* ⇒ *addr* ⇒ *typ-uinfo* ⇒ *bool* **where**
valid-root-footprint d x t ≡
let n = *size-td t in*
0 < *n* ∧ (∀ *y*. *y* < *n* ⟶
snd (*d* (*x* + *of-nat y*)) = *list-map* (*typ-slice-t t y*) ∧ *fst* (*d* (*x* + *of-nat y*)))

lemma *valid-root-footprint-valid-footprint*: *valid-root-footprint d x t* ⟹ *valid-footprint d x t*
⟨*proof*⟩

lemma *valid-root-footprint-valid-footprint-dom-conv*:
valid-root-footprint d a t
⟷
(*valid-footprint d a t* ∧
(∀ *n*. *n* < *size-td t* ⟶ *dom* (*snd* (*d* (*a* + *of-nat n*))) = {*0*..<*length* (*typ-slice-t t n*)}))
⟨*proof*⟩

lemma *valid-root-footprint-dom-typing*:
valid-root-footprint d a t ⟹ *n* < *size-td t* ⟹
dom (*snd* (*d* (*a* + *of-nat n*))) = {*0*..<*length* (*typ-slice-t t n*)}
⟨*proof*⟩

lemma *valid-root-footprint-typing-conv*:
fixes *p*::*'a*::*c-type ptr*
assumes *valid*: *valid-root-footprint d* (*ptr-val p*) (*typ-uinfo-t TYPE*(*'a*))
assumes *n*: *n* < *size-of* (*TYPE*(*'a*))
shows *d* (*ptr-val p* + *of-nat n*) = (*True*, *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*(*'a*)) *n*))
⟨*proof*⟩

definition
root-ptr-valid :: *heap-typ-desc* ⇒ *'a*::*c-type ptr* ⇒ *bool*
where
root-ptr-valid d p ≡
valid-root-footprint d (*ptr-val* (*p*::*'a ptr*)) (*typ-uinfo-t TYPE*(*'a*)) ∧
c-guard p

lemma *root-ptr-valid-c-guard*: *root-ptr-valid d p* ⟹ *c-guard p*

⟨*proof*⟩

lemma *root-ptr-valid-typing-conv*:
fixes *p*::'*a*::*c-type ptr*
assumes *valid*: *root-ptr-valid d p*
assumes *n*: *n* < *size-of* (*TYPE*('*a*))
shows *d* (*ptr-val p* + *of-nat n*) = (*True*, *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*('*a*)) *n*))
⟨*proof*⟩

lemma *root-ptr-valid-h-t-valid*: *root-ptr-valid d p* ⟹ *d*, *c-guard* ⊨$_t$ *p*
⟨*proof*⟩

lemma *valid-root-footprint-cong-state*:
assumes *t*: *wf-size-desc t*
assumes *eq*: ⋀*p'*. *p'* ∈ {*p* ..+*size-td t*} ⟹ *s p'* = *s' p'*
shows *valid-root-footprint s p t* ⟷ *valid-root-footprint s' p t*
⟨*proof*⟩

lemma (**in** *mem-type*) *valid-root-foot-print-ptr-force-type*:
valid-root-footprint
(*ptr-force-type* (*p*::'*a ptr*) *s*) (*ptr-val p*) (*typ-uinfo-t* (*TYPE*('*a*)))
⟨*proof*⟩

lemma *list-map-greatest-last*: *xs* ≠ [] ⟹ *last xs* = *v* ⟹ *list-map xs* (*GREATEST k*. ∃ *v*. *list-map xs k* = *Some v*) = *Some v*
⟨*proof*⟩

lemma *valid-root-footprint-valid-simple-footprint*:
assumes *valid-td*: *size-td t* ≤ *addr-card*
shows *valid-root-footprint d x t* ⟹ *valid-simple-footprint d x t*
⟨*proof*⟩

lemma *valid-root-footprint-valid-simple-footprint-typ-uinfo-t*:
assumes *valid-root*: *valid-root-footprint d x* (*typ-uinfo-t TYPE*('*a*::*mem-type*))
shows *valid-simple-footprint d x* (*typ-uinfo-t TYPE*('*a*::*mem-type*))
⟨*proof*⟩

lemma *first-in-intvl*:
b ≠ *0* ⟹ *a* ∈ {*a* ..+ *b*}
⟨*proof*⟩

lemma *list-map-comono*:
assumes *s*: *list-map m* ⊆$_m$ *list-map n*
shows *m* ≤ *n*
⟨*proof*⟩

lemma *valid-root-footprint-overlap-sub-typ*:

assumes *valid-root-x*: *valid-root-footprint d x t*
assumes *valid-y*: *valid-footprint d y s*
assumes *overlap*: $\{x\ ..+\ size\text{-}td\ t\} \cap \{y\ ..+\ size\text{-}td\ s\} \neq \{\}$
shows $s \leq t$
⟨*proof*⟩

lemma *valid-root-footprint-type-neq*:
$[\![$ *valid-root-footprint d p s*;
valid-root-footprint d q t;
$s \neq t\ ]\!] \Longrightarrow$
$p \notin \{q..+\ (size\text{-}td\ t)\}$
⟨*proof*⟩

lemma *valid-root-footprint-ptr-force-type-iff*:
fixes *p* :: *'a*::*mem-type ptr*
assumes *t*: *wf-size-desc t*
shows *valid-root-footprint* (*ptr-force-type p s*) *a t* $\longleftrightarrow$
(*valid-root-footprint s a t* $\wedge$ *disjnt* $\{a\ ..+\ size\text{-}td\ t\}$ (*ptr-span p*)) $\vee$
(t = *typ-uinfo-t TYPE*(*'a*) $\wedge$ p = *Ptr a*)
⟨*proof*⟩

lemma *valid-root-footprint-fold-ptr-force-type-iff*:
fixes *ps* :: *'a*::*mem-type ptr list*
assumes [*simp*]: *wf-size-desc t*
shows *distinct-prop* (λ*p1 p2*. *disjnt* (*ptr-span p1*) (*ptr-span p2*)) *ps* $\Longrightarrow$
valid-root-footprint (*fold ptr-force-type ps s*) *a t* $\longleftrightarrow$
(*valid-root-footprint s a t* $\wedge$ *disjnt* $\{a\ ..+\ size\text{-}td\ t\}$ $(\bigcup p{\in}set\ ps.\ ptr\text{-}span\ p)$)
$\vee$
(t = *typ-uinfo-t TYPE*(*'a*) $\wedge$ *Ptr a* $\in$ *set ps*)
⟨*proof*⟩

lemma *valid-simple-footprint-neq*:
assumes *valid-p*: *valid-simple-footprint d p s*
and *valid-q*: *valid-simple-footprint d q t*
and *neq*: $p \neq q$
shows $p \notin \{q..+\ (size\text{-}td\ t)\}$
⟨*proof*⟩

lemma *valid-simple-footprint-type-neq*:
$[\![$ *valid-simple-footprint d p s*;
valid-simple-footprint d q t;
$s \neq t\ ]\!] \Longrightarrow$
$p \notin \{q..+\ (size\text{-}td\ t)\}$
⟨*proof*⟩

lemma *valid-simple-footprint-neq-disjoint*:
$[\![$ *valid-simple-footprint d p s*; *valid-simple-footprint d q t*; $p \neq q\ ]\!] \Longrightarrow$

$\{p..+(size\text{-}td\ s)\} \cap \{q..+\ (size\text{-}td\ t)\} = \{\}$
⟨*proof*⟩

lemma *valid-simple-footprint-type-neq-disjoint*:
⟦ *valid-simple-footprint d p s*;
valid-simple-footprint d q t;
$s \neq t$ ⟧ ⟹
$\{p..+(size\text{-}td\ s)\} \cap \{q..+\ (size\text{-}td\ t)\} = \{\}$
⟨*proof*⟩

lemma *valid-simple-footprint-disjnt-or-eq*:
valid-simple-footprint d a1 t1 ⟹ *valid-simple-footprint d a2 t2* ⟹
disjnt $\{a1\ ..+\ size\text{-}td\ t1\}$ $\{a2\ ..+\ size\text{-}td\ t2\} \vee (a1 = a2 \wedge t1 = t2)$
⟨*proof*⟩

lemma *valid-root-footprint-type-neq-disjoint*:
⟦ *valid-root-footprint d p s*;
valid-root-footprint d q t;
$s \neq t$ ⟧ ⟹
$\{p..+(size\text{-}td\ s)\} \cap \{q..+\ (size\text{-}td\ t)\} = \{\}$
⟨*proof*⟩

lemma *valid-root-footprint-neq*:
assumes *valid-p*: *valid-root-footprint d p s*
and *valid-q*: *valid-root-footprint d q t*
and *neq*: $p \neq q$
shows $p \notin \{q..+\ (size\text{-}td\ t)\}$
⟨*proof*⟩

lemma *valid-root-footprint-neq-disjoint*:
⟦ *valid-root-footprint d p s*; *valid-root-footprint d q t*; $p \neq q$ ⟧ ⟹
$\{p..+(size\text{-}td\ s)\} \cap \{q..+\ (size\text{-}td\ t)\} = \{\}$
⟨*proof*⟩

lemma *valid-root-footprint-disjnt-or-eq*:
valid-root-footprint d a1 t1 ⟹ *valid-root-footprint d a2 t2* ⟹
disjnt $\{a1\ ..+\ size\text{-}td\ t1\}$ $\{a2\ ..+\ size\text{-}td\ t2\} \vee (a1 = a2 \wedge t1 = t2)$
⟨*proof*⟩

definition *ptr-aligned-u* :: *typ-uinfo* ⇒ *addr* ⇒ *bool* **where**
ptr-aligned-u t a ≡ $2\hat{}(align\text{-}td\ t)$ *dvd unat a*

lemma *ptr-aligned-ptr-aligned-u-conv*:
fixes *p*::*'a*::*c-type ptr*
shows *ptr-aligned p* = *ptr-aligned-u* (*typ-uinfo-t TYPE*(*'a*)) (*ptr-val p*)
⟨*proof*⟩

definition *c-null-guard-u* :: *typ-uinfo* ⇒ *addr* ⇒ *bool* **where**
c-null-guard-u t a ≡ $0 \notin \{a..+size\text{-}td\ t\}$

lemma *c-null-guard-c-null-guard-u-conv*:
fixes *p*::*'a*::*c-type ptr*
shows *c-null-guard p* = *c-null-guard-u* (*typ-uinfo-t TYPE*(*'a*)) (*ptr-val p*)
⟨*proof*⟩

definition *c-guard-u* :: *typ-uinfo* ⇒ *addr* ⇒ *bool* **where**
c-guard-u t a ≡ *ptr-aligned-u t a* ∧ *c-null-guard-u t a*

lemma *c-guard-c-guard-u-conv*:
fixes *p*::*'a*::*c-type ptr*
shows *c-guard p* = *c-guard-u* (*typ-uinfo-t TYPE*(*'a*)) (*ptr-val p*)
⟨*proof*⟩

definition
root-ptr-valid-u :: *typ-uinfo* ⇒ *heap-typ-desc* ⇒ *addr* ⇒ *bool* **where**
root-ptr-valid-u t d a ≡ *valid-root-footprint d a t* ∧ *c-guard-u t a*

definition
cvalid-u :: *typ-uinfo* ⇒ *heap-typ-desc* ⇒ *addr* ⇒ *bool* **where**
cvalid-u t d a ≡ *valid-footprint d a t* ∧ *c-guard-u t a*

lemma *root-ptr-valid-root-ptr-valid-u-conv*:
fixes *p*::*'a*::*c-type ptr*
shows *root-ptr-valid d p* = *root-ptr-valid-u* (*typ-uinfo-t TYPE*(*'a*)) *d* (*ptr-val p*)
⟨*proof*⟩

lemma *root-ptr-valid-ptr-force-type*:
c-guard p ⟹ *root-ptr-valid* (*ptr-force-type p s*) (*p*::*'a*::*mem-type ptr*)
⟨*proof*⟩
lemma *cvalid-cvalid-u-conv*:
fixes *p*::*'a*::*c-type ptr*
shows *d* ⊨*t p* = *cvalid-u* (*typ-uinfo-t TYPE*(*'a*)) *d* (*ptr-val p*)
⟨*proof*⟩

lemma *root-ptr-valid-u-cvalid-u*: *root-ptr-valid-u t d a* ⟹ *cvalid-u t d a*
⟨*proof*⟩

lemma *fold-root-ptr-valid-u*:
root-ptr-valid-u (*typ-uinfo-t TYPE*(*'a*::*c-type*)) *d a* = *root-ptr-valid d* (*PTR* (*'a*) *a*)
⟨*proof*⟩

lemma *ptr-force-type-eq-ptr-force-type-u*:
ptr-force-type (*p* :: *'a*::*c-type ptr*) = *ptr-force-type-u* (*typ-uinfo-t TYPE*(*'a*)) (*ptr-val p*)
⟨*proof*⟩

lemma *typ-slices-u-length* [*simp*]: *length* (*typ-slices-u t*) = *size-td t*

⟨*proof*⟩

lemma *typ-slices-u-index* [*simp*]:
n < *size-td t* ⟹ *typ-slices-u t* ! *n* = *list-map* (*typ-slice-t t n*)
⟨*proof*⟩

lemma *valid-root-footprint-ptr-force-type-u*:
wf-size-desc t ⟹ *size-td t* < *addr-card* ⟹
valid-root-footprint (*ptr-force-type-u t a h*) *a t*
⟨*proof*⟩

lemma *valid-root-footprint-ptr-force-type-u-size*:
wf-size-desc t ⟹ *size-td t* < *addr-card* ⟹
valid-root-footprint (*ptr-force-type-u t a h*) *a t*
⟨*proof*⟩

definition
stack-alloc:: *addr set* ⇒ *'a*:: *mem-type itself* ⇒ *heap-typ-desc* ⇒ (*'a ptr* × *heap-typ-desc*) *set* **where**
‹*stack-alloc* $\mathcal{S}$ *T d* = {
(*p*::*'a ptr*, *d'*).
ptr-span p ⊆ $\mathcal{S}$ ∧
typ-uinfo-t (*TYPE*(*'a*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*)) ∧
(∀ *a* ∈ *ptr-span p*. *root-ptr-valid d* (*PTR* (*stack-byte*) *a*)) ∧
ptr-aligned p ∧
d' = *ptr-force-type p d*
}›

definition
stack-allocs:: *nat* ⇒ *addr set* ⇒ *'a*:: *mem-type itself* ⇒ *heap-typ-desc* ⇒ (*'a ptr* × *heap-typ-desc*) *set* **where**
‹*stack-allocs n* $\mathcal{S}$ *T d* = {
(*p*::*'a ptr*, *d'*).
0 < *n* ∧
(∀ *i* < *n*. *ptr-span* (*p* +$_p$ *int i*) ⊆ $\mathcal{S}$) ∧
typ-uinfo-t (*TYPE*(*'a*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*)) ∧
(∀ *a* ∈ {*ptr-val p* ..+ *n* ∗ *size-of TYPE*(*'a*)} . *root-ptr-valid d* (*PTR* (*stack-byte*) *a*)) ∧
ptr-aligned p ∧
d' = *fold* (λ*i*. *ptr-force-type* (*p* +$_p$ *int i*)) [*0*..<*n*] *d*
}›

lemma *stack-alloc-stack-allocs-conv*: *stack-alloc* = *stack-allocs 1*
⟨*proof*⟩

lemma *htd-update-list-other*:
assumes *bound*: *length xs* < *addr-card*

assumes *notin*: $x \notin \{p..+length\ xs\}$
shows *htd-update-list p xs d x = d x*
⟨*proof*⟩

lemma *dom-typ-slice-t-stack-byte*:
dom (list-map (typ-slice-t (typ-uinfo-t TYPE(stack-byte)) n)) = {0}
⟨*proof*⟩

lemma *htd-update-list-same'*:
$[\![0 < unat\ k;\ unat\ k \leq addr\text{-}card - length\ v]\!] \Longrightarrow$ *htd-update-list (p + k) v h p = h p*
⟨*proof*⟩

lemma *dom-htd-update-list*:
assumes *xs-bound*: *length xs < addr-card*
assumes *n-bound*: *n < length xs*
shows *dom (snd (htd-update-list a xs d (a + word-of-nat n))) =*
dom (snd (d (a + word-of-nat n))) ∪ dom (xs ! n)
⟨*proof*⟩

lemma *dom-ptr-retyp*:
fixes *p::'a::mem-type ptr*
assumes *n*: *n < size-of TYPE('a)*
shows *dom (snd (ptr-retyp p d (ptr-val p + of-nat n))) =*
dom (snd (d (ptr-val p + of-nat n))) ∪
{0..<length (typ-slice-t (typ-uinfo-t TYPE('a)) n)}
⟨*proof*⟩

lemma *length-typ-slice-t*: *0 < length (typ-slice-t t n)*
⟨*proof*⟩

lemma *valid-root-footprint-retyp-stack'*:
fixes *p::'a::mem-type ptr*
assumes *stack*: *∀ a ∈ ptr-span p. valid-root-footprint d a (typ-uinfo-t TYPE(stack-byte))*
shows *valid-root-footprint(ptr-retyp p d) (ptr-val p) (typ-uinfo-t TYPE('a))*
⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-force-type-valid-footprint*:
valid-footprint (ptr-force-type p d) (ptr-val (p::'a ptr)) (typ-uinfo-t TYPE('a))
⟨*proof*⟩

lemma *ptr-force-type-valid-footprint*:
valid-footprint (ptr-force-type p d) (ptr-val (p::'a::mem-type ptr)) (typ-uinfo-t TYPE('a))
⟨*proof*⟩

lemma *valid-root-footprint-retyp-stack*:
fixes *p::'a::mem-type ptr*

assumes *stack*: $\forall\, a \in$ *ptr-span p*. *valid-root-footprint d a* (*typ-uinfo-t TYPE*(*stack-byte*))
shows *valid-root-footprint*(*ptr-force-type p d*) (*ptr-val p*) (*typ-uinfo-t TYPE*('*a*))
⟨*proof*⟩

lemma *root-ptr-valid-retyp-stack'*:
fixes *p*::'*a*::*mem-type ptr*
assumes *stack*: $\forall\, a \in$ *ptr-span p*. *root-ptr-valid d* (*PTR* (*stack-byte*) *a*)
assumes *aligned*: *ptr-aligned p*
shows *root-ptr-valid* (*ptr-retyp p d*) *p*
⟨*proof*⟩

lemma *root-ptr-valid-retyp-stack*:
fixes *p*::'*a*::*mem-type ptr*
assumes *stack*: $\forall\, a \in$ *ptr-span p*. *root-ptr-valid d* (*PTR* (*stack-byte*) *a*)
assumes *aligned*: *ptr-aligned p*
shows *root-ptr-valid* (*ptr-force-type p d*) *p*
⟨*proof*⟩

lemma *fold-ptr-retyp-other*:
fixes *p*::'*a*::*mem-type ptr*
assumes *a-notin*: $a \notin \{$*ptr-val p ..+ n* $*$ *size-of TYPE*('*a*)$\}$
shows (*fold* (λi. *ptr-retyp* (*p* $+_p$ *int i*)) [*0..<n*] *d*) *a* = *d a*
⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-force-type-d*:
$x \notin \{$*ptr-val* (*p*::'*a ptr*)*..+size-of TYPE*('*a*)$\} \Longrightarrow$
ptr-force-type p d x = *d x*
⟨*proof*⟩

lemma *fold-ptr-force-type-other*:
fixes *p*::'*a*::*mem-type ptr*
assumes *a-notin*: $a \notin \{$*ptr-val p ..+ n* $*$ *size-of TYPE*('*a*)$\}$
shows (*fold* (λi. *ptr-force-type* (*p* $+_p$ *int i*)) [*0..<n*] *d*) *a* = *d a*
⟨*proof*⟩

lemma *root-ptr-valid-domain*:
fixes *p*::'*a*::*mem-type ptr*
assumes *root-ptr-valid d p*
assumes $\bigwedge a.$ *a* $\in$ *ptr-span p* $\Longrightarrow$ *d' a* = *d a*
shows *root-ptr-valid d' p*
⟨*proof*⟩

lemma *root-ptr-valid-domain'*:
fixes *p*::'*a*::*mem-type ptr*
assumes $\bigwedge a.$ *a* $\in$ *ptr-span p* $\Longrightarrow$ *d' a* = *d a*
shows *root-ptr-valid d' p* = *root-ptr-valid d p*
⟨*proof*⟩

lemma *root-ptr-valid-range-not-NULL*:

root-ptr-valid htd (*p* :: (*'a* :: *c-type*) *ptr*)
⟹ *0* ∉ {*ptr-val p ..+ size-of TYPE*(*'a*)}
⟨*proof*⟩

lemma *intvl-no-overflow-nat'*:
assumes *no-overflow*: *unat a + b* ≤ *2* ^ *len-of TYPE*(*'a*::*len*)
shows (*x* ∈ {(*a* :: *'a word*) *..+ b*}) = (*unat a* ≤ *unat x* ∧ *unat x* < (*unat a* + *b*))
⟨*proof*⟩

lemma *intvl-no-overflow-nat*:
assumes *no-overflow*: *unat a + b* ≤ *addr-card*
shows (*x* ∈ {(*a* :: *addr-bitsize word*) *..+ b*}) = (*unat a* ≤ *unat x* ∧ *unat x* < (*unat a* + *b*))
⟨*proof*⟩

lemma *intvl-no-overflow-nat-conv*:
assumes *no-overflow*: *unat a + b* ≤ *addr-card*
shows {(*a* :: *addr-bitsize word*) *..+ b*} = {*x*. (*unat a* ≤ *unat x* ∧ *unat x* < (*unat a* + *b*))}
⟨*proof*⟩

lemma *zero-not-in-intvl-no-overflow*:
0 ∉ {*a* :: *'a*::*len word ..+ b*} ⟹ *unat a + b* ≤ *2* ^ *len-of TYPE*(*'a*)
⟨*proof*⟩

lemma *root-ptr-valid-last-byte-no-overflow*:
root-ptr-valid htd (*p* :: (*'a* :: *c-type*) *ptr*)
⟹ *unat* (*ptr-val p*) + *size-of TYPE*(*'a*) ≤ *2* ^ *len-of TYPE*(*addr-bitsize*)
⟨*proof*⟩

lemma *root-ptr-valid-retyps-stack'*:
fixes *p*::*'a*::*mem-type ptr*
assumes *stack*: ∀ *a* ∈ {*ptr-val p ..+ n* ∗ *size-of TYPE*(*'a*)}. *root-ptr-valid d* (*PTR* (*stack-byte*) *a*)
assumes *aligned*: *ptr-aligned p*
assumes *i-bound*: *i* < *n*
shows *root-ptr-valid* (*fold* (λ*i*. *ptr-retyp* (*p* +$_p$ *int i*)) [*0*..<*n*] *d*) (*p* +$_p$ *int i*)
⟨*proof*⟩

lemma *root-ptr-valid-retyps-stack*:
fixes *p*::*'a*::*mem-type ptr*
assumes *stack*: ∀ *a* ∈ {*ptr-val p ..+ n* ∗ *size-of TYPE*(*'a*)}. *root-ptr-valid d* (*PTR* (*stack-byte*) *a*)
assumes *aligned*: *ptr-aligned p*
assumes *i-bound*: *i* < *n*
shows *root-ptr-valid* (*fold* (λ*i*. *ptr-force-type* (*p* +$_p$ *int i*)) [*0*..<*n*] *d*) (*p* +$_p$ *int i*)
⟨*proof*⟩

definition *stack-free* :: *heap-typ-desc* ⇒ *addr set* **where**
stack-free d = {*a*. *root-ptr-valid d* (*PTR* (*stack-byte*) *a*)}

lemma *stack-alloc-cases* [*consumes 1*]:
fixes *p*::'*a*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-alloc* 𝒮 *T d*
assumes *dest*:
⟦*typ-uinfo-t* (*TYPE*('*a*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*));
ptr-span p ⊆ 𝒮;
∀ *a* ∈ *ptr-span p*. *root-ptr-valid d* (*PTR* (*stack-byte*) *a*);
ptr-aligned p; *c-guard p*; *root-ptr-valid d' p*;
d' = *ptr-force-type p d*⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *stack-allocs-cases* [*consumes 1*]:
fixes *p*::'*a*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-allocs n* 𝒮 *T d*
assumes *dest*:
⟦*0* < *n*; *0* ∉ {*ptr-val p* ..+ *n* * *size-of TYPE*('*a*)}; *unat* (*ptr-val p*) + *n* * *size-of TYPE*('*a*) ≤ *addr-card*;
typ-uinfo-t (*TYPE*('*a*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*));
∀ *i* < *n*. *ptr-span* (*p* +$_p$ *int i*) ⊆ 𝒮;
∀ *a* ∈ {*ptr-val p* ..+ *n* * *size-of TYPE*('*a*)}. *root-ptr-valid d* (*PTR* (*stack-byte*) *a*);
{*ptr-val p* ..+ *n* * *size-of TYPE*('*a*)} ⊆ *stack-free d*;
ptr-aligned p; *c-guard p*; *root-ptr-valid d' p*;
∀ *i* < *n*. *ptr-aligned* (*p* +$_p$ *int i*); ∀ *i* < *n*. *c-guard* (*p* +$_p$ *int i*); ∀ *i* < *n*. *root-ptr-valid d'* (*p* +$_p$ *int i*);
d' = *fold* (λ*i*. *ptr-force-type* (*p* +$_p$ *int i*)) [*0*..<*n*] *d*⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *stack-allocs-no-overflow*:
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-allocs n* 𝒮 (*TYPE*('*a*::*mem-type*)) *d*
shows *unat* (*ptr-val p*) + *n* * *size-of TYPE*('*a*) ≤ *addr-card*
⟨*proof*⟩

lemma *stack-alloc-ptr-force-type*: (*p*, *d'*) ∈ *stack-alloc* 𝒮 *T d* ⟹
d' = *ptr-force-type p d*
⟨*proof*⟩

lemma *stack-allocs-ptr-force-type*: (*p*, *d'*) ∈ *stack-allocs n* 𝒮 *T d* ⟹
d' = *fold* (λ*i*. *ptr-force-type* (*p* +$_p$ *int i*)) [*0*..<*n*] *d*
⟨*proof*⟩

lemma *stack-allocs-no-stack-byte*: (*p*::'*a*::*mem-type ptr*, *d'*) ∈ *stack-allocs n* 𝒮 *T d*

$\Longrightarrow$ *typ-uinfo-t* (*TYPE*('*a*)) $\neq$ *typ-uinfo-t* (*TYPE*(*stack-byte*))
⟨*proof*⟩

lemma *stack-allocs-*$\mathcal{S}$: (*p*::'*a*::*mem-type ptr*, *d'*) $\in$ *stack-allocs n* $\mathcal{S}$ *T d* $\Longrightarrow$ *i* < *n*
$\Longrightarrow$ *ptr-span* (*p* +$_p$ *int i*) $\subseteq$ $\mathcal{S}$
⟨*proof*⟩

lemma *ptr-retyp-split*: *ptr-retyp* (*p*::'*a*::*mem-type ptr*) *d a* =
(*if a* $\in$ *ptr-span p then*
(*True*, *snd* (*d a*) ++ *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*('*a*)) (*unat* (*a* − *ptr-val p*))))
else d a)
⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-force-type-footprint*:
x $\in$ {*ptr-val p*..+*size-of TYPE*('*a*)} $\Longrightarrow$
ptr-force-type (*p*::'*a ptr*) *d x* =
(*True*,*list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*('*a*)) (*unat* (*x* − *ptr-val p*))))
⟨*proof*⟩

lemma *ptr-force-type-split*: *ptr-force-type* (*p*::'*a*::*mem-type ptr*) *d a* =
(*if a* $\in$ *ptr-span p then*
(*True*, *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*('*a*)) (*unat* (*a* − *ptr-val p*))))
else d a)
⟨*proof*⟩

lemma *in-intvl-Suc*: *x* $\in$ {*x*..+*Suc n*}
⟨*proof*⟩

definition *zero-heap*:: *heap-mem* **where**
zero-heap = (λ*a*. *zero-class.zero*)

definition *stack-byte-typing*::*heap-typ-desc* **where**
stack-byte-typing = (λ*a*. *ptr-force-type* (*PTR*(*stack-byte*) *a*) *empty-htd a*)

definition *stack-release*:: '*a*::*mem-type ptr* ⇒ *heap-typ-desc* ⇒ *heap-typ-desc* **where**

stack-release p d = *override-on d stack-byte-typing* (*ptr-span p*)

definition *stack-releases*:: *nat* ⇒ '*a*::*mem-type ptr* ⇒ *heap-typ-desc* ⇒ *heap-typ-desc* **where**
stack-releases n p d = *override-on d stack-byte-typing* {*ptr-val p* ..+ *n* ∗ *size-of TYPE*('*a*)}

lemma *stack-release-stack-releases-conv*: *stack-release* = *stack-releases 1*
⟨*proof*⟩

lemma *stack-releases-0* [*simp*]: *stack-releases 0 d* = *id*
⟨*proof*⟩

lemma *stack-release-stack-alloc-inverse*: $(p,\ d') \in \mathit{stack\text{-}alloc}\ \mathcal{S}\ T\ d \Longrightarrow \mathit{stack\text{-}release}\ p\ d' = d$
⟨*proof*⟩

lemma *stack-releases-stack-allocs-inverse*: $(p,\ d') \in \mathit{stack\text{-}allocs}\ n\ \mathcal{S}\ T\ d \Longrightarrow \mathit{stack\text{-}releases}\ n\ p\ d' = d$
⟨*proof*⟩

lemma *sub-typ-stack-byte*:
$\mathit{TYPE}('b::\mathit{c\text{-}type}) \leq_\tau \mathit{TYPE}(\mathit{stack\text{-}byte}) \Longrightarrow \mathit{typ\text{-}uinfo\text{-}t}\ \mathit{TYPE}('b) = \mathit{typ\text{-}uinfo\text{-}t}\ \mathit{TYPE}(\mathit{stack\text{-}byte})$
⟨*proof*⟩

lemma *root-ptr-valid-not-subtype-disjoint*:
$[\![\ \mathit{root\text{-}ptr\text{-}valid}\ d\ (p::'a::\mathit{mem\text{-}type}\ \mathit{ptr});$
$d \models_t (q::'b::\mathit{mem\text{-}type}\ \mathit{ptr});$
$\neg\ \mathit{TYPE}('b) \leq_\tau \mathit{TYPE}('a)\ ]\!] \Longrightarrow$
$\mathit{ptr\text{-}span}\ p \cap \mathit{ptr\text{-}span}\ q = \{\}$
⟨*proof*⟩

lemma *stack-alloc-disjoint*:
fixes $q::'b::\mathit{mem\text{-}type}\ \mathit{ptr}$
assumes *stack-alloc*: $(p,\ d') \in \mathit{stack\text{-}alloc}\ \mathcal{S}\ (\mathit{TYPE}('a::\mathit{mem\text{-}type}))\ d$
assumes *no-stack*: $\mathit{typ\text{-}uinfo\text{-}t}\ (\mathit{TYPE}('b)) \neq \mathit{typ\text{-}uinfo\text{-}t}\ (\mathit{TYPE}(\mathit{stack\text{-}byte}))$
assumes *typed*: $d \models_t q$
shows $\mathit{ptr\text{-}span}\ p \cap \mathit{ptr\text{-}span}\ q = \{\}$
⟨*proof*⟩

lemma *stack-allocs-disjoint*:
fixes $q::'b::\mathit{mem\text{-}type}\ \mathit{ptr}$
assumes *stack-alloc*: $(p,\ d') \in \mathit{stack\text{-}allocs}\ n\ \mathcal{S}\ (\mathit{TYPE}('a::\mathit{mem\text{-}type}))\ d$
assumes *no-stack*: $\mathit{typ\text{-}uinfo\text{-}t}\ (\mathit{TYPE}('b)) \neq \mathit{typ\text{-}uinfo\text{-}t}\ (\mathit{TYPE}(\mathit{stack\text{-}byte}))$
assumes *typed*: $d \models_t q$
shows $\{\mathit{ptr\text{-}val}\ p\ ..+\ n * \mathit{size\text{-}of}\ \mathit{TYPE}('a)\} \cap \mathit{ptr\text{-}span}\ q = \{\}$
⟨*proof*⟩

lemma *stack-allocs-contained*:
assumes *stack-alloc*: $(p,\ d') \in \mathit{stack\text{-}allocs}\ n\ \mathcal{S}\ (\mathit{TYPE}('a::\mathit{mem\text{-}type}))\ d$
assumes *i*: $i < n$
shows $\mathit{ptr\text{-}span}\ (p +_p \mathit{int}\ i) \subseteq \{\mathit{ptr\text{-}val}\ p\ ..+\ n * \mathit{size\text{-}of}\ \mathit{TYPE}('a)\}$
⟨*proof*⟩

lemma *stack-allocs-disjoint'*:
fixes $q::'b::\mathit{mem\text{-}type}\ \mathit{ptr}$

assumes *stack-alloc*: $(p,\ d') \in \mathit{stack\text{-}allocs}\ n\ \mathcal{S}\ (\mathit{TYPE}('a{::}\mathit{mem\text{-}type}))\ d$
assumes *no-stack*: $\mathit{typ\text{-}uinfo\text{-}t}\ (\mathit{TYPE}('b)) \neq \mathit{typ\text{-}uinfo\text{-}t}\ (\mathit{TYPE}(\mathit{stack\text{-}byte}))$
assumes *typed*: $d \models_t q$
assumes *i*: $i < n$
shows $\mathit{ptr\text{-}span}\ (p +_p \mathit{int}\ i) \cap \mathit{ptr\text{-}span}\ q = \{\}$
⟨*proof*⟩

lemma *ptr-retyp-valid-footprint-disjoint2*:
$[\![\mathit{valid\text{-}footprint}\ (\mathit{ptr\text{-}retyp}\ (q{::}'b{::}\mathit{mem\text{-}type}\ \mathit{ptr})\ d)\ p\ s;\ \{p..{+}\mathit{size\text{-}td}\ s\} \cap \{\mathit{ptr\text{-}val}\ q..{+}\mathit{size\text{-}of}\ \mathit{TYPE}('b)\} = \{\}\ ]\!]$
$\Longrightarrow \mathit{valid\text{-}footprint}\ d\ p\ s$
⟨*proof*⟩

lemma *ptr-force-type-valid-footprint-disjoint2*:
$[\![\mathit{valid\text{-}footprint}\ (\mathit{ptr\text{-}force\text{-}type}\ (q{::}'b{::}\mathit{mem\text{-}type}\ \mathit{ptr})\ d)\ p\ s;\ \{p..{+}\mathit{size\text{-}td}\ s\} \cap \{\mathit{ptr\text{-}val}\ q..{+}\mathit{size\text{-}of}\ \mathit{TYPE}('b)\} = \{\}\ ]\!]$
$\Longrightarrow \mathit{valid\text{-}footprint}\ d\ p\ s$
⟨*proof*⟩

lemma *ptr-span-no-overflow-split-last-disjoint*:
fixes $p{::}'a{::}\mathit{mem\text{-}type}\ \mathit{ptr}$
assumes *no-overflow*: $0 \notin \{\mathit{ptr\text{-}val}\ p..{+}\mathit{Suc}\ n * \mathit{size\text{-}of}\ \mathit{TYPE}('a)\}$
shows $\mathit{ptr\text{-}span}\ (p +_p \mathit{int}\ n) \cap \{\mathit{ptr\text{-}val}\ p..{+}n * \mathit{size\text{-}of}\ \mathit{TYPE}('a)\} = \{\}$
⟨*proof*⟩

lemma *ptr-span-no-overflow-indexes-disjoint*:
fixes $p{::}'a{::}\mathit{mem\text{-}type}\ \mathit{ptr}$
assumes *no-overflow*: $0 \notin \{\mathit{ptr\text{-}val}\ p..{+}\ n * \mathit{size\text{-}of}\ \mathit{TYPE}('a)\}$
assumes *i*: $i < n$
assumes *j*: $j < n$
assumes *neq*: $i \neq j$
shows $\mathit{ptr\text{-}span}\ (p +_p \mathit{int}\ i) \cap \mathit{ptr\text{-}span}\ (p +_p \mathit{int}\ j) = \{\}$
⟨*proof*⟩

lemma *array-to-index-span*:
assumes *x-in*: $x \in \{\mathit{ptr\text{-}val}\ (p{::}'a{::}\mathit{mem\text{-}type}\ \mathit{ptr})..{+}\ n * \mathit{size\text{-}of}\ \mathit{TYPE}('a)\}$
shows $\exists\, i.\ i < n \wedge x \in \mathit{ptr\text{-}span}\ (p +_p \mathit{int}\ i)$
⟨*proof*⟩

lemma *array-to-index-span-exact*:
assumes *x-in*: $x \in \{\mathit{ptr\text{-}val}\ (p{::}'a{::}\mathit{mem\text{-}type}\ \mathit{ptr})..{+}\ n * \mathit{size\text{-}of}\ \mathit{TYPE}('a)\}$
shows $x \in \mathit{ptr\text{-}span}\ (p +_p \mathit{int}\ ((\mathit{unat}\ (x - \mathit{ptr\text{-}val}\ p))\ \mathit{div}\ \mathit{size\text{-}of}\ \mathit{TYPE}('a)))$
⟨*proof*⟩

lemma *array-index-span-conv*:
$\{\mathit{ptr\text{-}val}\ (p{::}'a{::}\mathit{mem\text{-}type}\ \mathit{ptr})..{+}\ n * \mathit{size\text{-}of}\ \mathit{TYPE}('a)\} = (\bigcup i{<}n.\ \mathit{ptr\text{-}span}\ (p +_p \mathit{int}\ i))$
⟨*proof*⟩

lemma *fold-ptr-retyp-footprint*:
fixes *p*::*'a*::*mem-type ptr*
assumes *no-overflow*: *0* ∉ {*ptr-val p..+ n* ∗ *size-of TYPE*(*'a*)}
assumes *i*: *i* < *n*
assumes *x*: *x* ∈ *ptr-span* (*p* +*p int i*)
shows *fold* (λ*i*. *ptr-retyp* (*p* +*p int i*)) [*0*..<*n*] *d x* =
(*True*, *snd* (*d x*) ++ *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*(*'a*)) (*unat* (*x* − *ptr-val* (*p* +*p int i*)))))
⟨*proof*⟩

lemma *ptr-retyp-idem*:
ptr-retyp p (*ptr-retyp* (*p*::*'a*::*mem-type ptr*) *d*) = *ptr-retyp p d*
⟨*proof*⟩

lemma *fold-ptr-retyp-d-empty*:
fixes *p*::*'a*::*mem-type ptr*
assumes *no-overflow*: *0* ∉ {*ptr-val p..+ n* ∗ *size-of TYPE*(*'a*)}
assumes *i*: *i* < *n*
assumes *x*: *x* ∈ *ptr-span* (*p* +*p int i*)
shows *fold* (λ*i*. *ptr-retyp* (*p* +*p int i*)) [*0*..<*n*] *d x* =
(*True*, *snd* (*d x*) ++ *snd* (*ptr-retyp* (*p* +*p int i*) *empty-htd x*))
⟨*proof*⟩

lemma *fold-ptr-retyp-eq-fst*:
assumes *no-overflow*: *0* ∉ {*ptr-val p..+ n* ∗ *size-of TYPE*(*'a*)}
shows*fst* (*fold* (λ*i*. *ptr-retyp* (*p* +*p int i*)) [*0*..<*n*] *d x*) =
(*if x* ∈ {*ptr-val* (*p*::*'a*::*mem-type ptr*)*..+ n* ∗ *size-of TYPE*(*'a*)} *then True else fst* (*d x*))
⟨*proof*⟩

lemma *fold-ptr-retyp-valid-footprint-disjoint2*:
assumes *no-overflow*: *0* ∉ {*ptr-val q..+ n* ∗ *size-of TYPE*(*'b*)}
shows ⟦*valid-footprint* (*fold* (λ*i*. *ptr-retyp* ((*q*::*'b*::*mem-type ptr*) +*p int i*)) [*0*..<*n*] *d*) *p s*;
{*p..+size-td s*} ∩ {*ptr-val q ..+ n* ∗ *size-of TYPE*(*'b*)} = {} ⟧
⟹ *valid-footprint d p s*
⟨*proof*⟩

lemma *ptr-retyp-disjoint2*:
⟦*ptr-retyp* (*p*::*'a*::*mem-type ptr*) *d*,*g* ⊨*t q*;
ptr-span p ∩ *ptr-span q* = {} ⟧
⟹ *d*,*g* ⊨*t* (*q*::*'b*::*mem-type ptr*)
⟨*proof*⟩

lemma *fold-ptr-retyp-disjoint2*:
fixes *p*::*'a*::*mem-type ptr*

assumes *no-overflow*: $0 \notin \{ptr\text{-}val\ p..+\ n * size\text{-}of\ TYPE('a)\}$
shows $[\![fold\ (\lambda i.\ ptr\text{-}retyp\ (p +_p\ int\ i))\ [0..<n]\ d,g \models_t\ q;$
$\{ptr\text{-}val\ p..+\ n * size\text{-}of\ TYPE('a)\} \cap ptr\text{-}span\ q = \{\}\ ]\!]$
$\Longrightarrow d,g \models_t (q::'b::mem\text{-}type\ ptr)$
⟨*proof*⟩

lemma *ptr-retyp-disjoint-iff*:
$\{ptr\text{-}val\ p..+size\text{-}of\ TYPE('a)\} \cap \{ptr\text{-}val\ q..+size\text{-}of\ TYPE('b)\} = \{\}$
$\Longrightarrow ptr\text{-}retyp\ (p::'a::mem\text{-}type\ ptr)\ d,g \models_t\ q = d,g \models_t (q::'b::mem\text{-}type\ ptr)$
⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-force-type-valid-footprint-disjoint*:
$[\![\ valid\text{-}footprint\ d\ p\ s; \{p..+size\text{-}td\ s\} \cap \{ptr\text{-}val\ q..+size\text{-}of\ TYPE('a)\} = \{\}\ ]\!]$
$\Longrightarrow valid\text{-}footprint\ (ptr\text{-}force\text{-}type\ (q::'a\ ptr)\ d)\ p\ s$
⟨*proof*⟩

lemma *ptr-force-type-disjoint*:
$[\![\ d,g \models_t (q::'b::mem\text{-}type\ ptr); \{ptr\text{-}val\ p..+size\text{-}of\ TYPE('a)\} \cap$
$\{ptr\text{-}val\ q..+size\text{-}of\ TYPE('b)\} = \{\}\ ]\!] \Longrightarrow$
$ptr\text{-}force\text{-}type\ (p::'a::mem\text{-}type\ ptr)\ d,g \models_t\ q$
⟨*proof*⟩

lemma *ptr-force-type-disjoint2*:
$[\![ptr\text{-}force\text{-}type\ (p::'a::mem\text{-}type\ ptr)\ d,g \models_t\ q;$
$ptr\text{-}span\ p \cap ptr\text{-}span\ q = \{\}\ ]\!]$
$\Longrightarrow d,g \models_t (q::'b::mem\text{-}type\ ptr)$
⟨*proof*⟩

lemma *ptr-force-type-disjoint-iff*:
$\{ptr\text{-}val\ p..+size\text{-}of\ TYPE('a)\} \cap \{ptr\text{-}val\ q..+size\text{-}of\ TYPE('b)\} = \{\}$
$\Longrightarrow ptr\text{-}force\text{-}type\ (p::'a::mem\text{-}type\ ptr)\ d,g \models_t\ q = d,g \models_t (q::'b::mem\text{-}type\ ptr)$
⟨*proof*⟩

lemma *fold-ptr-force-type-valid-footprint-disjoint2*:
assumes *no-overflow*: $0 \notin \{ptr\text{-}val\ q..+\ n * size\text{-}of\ TYPE('b)\}$
shows $[\![valid\text{-}footprint\ (fold\ (\lambda i.\ ptr\text{-}force\text{-}type\ ((q::'b::mem\text{-}type\ ptr) +_p\ int\ i))$
$[0..<n]\ d)\ p\ s;$
$\{p..+size\text{-}td\ s\} \cap \{ptr\text{-}val\ q\ ..+\ n * size\text{-}of\ TYPE('b)\} = \{\}\ ]\!]$
$\Longrightarrow valid\text{-}footprint\ d\ p\ s$
⟨*proof*⟩

lemma *fold-ptr-force-type-disjoint2*:
fixes $p::'a::mem\text{-}type\ ptr$
assumes *no-overflow*: $0 \notin \{ptr\text{-}val\ p..+\ n * size\text{-}of\ TYPE('a)\}$
shows $[\![fold\ (\lambda i.\ ptr\text{-}force\text{-}type\ (p +_p\ int\ i))\ [0..<n]\ d,g \models_t\ q;$
$\{ptr\text{-}val\ p..+\ n * size\text{-}of\ TYPE('a)\} \cap ptr\text{-}span\ q = \{\}\ ]\!]$
$\Longrightarrow d,g \models_t (q::'b::mem\text{-}type\ ptr)$
⟨*proof*⟩

lemma *fold-ptr-retyp-valid-footprint-disjoint*:
⟦ *valid-footprint d p s*; {*p*..+*size-td s*} ∩ {*ptr-val q* ..+ *n* ∗ *size-of TYPE*('*b*)} = {} ⟧
⟹ *valid-footprint* (*fold* (λ*i*. *ptr-retyp* ((*q*::'*b*:: *mem-type ptr*) +$_p$ *int i*)) [0..<*n*] *d*) *p s*
⟨*proof*⟩

lemma *fold-ptr-force-type-valid-footprint-disjoint*:
⟦ *valid-footprint d p s*; {*p*..+*size-td s*} ∩ {*ptr-val q* ..+ *n* ∗ *size-of TYPE*('*b*)} = {} ⟧
⟹ *valid-footprint* (*fold* (λ*i*. *ptr-force-type* ((*q*::'*b*:: *mem-type ptr*) +$_p$ *int i*)) [0..<*n*] *d*) *p s*
⟨*proof*⟩

lemma *fold-ptr-retyp-disjoint*:
fixes *p*::'*a*::*mem-type ptr*
shows ⟦ *d*,*g* ⊨$_t$ (*q*::'*b*::*mem-type ptr*); {*ptr-val p*..+ *n* ∗ *size-of TYPE*('*a*)} ∩ *ptr-span q* = {} ⟧ ⟹
fold (λ*i*. *ptr-retyp* (*p* +$_p$ *int i*)) [0..<*n*] *d*,*g* ⊨$_t$ *q*
⟨*proof*⟩

lemma *fold-ptr-force-type-disjoint*:
fixes *p*::'*a*::*mem-type ptr*
shows ⟦ *d*,*g* ⊨$_t$ (*q*::'*b*::*mem-type ptr*); {*ptr-val p*..+ *n* ∗ *size-of TYPE*('*a*)} ∩ *ptr-span q* = {} ⟧ ⟹
fold (λ*i*. *ptr-force-type* (*p* +$_p$ *int i*)) [0..<*n*] *d*,*g* ⊨$_t$ *q*
⟨*proof*⟩

lemma *fold-ptr-retyp-disjoint-iff*:
fixes *p*::'*a*::*mem-type ptr*
assumes *no-overflow*: 0 ∉ {*ptr-val p*..+ *n* ∗ *size-of TYPE*('*a*)}
shows {*ptr-val p*..+ *n* ∗ *size-of TYPE*('*a*)} ∩ *ptr-span q* = {}
⟹ *fold* (λ*i*. *ptr-retyp* (*p* +$_p$ *int i*)) [0..<*n*] *d*,*g* ⊨$_t$ *q* = *d*,*g* ⊨$_t$ (*q*::'*b*::*mem-type ptr*)
⟨*proof*⟩

lemma *fold-ptr-force-type-disjoint-iff*:
fixes *p*::'*a*::*mem-type ptr*
assumes *no-overflow*: 0 ∉ {*ptr-val p*..+ *n* ∗ *size-of TYPE*('*a*)}
shows {*ptr-val p*..+ *n* ∗ *size-of TYPE*('*a*)} ∩ *ptr-span q* = {}
⟹ *fold* (λ*i*. *ptr-force-type* (*p* +$_p$ *int i*)) [0..<*n*] *d*,*g* ⊨$_t$ *q* = *d*,*g* ⊨$_t$ (*q*::'*b*::*mem-type ptr*)
⟨*proof*⟩

lemma *stack-alloc-preserves-typing*:
fixes *q*::'*b*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d*') ∈ *stack-alloc* $\mathcal{S}$ (*TYPE*('*a*::*mem-type*)) *d*
assumes *no-stack*: *typ-uinfo-t* (*TYPE*('*b*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*))

assumes *typed*: $d \models_t q$
shows $d' \models_t q$
⟨*proof*⟩

lemma *stack-allocs-preserves-typing*:
fixes *q*::'*b*::*mem-type ptr*
assumes *stack-alloc*: $(p, d') \in$ *stack-allocs n* $\mathcal{S}$ (*TYPE*('*a*::*mem-type*)) *d*
assumes *no-stack*: *typ-uinfo-t* (*TYPE*('*b*)) $\neq$ *typ-uinfo-t* (*TYPE*(*stack-byte*))
assumes *typed*: $d \models_t q$
shows $d' \models_t q$
⟨*proof*⟩

lemma *h-t-valid-valid-footprint*: *d*,*g* $\models_t p \Longrightarrow$ *valid-footprint d* (*ptr-val* (*p*::'*a*::*c-type ptr*)) (*typ-uinfo-t TYPE*('*a*))
⟨*proof*⟩

lemma *stack-alloc-preserves-root-ptr-valid*:
fixes *q*::'*b*::*mem-type ptr*
assumes *stack-alloc*: $(p, d') \in$ *stack-alloc* $\mathcal{S}$ (*TYPE*('*a*::*mem-type*)) *d*
assumes *no-stack*: *typ-uinfo-t* (*TYPE*('*b*)) $\neq$ *typ-uinfo-t* (*TYPE*(*stack-byte*))
assumes *typed*: *root-ptr-valid d q*
shows *root-ptr-valid d' q*
⟨*proof*⟩

lemma *stack-allocs-preserves-root-ptr-valid*:
fixes *q*::'*b*::*mem-type ptr*
assumes *stack-alloc*: $(p, d') \in$ *stack-allocs n* $\mathcal{S}$ (*TYPE*('*a*::*mem-type*)) *d*
assumes *no-stack*: *typ-uinfo-t* (*TYPE*('*b*)) $\neq$ *typ-uinfo-t* (*TYPE*(*stack-byte*))
assumes *typed*: *root-ptr-valid d q*
shows *root-ptr-valid d' q*
⟨*proof*⟩

lemma *stack-alloc-root-ptr-valid-new-cases*:
fixes *q*::'*b*::*mem-type ptr*
assumes *stack-alloc*: $(p, d') \in$ *stack-alloc* $\mathcal{S}$ (*TYPE*('*a*::*mem-type*)) *d*
assumes *root-ptr-valid d' q*
shows (*ptr-val p* = *ptr-val q* $\wedge$ *typ-uinfo-t* (*TYPE*('*b*)) = *typ-uinfo-t* (*TYPE*('*a*))) $\vee$ *root-ptr-valid d q*
⟨*proof*⟩

lemma *valid-root-footprints-no-overlap*:
assumes *valid-root-footprint d a1 t1*
assumes *valid-root-footprint d a2 t2*
assumes *t1* $\neq$ *t2*
shows {*a1* ..+ *size-td t1*} $\cap$ {*a2* ..+ *size-td t2*} = {}
⟨*proof*⟩

lemma *root-ptr-valid-neq-disjoint*:
⟦ *root-ptr-valid d* (*p*::'*a*::*c-type ptr*);

root-ptr-valid d (q::'b::c-type ptr);
ptr-val p ≠ ptr-val q ⟧ ⟹
{ptr-val p..+size-of TYPE('a)} ∩
{ptr-val q..+size-of TYPE('b)} = {}
⟨*proof*⟩

lemma *root-ptr-valid-same-type-neq-disjoint*:
root-ptr-valid d p ⟹ root-ptr-valid d q ⟹ p ≠ q = (ptr-span p ∩ ptr-span q = {})
⟨*proof*⟩

lemma *cvalid-same-type-neq-disjoint*:
d ⊨ₜ p ⟹ d ⊨ₜ q ⟹ p ≠ q = (ptr-span p ∩ ptr-span q = {})
⟨*proof*⟩

lemma *root-ptr-valid-type-neq-disjoint*:
⟦ root-ptr-valid d (p::'a::c-type ptr);
root-ptr-valid d (q::'b::c-type ptr);
typ-uinfo-t TYPE('a) ≠ typ-uinfo-t TYPE('b) ⟧ ⟹
{ptr-val p..+size-of TYPE('a)} ∩
{ptr-val q..+size-of TYPE('b)} = {}
⟨*proof*⟩

lemma *valid-root-footprints-cases*:
assumes *valid-root-footprint d a1 t1*
assumes *valid-root-footprint d a2 t2*
shows *(t1 = t2 ∧ a1 = a2) ∨ ({a1 ..+ size-td t1} ∩ {a2 ..+ size-td t2} = {})*
⟨*proof*⟩

lemma *root-ptr-valid-cases*:
fixes *p::'a::mem-type ptr*
fixes *q:: 'b::mem-type ptr*
assumes *root-p*: *root-ptr-valid d p*
assumes *root-q*: *root-ptr-valid d q*
shows *(ptr-val p = ptr-val q ∧ typ-uinfo-t (TYPE('a)) = typ-uinfo-t (TYPE('b))) ∨*
(ptr-span p ∩ ptr-span q) = {}
⟨*proof*⟩

lemma *root-ptr-valid-casesE* [*consumes 2*]:
fixes *p::'a::mem-type ptr*
fixes *q:: 'b::mem-type ptr*
assumes *root-p*: *root-ptr-valid d p*
assumes *root-q*: *root-ptr-valid d q*
assumes *same*: *(ptr-val q = ptr-val p ∧ typ-uinfo-t (TYPE('a)) = typ-uinfo-t (TYPE('b))) ⟹ P*
assumes *disj*: *ptr-span p ∩ ptr-span q = {} ⟹ P*
shows *P*

⟨*proof*⟩

lemma *stack-allocs-root-ptr-valid-new-cases*:
fixes *q*::'*b*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-allocs n* $\mathcal{S}$ (*TYPE*('*a*::*mem-type*)) *d*
assumes *root-ptr-valid d' q*
shows (∃ *i*<*n*. *ptr-val q* = *ptr-val* (*p* +$_p$ *int i*) ∧ *typ-uinfo-t* (*TYPE*('*b*)) = *typ-uinfo-t* (*TYPE*('*a*))) ∨ *root-ptr-valid d q*
⟨*proof*⟩

lemma *stack-alloc-root-ptr-valid-same*:
fixes *q*::'*b*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-alloc* $\mathcal{S}$ (*TYPE*('*a*::*mem-type*)) *d*
assumes *addr-eq*: *ptr-val p* = *ptr-val q*
assumes *match*: *typ-uinfo-t* (*TYPE*('*b*)) = *typ-uinfo-t* (*TYPE*('*a*))
shows *root-ptr-valid d' q*
⟨*proof*⟩

lemma *stack-allocs-root-ptr-valid-same*:
fixes *q*::'*b*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-allocs n* $\mathcal{S}$ (*TYPE*('*a*::*mem-type*)) *d*
assumes *i*: *i* < *n*
assumes *addr-eq*: *ptr-val q* = *ptr-val* (*p* +$_p$ *int i*)
assumes *match*: *typ-uinfo-t* (*TYPE*('*b*)) = *typ-uinfo-t* (*TYPE*('*a*))
shows *root-ptr-valid d' q*
⟨*proof*⟩

lemma *stack-alloc-root-ptr-valid-other*:
fixes *q*::'*b*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-alloc* $\mathcal{S}$ (*TYPE*('*a*::*mem-type*)) *d*
assumes *valid-d*: *root-ptr-valid d q*
assumes *non-stack*: *typ-uinfo-t* (*TYPE*('*b*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*))
shows *root-ptr-valid d' q*
⟨*proof*⟩

lemma *stack-allocs-root-ptr-valid-other*:
fixes *q*::'*b*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-allocs n* $\mathcal{S}$ (*TYPE*('*a*::*mem-type*)) *d*
assumes *valid-d*: *root-ptr-valid d q*
assumes *non-stack*: *typ-uinfo-t* (*TYPE*('*b*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*))
shows *root-ptr-valid d' q*
⟨*proof*⟩

lemma *stack-alloc-root-ptr-valid-cases*:
fixes *q*::'*b*::*mem-type ptr*
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-alloc* $\mathcal{S}$ (*TYPE*('*a*::*mem-type*)) *d*
assumes *non-stack-byte*: *typ-uinfo-t* (*TYPE*('*b*)) ≠ *typ-uinfo-t* (*TYPE*(*stack-byte*))
shows *root-ptr-valid d' q* ⟷

$(ptr\text{-}val\ p = ptr\text{-}val\ q \wedge typ\text{-}uinfo\text{-}t\ (TYPE('b)) = typ\text{-}uinfo\text{-}t\ (TYPE('a))) \vee$
$root\text{-}ptr\text{-}valid\ d\ q$

⟨*proof*⟩

lemma *stack-allocs-root-ptr-valid-cases*:
fixes $q::'b::mem\text{-}type\ ptr$
assumes *stack-alloc*: $(p,\ d') \in stack\text{-}allocs\ n\ \mathcal{S}\ (TYPE('a::mem\text{-}type))\ d$
assumes *non-stack-byte*: $typ\text{-}uinfo\text{-}t\ (TYPE('b)) \neq typ\text{-}uinfo\text{-}t\ (TYPE(stack\text{-}byte))$
shows $root\text{-}ptr\text{-}valid\ d'\ q \longleftrightarrow$
$(\exists i{<}n.\ ptr\text{-}val\ q = ptr\text{-}val\ (p +_p int\ i) \wedge typ\text{-}uinfo\text{-}t\ (TYPE('b)) = typ\text{-}uinfo\text{-}t\ (TYPE('a))) \vee$
$root\text{-}ptr\text{-}valid\ d\ q$

⟨*proof*⟩

lemma *stack-alloc-root-ptr-valid-same-type-cases*:
assumes *stack-alloc*: $(p,\ d') \in stack\text{-}alloc\ \mathcal{S}\ (TYPE('a::mem\text{-}type))\ d$
shows $root\text{-}ptr\text{-}valid\ d'\ q \longleftrightarrow p = q \vee root\text{-}ptr\text{-}valid\ d\ q$
⟨*proof*⟩

lemma *stack-allocs-root-ptr-valid-same-type-cases*:
assumes *stack-alloc*: $(p,\ d') \in stack\text{-}allocs\ n\ \mathcal{S}\ (TYPE('a::mem\text{-}type))\ d$
shows $root\text{-}ptr\text{-}valid\ d'\ q \longleftrightarrow (\exists i{<}n.\ q = (p +_p int\ i) \vee root\text{-}ptr\text{-}valid\ d\ q)$
⟨*proof*⟩

lemma *root-ptr-valid-valid-root-footprint*:
$root\text{-}ptr\text{-}valid\ d\ (p::'a\ ptr) \Longrightarrow valid\text{-}root\text{-}footprint\ d\ (ptr\text{-}val\ p)\ (typ\text{-}uinfo\text{-}t\ TYPE('a::c\text{-}type))$
⟨*proof*⟩

definition
$allocated\text{-}ptr::\ addr\ set \Rightarrow\ 'a::mem\text{-}type\ itself \Rightarrow\ heap\text{-}typ\text{-}desc \Rightarrow\ heap\text{-}typ\text{-}desc \Rightarrow\ 'a\ ptr$ **where**
‹$allocated\text{-}ptr\ \mathcal{S}\ T\ d\ d' = (THE\ p.\ (p,\ d') \in stack\text{-}alloc\ \mathcal{S}\ TYPE('a)\ d)$›

definition
$allocated\text{-}ptrs::\ nat \Rightarrow addr\ set \Rightarrow\ 'a::mem\text{-}type\ itself \Rightarrow heap\text{-}typ\text{-}desc \Rightarrow heap\text{-}typ\text{-}desc \Rightarrow\ 'a\ ptr$ **where**
‹$allocated\text{-}ptrs\ n\ \mathcal{S}\ T\ d\ d' = (THE\ p.\ (p,\ d') \in stack\text{-}allocs\ n\ \mathcal{S}\ TYPE('a)\ d)$›

lemma *allocated-ptr-allocated-ptrs-def*: $allocated\text{-}ptr = allocated\text{-}ptrs\ 1$
⟨*proof*⟩

abbreviation (*input*)
$cptr\text{-}type :: ('a :: c\text{-}type)\ ptr \Rightarrow\ 'a\ itself$
where
$cptr\text{-}type\ p \equiv TYPE('a)$

lemma *h-t-valid-guard-subst*:
⟦ *d*,*g* ⊨$_t$ *p*; *g*′ *p* ⟧ ⟹ *d*,*g*′ ⊨$_t$ *p*
⟨*proof*⟩

lemma *h-t-valid-ptr-retyp-eq*:
¬ *cptr-type p* <$_\tau$ *cptr-type p*′ ⟹ *h-t-valid* (*ptr-retyp p td*) *g p*′
= (*if ptr-span* (*p*::′*a*::*mem-type ptr*) ∩ *ptr-span* (*p*′::′*b*::*mem-type ptr*) = {} *then*
h-t-valid td g p′
else field-of-t p′ *p* ∧ *g p*′)
⟨*proof*⟩

lemma *field-of-t-refl*:
field-of-t p p′ = (*p* = *p*′)
⟨*proof*⟩

lemma *ptr-retyp-same-cleared-region*:
fixes *p* :: ′*a* :: *mem-type ptr* **and** *p*′ :: ′*a* :: *mem-type ptr*
assumes *ht*: *ptr-retyp p td*, *g* ⊨$_t$ *p*′
shows *p* = *p*′ ∨ {*ptr-val p*..+ *size-of TYPE*(′*a*)} ∩ {*ptr-val p*′ ..+ *size-of TYPE*(′*a*)}
= {}
⟨*proof*⟩

lemma (**in** *mem-type*) *ptr-force-type-h-t-valid*:
g p ⟹ *ptr-force-type p d*,*g* ⊨$_t$ (*p*::′*a ptr*)
⟨*proof*⟩

lemma *h-t-valid-ptr-force-type-eq*:
¬ *cptr-type p* <$_\tau$ *cptr-type p*′ ⟹ *h-t-valid* (*ptr-force-type p td*) *g p*′
= (*if ptr-span* (*p*::′*a*::*mem-type ptr*) ∩ *ptr-span* (*p*′::′*b*::*mem-type ptr*) = {} *then*
h-t-valid td g p′
else field-of-t p′ *p* ∧ *g p*′)
⟨*proof*⟩

lemma *ptr-force-type-same-cleared-region*:
fixes *p* :: ′*a* :: *mem-type ptr* **and** *p*′ :: ′*a* :: *mem-type ptr*
assumes *ht*: *ptr-force-type p td*, *g* ⊨$_t$ *p*′
shows *p* = *p*′ ∨ {*ptr-val p*..+ *size-of TYPE*(′*a*)} ∩ {*ptr-val p*′ ..+ *size-of TYPE*(′*a*)}
= {}
⟨*proof*⟩

lemma *stack-alloc-unique*:
assumes *p*: (*p*, *d*′) ∈ *stack-alloc* $\mathcal{S}$ (*TYPE*(′*a*::*mem-type*)) *d*
assumes *q*: (*q*, *d*′) ∈ *stack-alloc* $\mathcal{S}$ (*TYPE*(′*a*::*mem-type*)) *d*
shows *p* = *q*
⟨*proof*⟩

lemma *stack-allocs-unique*:
assumes *p*: (*p*, *d*′) ∈ *stack-allocs n* $\mathcal{S}$ (*TYPE*(′*a*::*mem-type*)) *d*
assumes *q*: (*q*, *d*′) ∈ *stack-allocs n* $\mathcal{S}$ (*TYPE*(′*a*::*mem-type*)) *d*

shows $p = q$
⟨*proof*⟩

lemma *stack-alloc-allocated-ptr*:
$(p,\ d') \in$ *stack-alloc* $\mathcal{S}$ *TYPE*('a) $d \Longrightarrow$ *allocated-ptr* $\mathcal{S}$ *TYPE*('a::*mem-type*) d $d' = p$
⟨*proof*⟩

lemma *stack-allocs-allocated-ptrs*:
$(p,\ d') \in$ *stack-allocs* n $\mathcal{S}$ *TYPE*('a) $d \Longrightarrow$ *allocated-ptrs* n $\mathcal{S}$ *TYPE*('a::*mem-type*) d $d' = p$
⟨*proof*⟩

lemma *null-not-stack-free*: $0 \notin$ *stack-free* d
⟨*proof*⟩

lemma *stack-alloc-stack-subset-stack-free*:
$(p,\ d') \in$ *stack-alloc* $\mathcal{S}$ *TYPE*('a::*mem-type*) $d \Longrightarrow$
ptr-span $p \subseteq$ *stack-free* d
⟨*proof*⟩

lemma *stack-allocs-stack-subset-stack-free'*:
$(p,\ d') \in$ *stack-allocs* n $\mathcal{S}$ *TYPE*('a::*mem-type*) $d \Longrightarrow i < n \Longrightarrow$
ptr-span $(p +_p$ *int* $i) \subseteq$ *stack-free* d
⟨*proof*⟩

lemma *stack-allocs-stack-subset-stack-free*:
$(p,\ d') \in$ *stack-allocs* n $\mathcal{S}$ *TYPE*('a::*mem-type*) $d \Longrightarrow$
$\{$*ptr-val* p ..+ $n *$ *size-of TYPE*('a)$\} \subseteq$ *stack-free* d
⟨*proof*⟩

lemma *stack-alloc-stack-free-mono*:
assumes *sub*: *stack-free d1* $\subseteq$ *stack-free d2*
assumes *alloc-d1*: $(p,\ d1') \in$ *stack-alloc* $\mathcal{S}$ *TYPE*('a::*mem-type*) *d1*
shows $\exists\, d2'.\ (p,\ d2') \in$ *stack-alloc* $\mathcal{S}$ *TYPE*('a) *d2*
⟨*proof*⟩

lemma *stack-allocs-stack-free-mono*:
assumes *sub*: *stack-free d1* $\subseteq$ *stack-free d2*
assumes *alloc-d1*: $(p,\ d1') \in$ *stack-allocs* n $\mathcal{S}$ *TYPE*('a::*mem-type*) *d1*
shows $\exists\, d2'.\ (p,\ d2') \in$ *stack-allocs* n $\mathcal{S}$ *TYPE*('a) *d2*
⟨*proof*⟩

lemma *stack-alloc-stack-free-eq*:
assumes *sub*: *stack-free d1* $=$ *stack-free d2*
assumes *alloc-d1*: $(p,\ d1') \in$ *stack-alloc* $\mathcal{S}$ *TYPE*('a::*mem-type*) *d1*
shows $\exists\, d2'.\ (p,\ d2') \in$ *stack-alloc* $\mathcal{S}$ *TYPE*('a) *d2*
⟨*proof*⟩

lemma *stack-allocs-stack-free-eq*:
assumes *sub*: *stack-free d1* = *stack-free d2*
assumes *alloc-d1*: (*p*, *d1'*) ∈ *stack-allocs n* 𝒮 *TYPE*('*a*::*mem-type*) *d1*
shows ∃ *d2'*. (*p*, *d2'*) ∈ *stack-allocs n* 𝒮 *TYPE*('*a*) *d2*
⟨*proof*⟩

lemma *fresh-ptr-stack-free-disjunct*:
(*p*, *d'*) ∈ *stack-alloc* 𝒮 *TYPE*('*a*::*mem-type*) *d* ⟹ *ptr-span p* ∩ *stack-free d'* = {}
⟨*proof*⟩

lemma *fresh-ptrs-stack-free-disjunct'*:
(*p*, *d'*) ∈ *stack-allocs n* 𝒮 *TYPE*('*a*::*mem-type*) *d* ⟹ *i* < *n* ⟹ *ptr-span* (*p* +$_p$ *int i*) ∩ *stack-free d'* = {}
⟨*proof*⟩

lemma *fresh-ptrs-stack-free-disjunct*:
(*p*, *d'*) ∈ *stack-allocs n* 𝒮 *TYPE*('*a*::*mem-type*) *d* ⟹
{*ptr-val p* ..+ *n* ∗ *size-of TYPE*('*a*) } ∩ *stack-free d'* = {}
⟨*proof*⟩

lemma *stack-allocs-neq*: (*p*, *d'*) ∈ *stack-allocs n* 𝒮 *TYPE*('*a*::*mem-type*) *d* ⟹ *d* ≠ *d'*
⟨*proof*⟩

lemma *stack-free-stack-alloc*:
assumes *p*: (*p*, *d'*) ∈ *stack-alloc* 𝒮 *TYPE*('*a*::*mem-type*) *d*
shows *stack-free d'* = *stack-free d* − *ptr-span p*
⟨*proof*⟩

lemma *stack-free-stack-allocs*:
assumes *p*: (*p*, *d'*) ∈ *stack-allocs n* 𝒮 *TYPE*('*a*::*mem-type*) *d*
shows *stack-free d'* = *stack-free d* − {*ptr-val p* ..+ *n* ∗ *size-of TYPE*('*a*)}
⟨*proof*⟩

lemma *stack-release-other*: *x* ∉ *ptr-span p* ⟹ *stack-release p d x* = *d x*
⟨*proof*⟩

lemma *stack-releases-other*:
fixes *p*::'*a*::*mem-type ptr*
shows *x* ∉ {*ptr-val p* ..+ *n* ∗ *size-of TYPE*('*a*::*mem-type*)} ⟹ *stack-releases n p d x* = *d x*
⟨*proof*⟩

lemma *in-ptr-span-itself*: *x* ∈ *ptr-span* (*PTR*('*a*::*mem-type*) *x*)
⟨*proof*⟩

lemma *stack-byte-typing-footprint*:
stack-byte-typing x = (*True*, *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*(*stack-byte*))

0))
⟨*proof*⟩

lemma *stack-release-footprint*: *x* ∈ *ptr-span p* ⟹
stack-release p d x = (*True*, *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*(*stack-byte*))
0))
⟨*proof*⟩

lemma *stack-releases-footprint*:
fixes *p*::*'a*::*mem-type ptr*
shows *x* ∈ {*ptr-val p ..+ n* ∗ *size-of TYPE*(*'a*::*mem-type*)} ⟹
stack-releases n p d x = (*True*, *list-map* (*typ-slice-t* (*typ-uinfo-t TYPE*(*stack-byte*))
0))
⟨*proof*⟩

lemma *stack-byte-typing-valid-root-footprint*:
valid-root-footprint stack-byte-typing x (*typ-uinfo-t TYPE*(*stack-byte*))
⟨*proof*⟩

lemma *stack-release-valid-root-footprint*: *x* ∈ *ptr-span p* ⟹
valid-root-footprint (*stack-release p d*) *x* (*typ-uinfo-t TYPE*(*stack-byte*))
⟨*proof*⟩

lemma *stack-releases-valid-root-footprint*:
fixes *p*::*'a*::*mem-type ptr*
shows *x* ∈ {*ptr-val p ..+ n* ∗ *size-of TYPE*(*'a*::*mem-type*)} ⟹
valid-root-footprint (*stack-releases n p d*) *x* (*typ-uinfo-t TYPE*(*stack-byte*))
⟨*proof*⟩

lemma *stack-release-root-ptr-valid1*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *non-stack-p*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *non-stack-q*: *typ-uinfo-t TYPE*(*'b*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *root-q*: *root-ptr-valid* (*stack-release p d*) *q*
shows *ptr-span p* ∩ *ptr-span q* = {} ∧ *root-ptr-valid d q*
⟨*proof*⟩

lemma *stack-releases-root-ptr-valid1*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *non-stack-p*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *non-stack-q*: *typ-uinfo-t TYPE*(*'b*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *root-q*: *root-ptr-valid* (*stack-releases n p d*) *q*
shows {*ptr-val p ..+ n* ∗ *size-of TYPE*(*'a*::*mem-type*)} ∩ *ptr-span q* = {} ∧
root-ptr-valid d q
⟨*proof*⟩

lemma *stack-release-root-ptr-valid2*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *disj*: *ptr-span p* ∩ *ptr-span q* = {}
assumes *valid-q*: *root-ptr-valid d q*
shows *root-ptr-valid* (*stack-release p d*) *q*
⟨*proof*⟩

lemma *stack-releases-root-ptr-valid2*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *disj*: {*ptr-val p* ..+ *n* ∗ *size-of TYPE*(*'a*::*mem-type*)} ∩ *ptr-span q* = {}
assumes *valid-q*: *root-ptr-valid d q*
shows *root-ptr-valid* (*stack-releases n p d*) *q*
⟨*proof*⟩

lemma *stack-release-root-ptr-valid-cases*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *non-stack-p*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *non-stack-q*: *typ-uinfo-t TYPE*(*'b*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
shows *root-ptr-valid* (*stack-release p d*) *q* ⟷ *ptr-span p* ∩ *ptr-span q* = {} ∧ *root-ptr-valid d q*
⟨*proof*⟩

lemma *stack-releases-root-ptr-valid-cases*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *non-stack-p*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *non-stack-q*: *typ-uinfo-t TYPE*(*'b*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
shows *root-ptr-valid* (*stack-releases n p d*) *q* ⟷ {*ptr-val p* ..+ *n* ∗ *size-of TYPE*(*'a*::*mem-type*)} ∩ *ptr-span q* = {} ∧ *root-ptr-valid d q*
⟨*proof*⟩

lemma *stack-release-root-ptr-valid-same-type-cases*:
fixes *p*::*'a*::*mem-type ptr*
assumes *cvalid-p*: *d* ⊨*t p*
assumes *non-stack-p*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
shows *root-ptr-valid* (*stack-release p d*) *q* ⟷ *p* ≠ *q* ∧ *root-ptr-valid d q*
⟨*proof*⟩

lemma *stack-releases-root-ptr-valid-same-type-cases*:
fixes *p*::*'a*::*mem-type ptr*
assumes *cvalid-p*: ⋀*i*. *i* < *n* ⟹ *d* ⊨*t* (*p* +*p int i*)
assumes *non-stack-p*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
shows *root-ptr-valid* (*stack-releases n p d*) *q* ⟷ (∀ *i* < *n*. *q* ≠ *p* +*p* (*int i*)) ∧ *root-ptr-valid d q*

⟨*proof*⟩

lemma *ptr-aligned-stack-byte*[*simp*]: *ptr-aligned* (*PTR*(*stack-byte*) *x*)
⟨*proof*⟩

lemma *c-null-guard-cast-stack-byte*:
x ∈ *ptr-span* (*p*::'*a*::*mem-type ptr*) ⟹ *c-null-guard p* ⟹
c-null-guard (*PTR* (*stack-byte*) *x*)
⟨*proof*⟩

lemma *c-guard-cast-stack-byte*:
x ∈ *ptr-span* (*p*::'*a*::*mem-type ptr*) ⟹ *c-guard p* ⟹
c-guard (*PTR* (*stack-byte*) *x*)
⟨*proof*⟩

lemma *stack-heap-typing-root-ptr-valid-footprint*: *c-guard* (*p*::*stack-byte ptr*) ⟹
root-ptr-valid stack-byte-typing p
⟨*proof*⟩

lemma *stack-release-root-ptr-valid-footprint*: *x* ∈ *ptr-span p* ⟹ *c-guard p* ⟹
root-ptr-valid (*stack-release p d*) (*PTR* (*stack-byte*) *x*)
⟨*proof*⟩

lemma *stack-releases-root-ptr-valid-footprint*:
fixes *p*::'*a*::*mem-type ptr*
shows *x* ∈ {*ptr-val p ..+ n* ∗ *size-of TYPE*('*a*::*mem-type*)} ⟹ ∀ *i* < *n*. *c-guard* (*p* +$_p$ *int i*) ⟹
root-ptr-valid (*stack-releases n p d*) (*PTR* (*stack-byte*) *x*)
⟨*proof*⟩

lemma *stack-alloc-other*:
(*p*, *d'*) ∈ *stack-alloc* $\mathcal{S}$ *TYPE*('*a*::*mem-type*) *d* ⟹ *x* ∉ *ptr-span p* ⟹
d' x = *d x*
⟨*proof*⟩

lemma *stack-allocs-other*:
(*p*, *d'*) ∈ *stack-allocs n* $\mathcal{S}$ *TYPE*('*a*::*mem-type*) *d* ⟹ *x* ∉ {*ptr-val p ..+ n* ∗ *size-of TYPE*('*a*::*mem-type*)} ⟹
d' x = *d x*
⟨*proof*⟩

lemma *stack-free-stack-release-mono*:
shows *stack-free d* ⊆ *stack-free* (*stack-release p d*)
⟨*proof*⟩

lemma *stack-free-stack-release-mono'*:
stack-free d1 ⊆ *stack-free d2* ⟹ *stack-free* (*stack-release p d1*) ⊆ *stack-free* (*stack-release p d2*)

⟨*proof*⟩

lemma *stack-free-stack-releases-mono*:
shows *stack-free d* ⊆ *stack-free* (*stack-releases n p d*)
⟨*proof*⟩

lemma *stack-free-stack-releases-mono'*:
stack-free d1 ⊆ *stack-free d2* ⟹ *stack-free* (*stack-releases n p d1*) ⊆ *stack-free* (*stack-releases n p d2*)
⟨*proof*⟩

lemma *stack-free-ptr-span-stack-release*:
c-null-guard p ⟹ *ptr-span p* ⊆ *stack-free* (*stack-release p d*)
⟨*proof*⟩

lemma *stack-free-ptr-span-stack-releases*:
fixes *p*::*'a*::*mem-type ptr*
shows (⋀*i*. *i* < *n* ⟹ *c-null-guard* (*p* +$_p$ *int i*)) ⟹
{*ptr-val p* ..+ *n* ∗ *size-of TYPE*(*'a*::*mem-type*)} ⊆ *stack-free* (*stack-releases n p d*)
⟨*proof*⟩

lemma *stack-free-stack-release*:
assumes *c-null-guard*: *c-null-guard p*
shows *stack-free* (*stack-release p d*) = *ptr-span p* ∪ *stack-free d*
⟨*proof*⟩

lemma *stack-free-stack-releases*:
fixes *p*::*'a*::*mem-type ptr*
assumes *c-null-guard*: ⋀*i*. *i* < *n* ⟹ *c-null-guard* (*p* +$_p$ *int i*)
shows *stack-free* (*stack-releases n p d*) = {*ptr-val p* ..+ *n* ∗ *size-of TYPE*(*'a*::*mem-type*)} ∪ *stack-free d*
⟨*proof*⟩

definition *On-Exit*:: (*'s*, *'p*, *'f*) *com* ⇒ (*'s*, *'p*, *'f*) *com* ⇒ (*'s*, *'p*, *'f*) *com* **where**
On-Exit c cleanup = *Seq* (*Catch c* (*Seq cleanup Throw*)) *cleanup*

locale *heap-typing-state* =
lense htd htd-upd
for
htd:: *'s* ⇒ *heap-typ-desc* **and**
htd-upd:: (*heap-typ-desc* ⇒ *heap-typ-desc*) ⇒ *'s* ⇒ *'s*

locale *heap-mem-state* =
lense hmem hmem-upd
for
hmem:: *'s* ⇒ *heap-mem* **and**
hmem-upd:: (*heap-mem* ⇒ *heap-mem*) ⇒ *'s* ⇒ *'s*

locale *heap-state* =
typing: *heap-typing-state htd htd-upd* + *heap*: *heap-mem-state hmem hmem-upd*
for
htd:: *'s* ⇒ *heap-typ-desc* **and**
htd-upd:: (*heap-typ-desc* ⇒ *heap-typ-desc*) ⇒ *'s* ⇒ *'s* **and**
hmem:: *'s* ⇒ *heap-mem* **and**
hmem-upd:: (*heap-mem* ⇒ *heap-mem*) ⇒ *'s* ⇒ *'s* +
assumes *heap-commute*: *hmem-upd f* (*htd-upd g s*) = *htd-upd g* (*hmem-upd f s*)

begin
lemma *htd-hmem-upd* [*simp*]: *htd* (*hmem-upd f s*) = *htd s*
⟨*proof*⟩

lemma *hmem-htd-upd* [*simp*]: *hmem* (*htd-upd f s*) = *hmem s*
⟨*proof*⟩
end

locale *heap-state-global* =
heap-state htd htd-upd hmem hmem-upd + *lense glob glob-upd*
for
htd:: *'s* ⇒ *heap-typ-desc* **and**
htd-upd:: (*heap-typ-desc* ⇒ *heap-typ-desc*) ⇒ *'s* ⇒ *'s* **and**
hmem:: *'s* ⇒ *heap-mem* **and**
hmem-upd:: (*heap-mem* ⇒ *heap-mem*) ⇒ *'s* ⇒ *'s* **and**
glob:: *'s* ⇒ *'a* **and**
glob-upd:: (*'a* ⇒ *'a*) ⇒ *'s* ⇒ *'s* +
assumes *glob-htd-commute*: ⋀*g h*. *glob-upd g* (*htd-upd h s*) = *htd-upd h* (*glob-upd g s*)
assumes *glob-hmem-commute*: ⋀*g h*. *glob-upd g* (*hmem-upd h s*) = *hmem-upd h* (*glob-upd g s*)

locale *heap-raw-state* =
lense t-hrs t-hrs-update
for
t-hrs :: *'s* ⇒ *heap-raw-state* **and**
t-hrs-update:: (*heap-raw-state* ⇒ *heap-raw-state*) ⇒ *'s* ⇒ *'s*
begin
sublocale *heap-state*
λs. (*hrs-htd* (*t-hrs s*)) *λupd*. (*t-hrs-update* (*hrs-htd-update upd*))
λs. (*hrs-mem* (*t-hrs s*)) *λupd*. (*t-hrs-update* (*hrs-mem-update upd*))
⟨*proof*⟩
end

locale *heap-raw-state-global* =
heap-raw-state t-hrs t-hrs-update + *lense glob glob-upd*
for
t-hrs :: *'s* ⇒ *heap-raw-state* **and**
t-hrs-update:: (*heap-raw-state* ⇒ *heap-raw-state*) ⇒ *'s* ⇒ *'s* **and**
glob:: *'s* ⇒ *'a* **and**

glob-upd:: $('a \Rightarrow 'a) \Rightarrow 's \Rightarrow 's +$
assumes *glob-heap-commute*: $\bigwedge g\ h.$ *glob-upd* g (*t-hrs-update* $h\ s$) = *t-hrs-update* h (*glob-upd* $g\ s$)
begin
sublocale *heap-state-global*
$\lambda s.$ (*hrs-htd* (*t-hrs* s)) $\lambda upd.$ (*t-hrs-update* (*hrs-htd-update* *upd*))
$\lambda s.$ (*hrs-mem* (*t-hrs* s)) $\lambda upd.$ (*t-hrs-update* (*hrs-mem-update* *upd*))
glob glob-upd
⟨*proof*⟩
end

locale *stack-heap-state* = *heap-state htd htd-upd hmem hmem-upd*
for
htd:: $'s \Rightarrow$ *heap-typ-desc* **and**
htd-upd:: (*heap-typ-desc* $\Rightarrow$ *heap-typ-desc*) $\Rightarrow$ $'s \Rightarrow 's$ **and**
hmem:: $'s \Rightarrow$ *heap-mem* **and**
hmem-upd:: (*heap-mem* $\Rightarrow$ *heap-mem*) $\Rightarrow$ $'s \Rightarrow 's +$
fixes $\mathcal{S}$::*addr set*
begin

definition *With-Fresh-Stack-Ptr*:: *nat* $\Rightarrow$ ($'s \Rightarrow$ $'a$ *list set*) $\Rightarrow$ (($'a$::*mem-type ptr*) $\Rightarrow$ ($'s$, $'p$, *strictc-errortype*) *com*) $\Rightarrow$
($'s$, $'p$, *strictc-errortype*) *com* **where**
‹*With-Fresh-Stack-Ptr n init c* =
Guard StackOverflow ({s. *stack-allocs* n $\mathcal{S}$ *TYPE*($'a$) (*htd* s) $\neq$ {} $\wedge$ ($\exists$ *vs*. *vs* $\in$ *init* s $\wedge$ *length vs* = n)})
(*DynCom* ($\lambda s0$.
Spec {(s, t). $\exists p\ d'$ *vs bs*.
(p, d') $\in$ *stack-allocs* n $\mathcal{S}$ *TYPE*($'a$) (*htd* s) $\wedge$
vs $\in$ *init* s $\wedge$ *length vs* = n $\wedge$ *length bs* = n $*$ *size-of TYPE*($'a$) $\wedge$
t = *hmem-upd* (*fold* (λi. *heap-update-padding* (p $+_p$ *int* i) (*vs*!i) (*take* (*size-of TYPE*($'a$)) (*drop* (i $*$ *size-of TYPE*($'a$)) *bs*))) [*0*..<n])
(*htd-upd* (λ-. d') s)};;
DynCom ($\lambda s1$.
On-Exit
(*c* (*allocated-ptrs* n $\mathcal{S}$ *TYPE*($'a$) (*htd s0*) (*htd s1*)))
(*Spec* {(s, t). $\exists$ *bs*. *length bs* = n $*$ *size-of TYPE*($'a$) $\wedge$
t = *hmem-upd* (*heap-update-list* (*ptr-val* (*allocated-ptrs* n $\mathcal{S}$ *TYPE*($'a$) (*htd s0*) (*htd s1*))) *bs*)
(*htd-upd* (*stack-releases* n ((*allocated-ptrs* n $\mathcal{S}$ *TYPE*($'a$) (*htd s0*) (*htd s1*)))) s)})))
›

⟨*ML*⟩

end

locale *stack-heap-raw-state* = *heap-raw-state t-hrs t-hrs-update*
  for
  t-hrs :: *'s* ⇒ *heap-raw-state* **and**
  t-hrs-update:: (*heap-raw-state* ⇒ *heap-raw-state*) ⇒ *'s* ⇒ *'s* +
fixes $\mathcal{S}$::*addr set*
begin
  sublocale *stack-heap-state*
    λs. hrs-htd (*t-hrs s*) *λupd. t-hrs-update* (*hrs-htd-update upd*)
    λs. hrs-mem (*t-hrs s*) *λupd. t-hrs-update* (*hrs-mem-update upd*)
    $\mathcal{S}$
    ⟨*proof*⟩
end

locale *globals-stack-heap-state* = *stack-heap-state htd htd-upd hmem hmem-upd* $\mathcal{S}$
  for
    htd:: *'s* ⇒ *heap-typ-desc* **and**
    htd-upd:: (*heap-typ-desc* ⇒ *heap-typ-desc*) ⇒ *'s* ⇒ *'s* **and**
    hmem:: *'s* ⇒ *heap-mem* **and**
    hmem-upd:: (*heap-mem* ⇒ *heap-mem*) ⇒ *'s* ⇒ *'s* **and**
    $\mathcal{S}$::*addr set* +
  fixes $\mathcal{G}$::*addr set*

locale *globals-stack-heap-raw-state* = *stack-heap-raw-state t-hrs t-hrs-update* $\mathcal{S}$
  for
  t-hrs :: *'s* ⇒ *heap-raw-state* **and**
  t-hrs-update:: (*heap-raw-state* ⇒ *heap-raw-state*) ⇒ *'s* ⇒ *'s* **and**
  $\mathcal{S}$::*addr set* +
  fixes $\mathcal{G}$::*addr set*
begin
sublocale *globals-stack-heap-state*
    λs. hrs-htd (*t-hrs s*) *λupd. t-hrs-update* (*hrs-htd-update upd*)
    λs. hrs-mem (*t-hrs s*) *λupd. t-hrs-update* (*hrs-mem-update upd*)
    $\mathcal{S}$ $\mathcal{G}$
    ⟨*proof*⟩
end

12.4 Misc derived language elements

definition
  creturn :: ((*'e c-exntype* ⇒ *'e c-exntype*) ⇒ (*'g*, *'l*, *'e*, *'x*) *state-scheme* ⇒ (*'g*, *'l*, *'e*, *'x*) *state-scheme*)
      ⇒ ((*'a* ⇒ *'a*) ⇒ (*'g*, *'l*, *'e*, *'x*) *state-scheme* ⇒ (*'g*, *'l*, *'e*, *'x*) *state-scheme*)
      ⇒ ((*'g*, *'l*, *'e*, *'x*) *state-scheme* ⇒ *'a*) ⇒ ((*'g*, *'l*, *'e*, *'x*) *state-scheme*,*'p*,*'f*) *com*
where
  creturn rtu xfu v ≡ (*Basic* (*λs. xfu* (*λ-. v s*) *s*);; (*Basic* (*rtu* (*λ-. Return*));; *THROW*))

definition

creturn-void :: ((*'e c-exntype* ⇒ *'e c-exntype*) ⇒ (*'g*, *'l*, *'e*, *'x*) *state-scheme*
⇒ (*'g*, *'l*, *'e*, *'x*) *state-scheme*) ⇒ ((*'g*, *'l*, *'e*, *'x*) *state-scheme*,*'p*,*'f*) *com*
where
creturn-void rtu ≡ (*Basic* (*rtu* (λ-. *Return*));; *THROW*)

definition
cexit :: ((*'g*, *'l*, *'e*, *'x*) *state-scheme* ⇒ (*'g*, *'l*, *'e*, *'x*) *state-scheme*) ⇒ ((*'g*, *'l*, *'e*, *'x*) *state-scheme*,*'p*,*'f*) *com*
where
cexit xfu ≡ (*Basic xfu*;; *THROW*)

definition
cbreak :: ((*'e c-exntype* ⇒ *'e c-exntype*) ⇒ (*'g*, *'l*, *'e*, *'x*) *state-scheme*
⇒ (*'g*, *'l*, *'e*, *'x*) *state-scheme*) ⇒ ((*'g*, *'l*, *'e*, *'x*) *state-scheme*,*'p*,*'f*) *com*
where
cbreak rtu ≡ (*Basic* (*rtu* (λ-. *Break*));; *THROW*)

definition

ccatchbrk :: ((*'g*, *'l*, *'e*, *'x*) *state-scheme* ⇒*'e c-exntype*) ⇒ ((*'g*, *'l*, *'e*, *'x*) *state-scheme*,*'p*,*'f*) *com*
where
ccatchbrk rt ≡ *Cond* {*s*. *rt s* = *Break*} *SKIP THROW*

definition
cgoto :: *string* ⇒ ((*'e c-exntype* ⇒ *'e c-exntype*) ⇒ (*'g*, *'l*, *'e*, *'x*) *state-scheme*
⇒ (*'g*, *'l*, *'e*, *'x*) *state-scheme*) ⇒ ((*'g*, *'l*, *'e*, *'x*) *state-scheme*,*'p*,*'f*) *com*
where
cgoto l rtu ≡ (*Basic* (*rtu* (λ-. *Goto l*));; *THROW*)

definition
ccatchgoto :: *string* ⇒ ((*'g*, *'l*, *'e*, *'x*) *state-scheme* ⇒ *'e c-exntype*) ⇒ ((*'g*, *'l*, *'e*, *'x*) *state-scheme*,*'p*,*'f*) *com*
where
ccatchgoto l rt ≡ *Cond* {*s*. *rt s* = *Goto l*} *SKIP THROW*

definition
ccatchreturn :: ((*'g*, *'l*, *'e*, *'x*) *state-scheme* ⇒ *'e c-exntype*) ⇒ ((*'g*, *'l*, *'e*, *'x*) *state-scheme*,*'p*,*'f*) *com*
where
ccatchreturn rt ≡ *Cond* {*s*. *is-local* (*rt s*)} *SKIP THROW*

definition
cchaos :: (*'b* ⇒ *'a* ⇒ *'a*) ⇒ (*'a*,*'c*,*'d*) *com*
where
cchaos upd ≡ *Spec* { (*s0*,*s*) . ∃ *v*. *s* = *upd v s0* }

definition
guarded-spec-body F R = Guard F (fst ‘ R) (Spec R)

end

theory *UMM*
imports
Padding-Equivalence
CLanguage
begin

instantiation *word* :: (*len8*) *stack-type*
begin
instance
⟨*proof*⟩
end

instantiation *ptr* :: (*c-type*) *stack-type*
begin
instance
⟨*proof*⟩
end

lemma *list-neq-witness*: *x* ∈ *set ys* ⟹ *x* ∉ *set xs* ⟹ *xs* ≠ *ys*
⟨*proof*⟩

lemma *stack-typ-info-array-tag-n*:
stack-typ-info ((*array-tag-n n*)::('*a*::{*stack-type*},'*b*::*finite*) *array xtyp-info*)
⟨*proof*⟩

instantiation *array* ::(*stack-type*, *finite*) *stack-type*
begin
instance
⟨*proof*⟩
end

lemma *max-non-zero-unfold*: *NO-MATCH 0 a* ⟹ *NO-MATCH 0 b* ⟹ *max a b* = (*if a* ≤ *b then b else a*)
⟨*proof*⟩

lemma *eq-comp*:
assumes *eq1*: *field-update* (*component-desc x*) *bs v* ≡ *g*
assumes *eq2*: *field-update* (*component-desc x*) *bs w* ≡ *g*
shows *field-update* (*component-desc x*) *bs v* = *field-update* (*component-desc x*) *bs w*

⟨*proof*⟩

lemma *word-rcat-single*: *word-rcat* [*x*] = *x*
⟨*proof*⟩

lemma *length-word-rsplit-8*: *length* ((*word-rsplit* (*x*::*8 word*)) :: *8 word list*) = *1*
⟨*proof*⟩
lemma *length-word-rsplit-16*: *length* ((*word-rsplit* (*x*::*16 word*)) :: *8 word list*) = *2*
⟨*proof*⟩
lemma *length-word-rsplit-32*: *length* ((*word-rsplit* (*x*::*32 word*)) :: *8 word list*) = *4*
⟨*proof*⟩
lemma *length-word-rsplit-64*: *length* ((*word-rsplit* (*x*::*64 word*)) :: *8 word list*) = *8*
⟨*proof*⟩
lemma *length-word-rsplit-128*: *length* ((*word-rsplit* (*x*::*128 word*)) :: *8 word list*) = *16*
⟨*proof*⟩

lemma *length-word-rsplit-signed-8*: *length* ((*word-rsplit* (*x*::*8 signed word*)) :: *8 word list*) = *1*
⟨*proof*⟩
lemma *length-word-rsplit-signed-16*: *length* ((*word-rsplit* (*x*::*16 signed word*)) :: *8 word list*) = *2*
⟨*proof*⟩
lemma *length-word-rsplit-signed-32*: *length* ((*word-rsplit* (*x*::*32 signed word*)) :: *8 word list*) = *4*
⟨*proof*⟩
lemma *length-word-rsplit-signed-64*: *length* ((*word-rsplit* (*x*::*64 signed word*)) :: *8 word list*) = *8*
⟨*proof*⟩
lemma *length-word-rsplit-signed-128*: *length* ((*word-rsplit* (*x*::*128 signed word*)) :: *8 word list*) = *16*
⟨*proof*⟩

lemmas *length-word-rsplit* =
length-word-rsplit-8
length-word-rsplit-16
length-word-rsplit-32
length-word-rsplit-64
length-word-rsplit-128
length-word-rsplit-signed-8
length-word-rsplit-signed-16
length-word-rsplit-signed-32
length-word-rsplit-signed-64
length-word-rsplit-signed-128

lemmas *wf-component-descs-intros* =
wf-component-descs-empty-typ-info

wf-component-descs-final-pad
wf-xfield.wf-component-descs-ti-typ-combine
wf-xfield.wf-component-descs-ti-typ-pad-combine

lemmas *component-descs-independent-intros* =
component-descs-independent-empty-typ-info
component-descs-independent-final-pad
wf-xfield.component-descs-independent-ti-typ-combine
wf-xfield.component-desc-independent-ti-typ-pad-combine

lemmas *wf-field-descs-intros* =
wf-field-descs-empty-typ-info
wf-field-descs-final-pad
wf-xfield.wf-field-descs-ti-typ-combine
wf-xfield.wf-field-descs-ti-typ-pad-combine

lemmas *contained-field-descs-intros* =
contained-field-descs-empty-typ-info
contained-field-descs-final-pad
contained-field-descs-ti-typ-combine
contained-field-descs-ti-typ-pad-combine

lemmas *field-update-simps* =
size-of-def ti-typ-pad-combine-def empty-typ-info-def ti-typ-combine-def ti-pad-combine-def
final-pad-def padup-def Let-def

lemmas *size-td-simps-arr-fl* =
size-td-simps
size-td-array align-td-array max-def

method *wf-xfield-solver* =
(*intro-locales*, *rule wf-field.intro*; *simp add*: *comp-def*)

method *try-wf-xfield-solver* **methods** *m* =
(*match* **conclusion in** *wf-xfield acc upd* **for** *acc upd* ⇒ *wf-xfield-solver* | *m*)

method *wf-component-descs-solver* =
(*try-wf-xfield-solver* ‹(*rule wf-component-descs-intros*)›)+

method *field-desc-independent-solver* =
(*rule field-desc-independent-PAD-expand*,
simp only: *aggregate-typ-combinators-simps set-toplevel-field-descs-combinator-simps*,
(*simp only*: *insert-union-out*)*?*,
rule field-desc-independent-PAD-collapse,
rule field-desc-independent.intro;
fastforce simp add: *fu-commutes-def*)

method *component-descs-independent-solver* =
((*try-wf-xfield-solver* ‹*rule component-descs-independent-intros*›)+;
field-desc-independent-solver)

method *wf-field-descs-solver* =
(*try-wf-xfield-solver* ‹(*rule wf-field-descs-intros*)›)+

method *contained-field-descs-solver* =
(*rule contained-field-descs-intros*)+

lemma *unat-less-helper'*: $x < \textit{of-nat}\ n \equiv \textit{True} \Longrightarrow \textit{unat}\ x < n$
⟨*proof*⟩

lemma *unat-less-helper-numeral*:
$x < (\textit{numeral}\ n) \Longrightarrow \textit{unat}\ x < (\textit{numeral}\ n)$
$x < 1 \Longrightarrow \textit{unat}\ x < 1$
⟨*proof*⟩

lemma *unat-less-helper-numeral'*:
$x < (\textit{numeral}\ n) \equiv \textit{True} \Longrightarrow \textit{unat}\ x < (\textit{numeral}\ n)$
$x < 1 \equiv \textit{True} \Longrightarrow \textit{unat}\ x < 1$
⟨*proof*⟩

lemma *nat-sint-less-helper*:
$i <s \textit{of-nat}\ n \Longrightarrow 0 \leq s\ i \Longrightarrow (\textit{nat}\ (\textit{sint}\ i)) < n$
⟨*proof*⟩

lemma *nat-sint-less-helper'*:
$i <s \textit{of-nat}\ n \equiv \textit{True} \Longrightarrow 0 \leq s\ i \equiv \textit{True} \Longrightarrow (\textit{nat}\ (\textit{sint}\ i)) < n$
⟨*proof*⟩

lemma *nat-sint-less-helper-numeral*:
$i <s (\textit{numeral}\ n) \Longrightarrow 0 \leq s\ i \Longrightarrow \textit{nat}\ (\textit{sint}\ i) < (\textit{numeral}\ n)$
$i <s 1 \Longrightarrow 0 \leq s\ i \Longrightarrow \textit{nat}\ (\textit{sint}\ i) < 1$
⟨*proof*⟩

lemma *nat-sint-less-helper-numeral'*:
$i <s (\textit{numeral}\ n) \equiv \textit{True} \Longrightarrow 0 \leq s\ i \equiv \textit{True} \Longrightarrow \textit{nat}\ (\textit{sint}\ i) < (\textit{numeral}\ n)$
$i <s 1 \equiv \textit{True} \Longrightarrow 0 \leq s\ i \equiv \textit{True} \Longrightarrow \textit{nat}\ (\textit{sint}\ i) < 1$
⟨*proof*⟩

lemma *sint-ucast-eq-uint'*:
$[\![\ \textit{LENGTH}('a) < \textit{LENGTH}('b)]\!]$
$\Longrightarrow \textit{sint}\ ((\textit{ucast} :: ('a{::}\textit{len word} \Rightarrow 'b{::}\textit{len word}))\ x) = \textit{uint}\ x$
⟨*proof*⟩

lemma *sint-ucast-signed-eq-uint*:
$\textit{LENGTH}('a) < \textit{LENGTH}('b) \Longrightarrow \textit{sint}\ (\textit{ucast}\ (x :: 'a :: \textit{len word}) :: 'b :: \textit{len}$

signed word) = *uint x*
⟨*proof*⟩

lemma *ucast-unat-sless-helper*:
UCAST('*a*::*len* → '*b*::*len signed*) *x* <*s of-nat n* ⟹
LENGTH('*a*::*len*) < *LENGTH*('*b*::*len*) ⟹ *unat x* < *n*
⟨*proof*⟩

lemma *ucast-unat-sless-helper'*:
UCAST('*a*::*len* → '*b*::*len signed*) *x* <*s of-nat n* ≡ *True* ⟹
LENGTH('*a*::*len*) < *LENGTH*('*b*::*len*) ⟹ *unat x* < *n*
⟨*proof*⟩

lemma *ucast-unat-sless-helper-numeral-n*:
UCAST('*a*::*len* → '*b*::*len signed*) *x* <*s* (*numeral n*) ⟹
LENGTH('*a*::*len*) < *LENGTH*('*b*::*len*) ⟹ *unat x* < (*numeral n*)
⟨*proof*⟩

lemma *ucast-unat-sless-helper-numeral-1*:
UCAST('*a*::*len* → '*b*::*len signed*) *x* <*s 1* ⟹
LENGTH('*a*::*len*) < *LENGTH*('*b*::*len*) ⟹ *unat x* < *1*
⟨*proof*⟩

lemmas *ucast-unat-sless-helper-numeral* =
ucast-unat-sless-helper-numeral-n
ucast-unat-sless-helper-numeral-1

lemma *ucast-unat-sless-helper-numeral'*:
UCAST('*a*::*len* → '*b*::*len signed*) *x* <*s* (*numeral n*) ≡ *True* ⟹
LENGTH('*a*::*len*) < *LENGTH*('*b*::*len*) ⟹ *unat x* < (*numeral n*)
UCAST('*a*::*len* → '*b*::*len signed*) *x* <*s 1* ≡ *True* ⟹
LENGTH('*a*::*len*) < *LENGTH*('*b*::*len*) ⟹ *unat x* < *1*
⟨*proof*⟩

lemma *ucast-unat-less-helper*:
UCAST('*a*::*len* → '*b*::*len*) *x* < *of-nat n* ⟹
LENGTH('*a*::*len*) ≤ *LENGTH*('*b*::*len*) ⟹ *unat x* < *n*
⟨*proof*⟩

lemma *ucast-unat-less-helper'*:
UCAST('*a*::*len* → '*b*::*len*) *x* < *of-nat n* ≡ *True* ⟹
LENGTH('*a*::*len*) ≤ *LENGTH*('*b*::*len*) ⟹ *unat x* < *n*
⟨*proof*⟩

lemma *ucast-unat-less-helper-numeral-n*:
UCAST('*a*::*len* → '*b*::*len*) *x* < (*numeral n*) ⟹
LENGTH('*a*::*len*) ≤ *LENGTH*('*b*::*len*) ⟹ *unat x* < (*numeral n*)
⟨*proof*⟩

lemma *ucast-unat-less-helper-numeral-1*:
UCAST('a::*len* → 'b::*len*) *x* < *1* ⟹
LENGTH('a::*len*) ≤ *LENGTH*('b::*len*) ⟹ *unat x* < *1*
⟨*proof*⟩

lemmas *ucast-unat-less-helper-numeral* =
ucast-unat-less-helper-numeral-n
ucast-unat-less-helper-numeral-1

lemma *ucast-unat-less-helper-numeral'*:
UCAST('a::*len* → 'b::*len*) *x* < (*numeral n*) ≡ *True* ⟹
LENGTH('a::*len*) ≤ *LENGTH*('b::*len*) ⟹ *unat x* < (*numeral n*)
UCAST('a::*len* → 'b::*len*) *x* < *1* ≡ *True* ⟹
LENGTH('a::*len*) ≤ *LENGTH*('b::*len*) ⟹ *unat x* < *1*
⟨*proof*⟩

lemma *len-of-less-basic-cases*:
LENGTH(*8*) < *LENGTH*(*16*)
LENGTH(*8*) < *LENGTH*(*32*)
LENGTH(*8*) < *LENGTH*(*64*)
LENGTH(*8*) < *LENGTH*(*128*)
LENGTH(*16*) < *LENGTH*(*32*)
LENGTH(*16*) < *LENGTH*(*64*)
LENGTH(*16*) < *LENGTH*(*128*)
LENGTH(*32*) < *LENGTH*(*64*)
LENGTH(*32*) < *LENGTH*(*128*)
LENGTH(*64*) < *LENGTH*(*128*)
⟨*proof*⟩

lemma *len-of-le-basic-cases*:
LENGTH(*8*) ≤ *LENGTH*(*16*)
LENGTH(*8*) ≤ *LENGTH*(*32*)
LENGTH(*8*) ≤ *LENGTH*(*64*)
LENGTH(*8*) ≤ *LENGTH*(*128*)
LENGTH(*16*) ≤ *LENGTH*(*32*)
LENGTH(*16*) ≤ *LENGTH*(*64*)
LENGTH(*16*) ≤ *LENGTH*(*128*)
LENGTH(*32*) ≤ *LENGTH*(*64*)
LENGTH(*32*) ≤ *LENGTH*(*128*)
LENGTH(*64*) ≤ *LENGTH*(*128*)
⟨*proof*⟩

⟨*ML*⟩

lemma *heap-update-fold-comp-apply* : *heap-update p v* (*g z*) ≡ (*heap-update p v* ∘ *g*) *z*
⟨*proof*⟩

named-theorems
fg-cons-simps **and**
typ-info-simps **and**
td-names-simps **and**
typ-name-simps **and**
upd-lift-simps **and**
upd-other-simps **and**
size-align-simps **and**
fl-Some-simps **and**
fl-ti-simps **and**
sub-typ-simps **and**
typ-tag-defs **and**
size-simps **and**
typ-name-itself **and**
heap-update-fold-toplevel-fields **and**
heap-update-fold-toplevel-fields-pointless **and**
h-val-fields **and** *heap-update-fields* **and**
h-val-unfold

named-theorems *field-lookup-prems*

declare
size-of-words[*size-simps*]
size-of-swords[*size-simps*]
size-of-array[*size-simps*]
size-of-stack-byte[*size-simps*]
size-of-ptr[*size-simps*]

lemma *field-of-lookup-info*:
fixes *p*::*'a*:: *mem-type ptr*
assumes *field*: *field-of off* (*typ-uinfo-t TYPE*(*'b*::*mem-type*)) (*typ-uinfo-t TYPE*(*'a*))
shows $\exists f.\ f \in$ *set* (*field-names-u* (*typ-uinfo-t TYPE*(*'a*)) (*typ-uinfo-t TYPE*(*'b*)))
$\wedge$
field-lookup (*typ-uinfo-t TYPE*(*'a*)) *f 0* = *Some* (*typ-uinfo-t TYPE*(*'b*), *unat off*) $\wedge$
field-of-t (*PTR*(*'b*) &(*p*→*f*)) *p* $\wedge$
&(*p*→*f*) = *ptr-val* (*p*::*'a ptr*) + *off* $\wedge$
{&(*p*→*f*)..+*size-td* (*typ-uinfo-t TYPE*(*'b*))} $\subseteq$ *ptr-span p*
⟨*proof*⟩

lemma *sub-typ-field-names-u-nonempty*:
assumes *s-t*: $s \leq t$
shows *field-names-u t s* $\neq$ []
⟨*proof*⟩

definition *TO-SUC* (*n*::*nat*) ≡ *n*

⟨*ML*⟩
declare [[*simproc del*: *TO-SUC*]]

lemma *array-tag-SUC*:
array-tag (*t*::('*a*::*c-type*,'*b*::*finite*) *array itself*) = *array-tag-n* (*TO-SUC* (*CARD*('*b*)))
⟨*proof*⟩

lemma *field-lookup-cons*: *field-lookup t* [*f*] *m* = *Some* (*t'*, *n*) ⟹ *wf-desc t* ⟹
field-lookup t (*f* # *g* # *gs*) *m* = *field-lookup t'* (*g*#*gs*) *n*
⟨*proof*⟩

lemma *field-lookup-cons'*:
field-lookup (*typ-info-t* (*TYPE*('*a*::*mem-type*))) [*f*] *m* = *Some* (*t'*, *n*) ⟹
field-lookup (*typ-info-t* (*TYPE*('*a*::*mem-type*))) (*f* # *g* # *gs*) *m* = *field-lookup t'*
(*g*#*gs*) *n*
⟨*proof*⟩

lemma *exists-conj-disj*: *P* ⟹ (∃ *n*. (*P* ∧ *Q n*) ∨ *R n*) = (∃ *n*. *Q n* ∨ *R n*)
⟨*proof*⟩

lemma *abs-if-eq*:
assumes ⋀*x*. *b x* ⟹ *f1 x* = *f2 x*
assumes ⋀*x*. ¬ *b x* ⟹ *g1 x* = *g2 x*
shows (λ*x*. *if b x then f1 x else g1 x*) = (λ*x*. *if b x then f2 x else g2 x*) ⟷ *True*
⟨*proof*⟩

lemma *field-lookup-typ-uinfo-t-Some*:
field-lookup (*typ-info-t TYPE*('*a*::*c-type*)) *f m* = *Some* (*s*, *n*) ⟹
field-lookup (*typ-uinfo-t TYPE*('*a*)) *f m* = *Some* (*export-uinfo s*, *n*)
⟨*proof*⟩

lemma *adjust-ti-wf-fd-pres'*:
fixes *t*::'*a xtyp-info*
and *st*::'*a xtyp-info-struct*
and *ts*::'*a xtyp-info-tuple list*
and *x*::'*a xtyp-info-tuple*
assumes *fg-cons*: *fg-cons acc upd*
shows
wf-fd t ⟹
wf-fd (*map-td* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *t*)
wf-fd-struct st ⟹
wf-fd-struct (*map-td-struct* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*)
st)
wf-fd-list ts ⟹
wf-fd-list (*map-td-list* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*) *ts*)
wf-fd-tuple x ⟹
wf-fd-tuple (*map-td-tuple* (λ*n algn*. *update-desc acc upd*) (*update-desc acc upd*)
x)
⟨*proof*⟩

lemma *adjust-ti-wf-fd-pres*[*simp*]: *fg-cons acc upd* $\Longrightarrow$ *wf-fd t* $\Longrightarrow$ *wf-fd* (*adjust-ti t acc upd*)
⟨*proof*⟩

lemma *neq-td-names-eq-neq-export-uinfo*: *td-names* (*typ-info-t t*) $\neq$ *td-names* (*typ-info-t s*)
$\Longrightarrow$ *export-uinfo* (*typ-info-t t*) = *export-uinfo* (*typ-info-t s*) $\longleftrightarrow$ *False*
⟨*proof*⟩

lemma *set-field-names-no-padding-all-field-names-no-padding-conv'*:
set (*field-names-no-padding* (*typ-info-t TYPE*('*a*::*mem-type*)) *t*) =
Set.filter
(λ*f*. ∃ *s n*. *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*) ∧ *export-uinfo s* = *t*)
(*set* (*all-field-names-no-padding* (*typ-info-t TYPE*('*a*))))
⟨*proof*⟩

lemma *field-names-no-padding-all-field-names-no-padding-conv'*:
field-names-no-padding (*typ-info-t TYPE*('*a*::*mem-type*)) *t* =
filter
(λ*f*. ∃ *s n*. *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*s*, *n*) ∧ *export-uinfo s* = *t*)
(*all-field-names-no-padding* (*typ-info-t TYPE*('*a*)))
⟨*proof*⟩

lemma *set-filter-insert*: *Set.filter P* (*insert x S*) =
(*if P x then insert x* (*Set.filter P S*) *else Set.filter P S*)
⟨*proof*⟩

lemma *set-filter-cons-image*: *Set.filter P* ((#) *x* ' *S*) = (#) *x* ' *Set.filter* (λ*fs*. *P* (*x*#*fs*)) *S*
⟨*proof*⟩

lemma *set-filter-Sup*: *Set.filter P* (⋃ *x*∈*X*. *S x*) = (⋃ *x*∈*X*. *Set.filter P* (*S x*))
⟨*proof*⟩

lemma *set-filter-empty*: *Set.filter P* {} = {}
⟨*proof*⟩

lemma *cons-image-Sup*: (#) *x* ' (⋃ *xs*∈*X*. *S xs*) = (⋃ *xs*∈*X*. ((#) *x* ' *S xs*))
⟨*proof*⟩

lemma *set-filter-image-all*:
assumes ⋀*x*. *x* < *n* $\Longrightarrow$ *P* (*f x*)
shows *Set.filter P* (*f* ' {*0*..<*n*}) = *f* ' {*0*..<*n*}
⟨*proof*⟩

lemma *set-filter-image-none*:
assumes $\bigwedge x.\ x < n \Longrightarrow \neg\ P\ (f\ x)$
shows *Set.filter P* $(f\ `\ \{0..<n\}) = \ \{\}$
⟨*proof*⟩

lemma *set-filter-union-distrib*: *Set.filter P* $(X \cup Y)$ = *Set.filter P X* $\cup$ *Set.filter P Y*
⟨*proof*⟩

lemma *sub-typ-refl* [*simp*]: $TYPE('a) \leq_\tau TYPE('a::c\text{-}type)$
⟨*proof*⟩

lemma *not-sub-typ-via-td-name*:
assumes *ta*: *typ-name* (*typ-info-t* $TYPE('a :: c\text{-}type)) \neq$ *pad-typ-name*
and *tina*: *typ-name* (*typ-info-t* $TYPE('a :: c\text{-}type)) \notin$ *td-names* (*typ-info-t* $TYPE('b :: c\text{-}type)$)
shows $\neg\ TYPE('a :: c\text{-}type) \leq_\tau TYPE('b :: c\text{-}type)$
⟨*proof*⟩

lemma *nat-to-bin-string-eq-to-nat-eq*:
assumes *eq*: *nat-to-bin-string n* = *nat-to-bin-string m*
shows $n = m$
⟨*proof*⟩

lemma *nat-to-bin-string-inj* [*simp*]: *nat-to-bin-string n* = *nat-to-bin-string m* $\longleftrightarrow$ $n = m$
⟨*proof*⟩

⟨*ML*⟩

lemma *rewrite-solve-prop*:
assumes *rew*: $(PROP\ P) \equiv Trueprop\ Q$
assumes *solve*: $PROP\ Trueprop\ Q$
shows $(PROP\ P) \equiv Trueprop\ True$
⟨*proof*⟩

lemma *trueprop-eq-bool-eq*:
assumes *prop-eq*: $PROP\ Trueprop\ P \equiv PROP\ Trueprop\ Q$
shows $P = Q$
⟨*proof*⟩

lemmas *rewrite-solve-prop-eq* = *eq-reflection* [*OF trueprop-eq-bool-eq*, *OF rewrite-solve-prop*]
lemmas *trueprop-eq-bool-meta-eq* = *trueprop-eq-bool-eq* [*THEN eq-reflection*]

lemma *export-uinfo-typ-uinfo-t-match*[*simp*]:
export-uinfo (*typ-info-t* $TYPE('a)) =$ *typ-uinfo-t* $(t::'a::c\text{-}type\ itself) = True$
⟨*proof*⟩

lemma *export-uinfo-eq-sub-typ-conv*:

export-uinfo (*typ-info-t TYPE*('*a*::*c-type*)) = *export-uinfo* (*typ-info-t TYPE*('*b*::*c-type*))
⟷
TYPE('*a*) ≤$_\tau$ *TYPE*('*b*) ∧ *TYPE*('*b*) ≤$_\tau$ *TYPE* ('*a*)
⟨*proof*⟩

lemma *typ-uinfo-eq-sub-typ-conv*:
typ-uinfo-t TYPE('*a*::*c-type*) = *export-uinfo* (*typ-info-t TYPE*('*b*::*c-type*))
⟷
TYPE('*a*) ≤$_\tau$ *TYPE*('*b*) ∧ *TYPE*('*b*) ≤$_\tau$ *TYPE* ('*a*)
export-uinfo (*typ-info-t TYPE*('*a*::*c-type*)) = *typ-uinfo-t TYPE*('*b*::*c-type*)
⟷
TYPE('*a*) ≤$_\tau$ *TYPE*('*b*) ∧ *TYPE*('*b*) ≤$_\tau$ *TYPE* ('*a*)
typ-uinfo-t TYPE('*a*::*c-type*) = *typ-uinfo-t TYPE*('*b*::*c-type*)
⟷
TYPE('*a*) ≤$_\tau$ *TYPE*('*b*) ∧ *TYPE*('*b*) ≤$_\tau$ *TYPE* ('*a*)
⟨*proof*⟩

lemma *array-typ-subtyp-array-typ*:
assumes *typ-uinfo-t* (*TYPE* ('*a*::*mem-type*)) = *typ-uinfo-t* (*TYPE*('*c*::*mem-type*))
shows *typ-uinfo-t* (*TYPE* ('*a*::*mem-type*['*b*::*finite*])) = *typ-uinfo-t* (*TYPE*('*c*['*b*::*finite*]))
⟨*proof*⟩

lemma *le-array-typ-intro*:
TYPE ('*a*::*mem-type*) ≤$_\tau$ *TYPE* ('*c*::*mem-type*) ⟹
TYPE ('*c*::*mem-type*) ≤$_\tau$ *TYPE* ('*a*::*mem-type*) ⟹
TYPE('*a*::*mem-type*['*b*::*finite*]) ≤$_\tau$ *TYPE*('*c*::*mem-type*['*b*::*finite*])
⟨*proof*⟩

lemma *sub-typ-signed-unsiged*: *TYPE*('*a*::*len8 signed word*) ≤$_\tau$ *TYPE*('*a word*)
⟨*proof*⟩

lemma *sub-typ-unsigned-signed*: *TYPE*('*a word*) ≤$_\tau$ *TYPE*('*a*::*len8 signed word*)
⟨*proof*⟩

lemma *sub-typ-proper-conv*: *TYPE*('*a*::*c-type*) <$_\tau$ *TYPE*('*b*::*c-type*) ⟷
typ-uinfo-t TYPE('*a*) ≠ *typ-uinfo-t TYPE*('*b*) ∧ *TYPE*('*a*::*c-type*) ≤$_\tau$ *TYPE*('*b*::*c-type*)
⟨*proof*⟩

lemma *sub-typ-proper-to-sub-typ*:
TYPE('*a*::*c-type*) <$_\tau$ *TYPE*('*b*::*c-type*) ⟹ *TYPE*('*a*::*c-type*) ≤$_\tau$ *TYPE*('*b*::*c-type*)
⟨*proof*⟩

lemma *to-bytes-p-zero*: *to-bytes-p* (*c-type-class.zero*::'*a*::*xmem-type*) = *replicate* (*size-of TYPE*('*a*)) *0*
⟨*proof*⟩

lemma *field-lookup-zero*:
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*::*xmem-type*)) *f 0* = *Some* (*t*, *n*)

assumes *match*: *export-uinfo* t = *typ-uinfo-t* $TYPE('b::c\text{-}type)$
shows *from-bytes* (*access-ti*$_0$ t (*c-type-class.zero*$::'a$)) = (*c-type-class.zero*$::'b$)
⟨*proof*⟩

lemma *field-lookup-zero'*:
assumes *fl*: *field-lookup* (*typ-info-t* $TYPE('a::xmem\text{-}type)$) f 0 ≡ *Some* (t, n)
assumes *match*: *export-uinfo* t = *export-uinfo* (*typ-info-t* $TYPE('b::c\text{-}type)$)
shows *from-bytes* (*access-ti*$_0$ t (*c-type-class.zero*$::'a$)) = (*c-type-class.zero*$::'b$)
⟨*proof*⟩

lemma *array-index-zero*:
assumes *i-bound*: i < $CARD('b)$
shows (*c-type-class.zero*$::(('a :: array\text{-}outer\text{-}max\text{-}size)['b :: array\text{-}max\text{-}count]))$.[$i$]
= (*c-type-class.zero*$::'a$)
⟨*proof*⟩

named-theorems *zero-simps* **and** *make-zero*
end

Chapter 13

Packed Types (no implicit padding)

theory *PackedTypes*
imports *WordSetup CProof*
begin

13.1 Underlying definitions for the class axioms

field-access / *field-update* is the identity for packed types

definition *fa-fu-idem* :: *'a field-desc* ⇒ *nat* ⇒ *bool* **where**
fa-fu-idem fd n ≡
∀ *bs bs' v*. *length bs* = *n* ⟶ *length bs'* = *n* ⟶ *field-access fd* (*field-update fd bs v*) *bs'* = *bs*

primrec
td-fafu-idem :: (*'a field-desc*,*'b*)*typ-desc* ⇒ *bool* **and**
td-fafu-idem-struct :: (*'a field-desc*,*'b*) *typ-struct* ⇒ *bool* **and**
td-fafu-idem-list :: ((*'a field-desc*,*'b*) *typ-desc*, *char list*,*'b*) *dt-tuple list* ⇒ *bool* **and**
td-fafu-idem-tuple :: ((*'a field-desc*,*'b*) *typ-desc*, *char list*,*'b*) *dt-tuple* ⇒ *bool* **where**
fai0: *td-fafu-idem* (*TypDesc algn ts n*) = *td-fafu-idem-struct ts*

| *fai1*: *td-fafu-idem-struct* (*TypScalar n algn d*) = *fa-fu-idem d n*
| *fai2*: *td-fafu-idem-struct* (*TypAggregate ts*) = *td-fafu-idem-list ts*

| *fai3*: *td-fafu-idem-list* [] = *True*
| *fai4*: *td-fafu-idem-list* (*x*#*xs*) = (*td-fafu-idem-tuple x* ∧ *td-fafu-idem-list xs*)

| *fai5*: *td-fafu-idem-tuple* (*DTuple x n d*) = *td-fafu-idem x*

lemmas *td-fafu-idem-simps* = *fai0 fai1 fai2 fai3 fai4 fai5*

field-access is independent of the underlying bytes

definition *fa-heap-indep* :: *'a field-desc* ⇒ *nat* ⇒ *bool* **where**
fa-heap-indep fd n ≡
∀ *bs bs' v*. *length bs* = *n* ⟶ *length bs'* = *n* ⟶ *field-access fd v bs* = *field-access fd v bs'*

primrec
td-fa-hi :: (*'a field-desc*,*'b*) *typ-desc* ⇒ *bool* **and**
td-fa-hi-struct :: (*'a field-desc*,*'b*) *typ-struct* ⇒ *bool* **and**
td-fa-hi-list :: ((*'a field-desc*,*'b*) *typ-desc*, *char list*,*'b*) *dt-tuple list* ⇒ *bool* **and**
td-fa-hi-tuple :: ((*'a field-desc*,*'b*) *typ-desc*, *char list*,*'b*) *dt-tuple* ⇒ *bool*
where
fahi0: *td-fa-hi* (*TypDesc algn ts n*) = *td-fa-hi-struct ts*

| *fahi1*: *td-fa-hi-struct* (*TypScalar n algn d*) = *fa-heap-indep d n*
| *fahi2*: *td-fa-hi-struct* (*TypAggregate ts*) = *td-fa-hi-list ts*

| *fahi3*: *td-fa-hi-list* [] = *True*
| *fahi4*: *td-fa-hi-list* (*x*#*xs*) = (*td-fa-hi-tuple x* ∧ *td-fa-hi-list xs*)

| *fahi5*: *td-fa-hi-tuple* (*DTuple x n d*) = *td-fa-hi x*

lemmas *td-fa-hi-simps* = *fahi0 fahi1 fahi2 fahi3 fahi4 fahi5*

13.2 Lemmas about *td-fafu-idem*

lemma *field-lookup-td-fafu-idem*:
shows ⋀(*s* :: (*'a field-desc*,*'b*) *typ-desc*) *f m n*.
⟦ *field-lookup t f m* = *Some* (*s*, *n*); *td-fafu-idem t* ⟧ ⟹ *td-fafu-idem s*
and ⋀(*s* :: (*'a field-desc*,*'b*) *typ-desc*) *f m n*.
⟦ *field-lookup-struct st f m* = *Some* (*s*, *n*); *td-fafu-idem-struct st* ⟧ ⟹ *td-fafu-idem s*
and ⋀(*s* :: (*'a field-desc*,*'b*) *typ-desc*) *f m n*.
⟦ *field-lookup-list ts f m* = *Some* (*s*, *n*); *td-fafu-idem-list ts* ⟧ ⟹ *td-fafu-idem s*
and ⋀(*s* :: (*'a field-desc*,*'b*) *typ-desc*) *f m n*.
⟦ *field-lookup-tuple p f m* = *Some* (*s*, *n*); *td-fafu-idem-tuple p* ⟧ ⟹ *td-fafu-idem s*
⟨*proof*⟩

lemma *field-access-update-same*:
fixes *t* :: (*'a* :: *mem-type field-desc*,*'b*) *typ-desc* **and** *st* :: (*'a field-desc*,*'b*) *typ-struct* **and**
ts:: (*'a field-desc*, *'b*) *typ-tuple list* **and**
p:: (*'a field-desc*, *'b*) *typ-tuple*
shows ⋀(*v* :: *'a*) *bs bs'*. ⟦ *td-fafu-idem t*; *wf-fd t*; *length bs* = *size-td t*; *length bs'* = *size-td t*⟧

$\Longrightarrow$ *access-ti t (update-ti t bs v) bs′ = bs*
and $\bigwedge$*(v :: 'a) bs bs′.* ⟦ *td-fafu-idem-struct st*; *wf-fd-struct st*; *length bs = size-td-struct st*; *length bs′ = size-td-struct st* ⟧
$\Longrightarrow$ *access-ti-struct st (update-ti-struct st bs v) bs′ = bs*
and $\bigwedge$*(v :: 'a) bs bs′.* ⟦ *td-fafu-idem-list ts*; *wf-fd-list ts*; *length bs = size-td-list ts*; *length bs′ = size-td-list ts*⟧
$\Longrightarrow$ *access-ti-list ts (update-ti-list ts bs v) bs′ = bs*
and $\bigwedge$*(v :: 'a) bs bs′.* ⟦ *td-fafu-idem-tuple p*; *wf-fd-tuple p*; *length bs = size-td-tuple p*; *length bs′ = size-td-tuple p*⟧
$\Longrightarrow$ *access-ti-tuple p (update-ti-tuple p bs v) bs′ = bs*
⟨*proof*⟩

lemma *access-ti-tuple-dt-fst*:
access-ti-tuple p v bs = access-ti (dt-fst p) v bs
⟨*proof*⟩

lemma *wf-fd-tuple-dt-fst*:
wf-fd-tuple p = wf-fd (dt-fst p)
⟨*proof*⟩

lemma *field-lookup-offset2*:
assumes *fl*: *(field-lookup t f (m + n) = Some (s, q))*
shows *field-lookup t f m = Some (s, q − n)*
⟨*proof*⟩

lemma *field-lookup-offset2-list*:
assumes *fl*: *(field-lookup-list ts f (m + n) = Some (s, q))*
shows *field-lookup-list ts f m = Some (s, q − n)*
⟨*proof*⟩

lemma *field-lookup-offset2-pair*:
assumes *fl*: *(field-lookup-tuple p f (m + n) = Some (s, q))*
shows *field-lookup-tuple p f m = Some (s, q − n)*
⟨*proof*⟩

lemma *field-access-update-nth-inner*:
shows $\bigwedge$*f (s :: ('a :: mem-type field-desc,'b) typ-desc) n x v bs bs′.*
⟦ *field-lookup t f 0 = Some (s, n)*; *n ≤ x*; *x < n + size-td s*; *td-fafu-idem s*; *wf-fd s*; *wf-fd t*;
length bs = size-td s; *length bs′ = size-td t* ⟧
$\Longrightarrow$ *access-ti t (update-ti s bs v) bs′ ! x = bs ! (x − n)*

and $\bigwedge$*f (s :: ('a :: mem-type field-desc,'b) typ-desc) n x v bs bs′.*
⟦*field-lookup-struct st f 0 = Some (s, n)*; *n ≤ x*; *x < n + size-td s*; *td-fafu-idem s*; *wf-fd s*; *wf-fd-struct st*;
length bs = size-td s; *length bs′ = size-td-struct st* ⟧
$\Longrightarrow$ *access-ti-struct st (update-ti s bs v) bs′ ! x = bs ! (x − n)*

and $\bigwedge f$ $(s :: ('a\ ::\ mem\text{-}type\ field\text{-}desc, 'b)\ typ\text{-}desc)\ n\ x\ v\ bs\ bs'$.
$[\![field\text{-}lookup\text{-}list\ ts\ f\ 0 = Some\ (s,\ n);\ n \leq x;\ x < n + size\text{-}td\ s;\ td\text{-}fafu\text{-}idem\ s;$
$wf\text{-}fd\ s;\ wf\text{-}fd\text{-}list\ ts;$
$length\ bs = size\text{-}td\ s;\ length\ bs' = size\text{-}td\text{-}list\ ts]\!]$
$\Longrightarrow access\text{-}ti\text{-}list\ ts\ (update\text{-}ti\ s\ bs\ v)\ bs'\ !\ x = bs\ !\ (x - n)$

and $\bigwedge f$ $(s :: ('a\ ::\ mem\text{-}type\ field\text{-}desc, 'b)\ typ\text{-}desc)\ n\ x\ v\ bs\ bs'$.
$[\![field\text{-}lookup\text{-}tuple\ p\ f\ 0 = Some\ (s,\ n);\ n \leq x;\ x < n + size\text{-}td\ s;\ td\text{-}fafu\text{-}idem\ s;$
$wf\text{-}fd\ s;\ wf\text{-}fd\text{-}tuple\ p;$
$length\ bs = size\text{-}td\ s;\ length\ bs' = size\text{-}td\text{-}tuple\ p]\!]$
$\Longrightarrow access\text{-}ti\text{-}tuple\ p\ (update\text{-}ti\ s\ bs\ v)\ bs'\ !\ x = bs\ !\ (x - n)$
⟨*proof*⟩

13.2.1 *td-fa-hi*

lemma *fa-heap-indepD*:
$[\![\ fa\text{-}heap\text{-}indep\ fd\ n;\ length\ bs = n;\ length\ bs' = n\ ]\!] \Longrightarrow$
$field\text{-}access\ fd\ v\ bs = field\text{-}access\ fd\ v\ bs'$
⟨*proof*⟩

lemma *td-fa-hi-heap-independence*:
fixes t::$('a$::$mem\text{-}type,\ 'b)\ typ\text{-}info$ **and**
st::$('a$::$mem\text{-}type, 'b)\ typ\text{-}info\text{-}struct$ **and**
ts::$('a$::$mem\text{-}type,\ 'b)\ typ\text{-}info\text{-}tuple\ list$ **and**
p::$('a$::$mem\text{-}type,\ 'b)\ typ\text{-}info\text{-}tuple$

shows $\bigwedge(v :: 'a :: mem\text{-}type)\ h\ h'.\ [\![\ td\text{-}fa\text{-}hi\ t;\ length\ h = size\text{-}td\ t;\ length\ h' = size\text{-}td\ t\ ]\!]$
$\Longrightarrow access\text{-}ti\ t\ v\ h = access\text{-}ti\ t\ v\ h'$
and $\bigwedge(v :: 'a :: mem\text{-}type)\ h\ h'.\ [\![\ td\text{-}fa\text{-}hi\text{-}struct\ st;\ length\ h = size\text{-}td\text{-}struct\ st;\ length\ h' = size\text{-}td\text{-}struct\ st]\!]$
$\Longrightarrow access\text{-}ti\text{-}struct\ st\ v\ h = access\text{-}ti\text{-}struct\ st\ v\ h'$
and $\bigwedge(v :: 'a :: mem\text{-}type)\ h\ h'.\ [\![\ td\text{-}fa\text{-}hi\text{-}list\ ts;\ \ length\ h = size\text{-}td\text{-}list\ ts;\ length\ h' = size\text{-}td\text{-}list\ ts\ ]\!]$
$\Longrightarrow access\text{-}ti\text{-}list\ ts\ v\ h = access\text{-}ti\text{-}list\ ts\ v\ h'$
and $\bigwedge(v :: 'a :: mem\text{-}type)\ h\ h'.\ [\![\ td\text{-}fa\text{-}hi\text{-}tuple\ p;\ \ length\ h = size\text{-}td\text{-}tuple\ p;\ length\ h' = size\text{-}td\text{-}tuple\ p\ ]\!]$
$\Longrightarrow access\text{-}ti\text{-}tuple\ p\ v\ h = access\text{-}ti\text{-}tuple\ p\ v\ h'$
⟨*proof*⟩

13.3 Simp rules for deriving packed props from the type combinators

13.3.1 *td-fafu-idem*

lemma *td-fafu-idem-map-align* [*simp*]: $td\text{-}fafu\text{-}idem\ (map\text{-}align\ f\ t) = td\text{-}fafu\text{-}idem\ t$

⟨*proof*⟩

lemma *td-fafu-idem-final-pad*:
padup (*2* ^ *max algn* (*align-td t*)) (*size-td t*) = *0*
⟹ *td-fafu-idem* (*final-pad algn t*) = *td-fafu-idem t*
⟨*proof*⟩

lemma *td-fafu-idem-ti-typ-pad-combine*:
fixes *t* :: *'a* :: *c-type itself* **and** *s* :: (*'b* :: *c-type*) *xtyp-info*
assumes *pad*: *padup* (*max* (*2* ^ *algn*) (*align-of TYPE*(*'a*))) (*size-td s*) = *0*
shows *td-fafu-idem* (*ti-typ-pad-combine t xf xfu algn nm s*) = *td-fafu-idem* (*ti-typ-combine t xf xfu algn nm s*)
⟨*proof*⟩

lemma *td-fafu-idem-list-append*:
fixes *xs* :: *'a* :: *c-type xtyp-info-tuple list*
shows *td-fafu-idem-list* (*xs* @ *ys*) = (*td-fafu-idem-list xs* ∧ *td-fafu-idem-list ys*)
⟨*proof*⟩

lemma *td-fafu-idem-extend-ti*:
fixes *t* :: *'a* :: *c-type xtyp-info*
fixes *s* :: *'a* :: *c-type xtyp-info*
assumes *as*: *td-fafu-idem s*
and *at*: *td-fafu-idem t*
shows *td-fafu-idem* (*extend-ti s t algn nm d*) ⟨*proof*⟩

lemma *fd-cons-access-updateD*:
⟦ *fd-cons-access-update d n*; *length bs* = *n*; *length bs'* = *n*⟧ ⟹
field-access d (*field-update d bs v*) *bs'* = *field-access d* (*field-update d bs v'*) *bs'*
⟨*proof*⟩

lemma *fa-fu-idem-update-desc*:
fixes *a* :: *'a field-desc*
assumes *fg*: *fg-cons xf xfu*
and *fd*: *fd-cons-struct* (*TypScalar n n' a*)
shows *fa-fu-idem* (*update-desc xf xfu a*) *n* = *fa-fu-idem a n*
⟨*proof*⟩

lemma *td-fafu-idem-map-td-update-desc*:
assumes *fg*: *fg-cons xf xfu*
shows *wf-fd t* ⟹ *td-fafu-idem* (*map-td* (λ- -. *update-desc xf xfu*) (*update-desc xf xfu*) *t*) = *td-fafu-idem t*
and *wf-fd-struct st* ⟹ *td-fafu-idem-struct* (*map-td-struct* (λ- -. *update-desc xf xfu*) (*update-desc xf xfu*) *st*) = *td-fafu-idem-struct st*
and *wf-fd-list ts* ⟹ *td-fafu-idem-list* (*map-td-list* (λ- -. *update-desc xf xfu*) (*update-desc xf xfu*) *ts*) = *td-fafu-idem-list ts*
and *wf-fd-tuple p* ⟹ *td-fafu-idem-tuple* (*map-td-tuple* (λ- -. *update-desc xf xfu*) (*update-desc xf xfu*) *p*) = *td-fafu-idem-tuple p*
⟨*proof*⟩

lemmas *td-fafu-idem-adjust-ti* = *td-fafu-idem-map-td-update-desc*(*1*)[*folded adjust-ti-def*]

lemma *td-fafu-idem-ti-typ-combine*:
fixes *s* :: *'b* :: *c-type xtyp-info*
assumes *fg*: *fg-cons xf xfu*
and *tda*: *td-fafu-idem* (*typ-info-t TYPE*(*'a* :: *mem-type*))
and *tds*: *td-fafu-idem s*
shows *td-fafu-idem* (*ti-typ-combine TYPE*(*'a* :: *mem-type*) *xf xfu algn nm s*)
⟨*proof*⟩

lemma *td-fafu-idem-ptr*:
td-fafu-idem (*typ-info-t TYPE*(*'a* :: *c-type ptr*))
⟨*proof*⟩

lemma *td-fafu-idem-word*:
td-fafu-idem (*typ-info-t TYPE*(*'a* :: *len8 word*))
⟨*proof*⟩

lemma *td-fafu-idem-array-n*:
⟦ *td-fafu-idem* (*typ-info-t TYPE*(*'a*)); *n* ≤ *card* (*UNIV* :: *'b set*) ⟧ ⟹
td-fafu-idem (*array-tag-n n* :: (*'a* :: *mem-type* [*'b* :: *finite*]) *xtyp-info*)
⟨*proof*⟩

lemma *td-fafu-idem-array*:
td-fafu-idem (*typ-info-t TYPE*(*'a*)) ⟹ *td-fafu-idem* (*typ-info-t TYPE*(*'a* :: *mem-type* [*'b* :: *finite*]))
⟨*proof*⟩

lemma *td-fafu-idem-empty-typ-info*:
td-fafu-idem (*empty-typ-info algn t*)
⟨*proof*⟩

13.3.2 *td-fa-hi*

lemma *td-fa-hi-final-pad*:
padup (*2* ^ *max algn* (*align-td t*)) (*size-td t*) = *0*
⟹ *td-fa-hi* (*final-pad algn t*) = *td-fa-hi t*
⟨*proof*⟩

lemma *td-fa-hi-ti-typ-pad-combine*:
fixes *t* :: *'a* :: *c-type itself* **and** *s* :: *'b* :: *c-type xtyp-info*
assumes *pad*: *padup* (*max* (*2* ^ *algn*) (*align-of TYPE*(*'a*))) (*size-td s*) = *0*
shows *td-fa-hi* (*ti-typ-pad-combine t xf xfu algn nm s*) = *td-fa-hi* (*ti-typ-combine t xf xfu algn nm s*)
⟨*proof*⟩

lemma *td-fa-hi-list-append*:

fixes *xs* :: *'a* :: *c-type xtyp-info-tuple list*
shows *td-fa-hi-list* (*xs* @ *ys*) = (*td-fa-hi-list xs* ∧ *td-fa-hi-list ys*)
⟨*proof*⟩

lemma *td-fa-hi-extend-ti*:
fixes *t* :: *'a* :: *c-type xtyp-info*
assumes *as*: *td-fa-hi s*
and *at*: *td-fa-hi t*
shows *td-fa-hi* (*extend-ti s t algn nm d*) ⟨*proof*⟩

lemma *fa-heap-indep-update-desc*:
fixes *a* :: *'a field-desc*
assumes *fg*: *fg-cons xf xfu*
and *fd*: *fd-cons-struct* (*TypScalar n n' a*)
shows *fa-heap-indep* (*update-desc xf xfu a*) *n* = *fa-heap-indep a n*
⟨*proof*⟩

lemma *td-fa-hi-map-td-update-desc*:
assumes *fg*: *fg-cons xf xfu*
shows *wf-fd t* ⟹ *td-fa-hi* (*map-td* (λ- -. *update-desc xf xfu*) (*update-desc xs xfu*) *t*) = *td-fa-hi t*
and *wf-fd-struct st* ⟹ *td-fa-hi-struct* (*map-td-struct* (λ- -. *update-desc xf xfu*) (*update-desc xs xfu*) *st*) = *td-fa-hi-struct st*
and *wf-fd-list ts* ⟹ *td-fa-hi-list* (*map-td-list* (λ- -. *update-desc xf xfu*) (*update-desc xs xfu*) *ts*) = *td-fa-hi-list ts*
and *wf-fd-tuple p* ⟹ *td-fa-hi-tuple* (*map-td-tuple* (λ- -. *update-desc xf xfu*) (*update-desc xs xfu*) *p*) = *td-fa-hi-tuple p*
⟨*proof*⟩

lemma *td-fa-hi-adjust-ti*:
assumes *fg*: *fg-cons xf xfu*
assumes *wf*: *wf-fd t*
shows *td-fa-hi* (*adjust-ti t xf xfu*) = *td-fa-hi t*
⟨*proof*⟩

lemma *td-fa-hi-ti-typ-combine*:
fixes *s* :: *'b* :: *c-type xtyp-info*
assumes *fg*: *fg-cons xf xfu*
and *tda*: *td-fa-hi* (*typ-info-t TYPE*(*'a* :: *mem-type*))
and *tds*: *td-fa-hi s*
shows *td-fa-hi* (*ti-typ-combine TYPE*(*'a* :: *mem-type*) *xf xfu algn nm s*)
⟨*proof*⟩

lemma *td-fa-hi-ptr*:
td-fa-hi (*typ-info-t TYPE*(*'a* :: *c-type ptr*))
⟨*proof*⟩

lemma *td-fa-hi-word*:
td-fa-hi (*typ-info-t TYPE*(*'a* :: *len8 word*))

⟨*proof*⟩

lemma *td-fa-hi-array-n*:
⟦*td-fa-hi* (*typ-info-t TYPE*('*a*)); *n* ≤ *card* (*UNIV* :: '*b set*) ⟧ ⟹ *td-fa-hi* (*array-tag-n n* :: ('*a* :: *mem-type* ['*b* :: *finite*]) *xtyp-info*)
⟨*proof*⟩

lemma *td-fa-hi-array*:
td-fa-hi (*typ-info-t TYPE*('*a*)) ⟹ *td-fa-hi* (*typ-info-t TYPE*('*a* :: *mem-type* ['*b* :: *finite*]))
⟨*proof*⟩

lemma *td-fa-hi-empty-typ-info*:
td-fa-hi (*empty-typ-info algn t*)
⟨*proof*⟩

13.4 The type class and simp sets

Packed types, with no padding, have the defining property that access is invariant under substitution of the underlying heap and access/update is the identity

class *packed-type* = *mem-type* +
assumes *td-fafu-idem*: *td-fafu-idem* (*typ-info-t TYPE*('*a*))
assumes *td-fa-hi*: *td-fa-hi* (*typ-info-t TYPE*('*a*))

lemmas *td-fafu-idem-intro-simps* =
— Axioms
td-fafu-idem
— Combinators
td-fafu-idem-final-pad td-fafu-idem-ti-typ-pad-combine td-fafu-idem-ti-typ-combine td-fafu-idem-empty-typ-info
— Constructors
td-fafu-idem-ptr td-fafu-idem-word td-fafu-idem-array

lemmas *td-fa-hi-intro-simps* =
— Axioms
td-fa-hi
— Combinators
td-fa-hi-final-pad td-fa-hi-ti-typ-pad-combine td-fa-hi-ti-typ-combine td-fa-hi-empty-typ-info
— Constructors
td-fa-hi-ptr td-fa-hi-word td-fa-hi-array

lemma *align-td-wo-align-array'*:
align-td-wo-align (*typ-info-t TYPE*('*a* :: *c-type*['*b* :: *finite*])) = *align-td-wo-align* (*typ-info-t TYPE*('*a*))
⟨*proof*⟩

lemma *align-td-array'*:

align-td (*typ-info-t TYPE*('*a* :: *c-type*['*b* :: *finite*])) = *align-td* (*typ-info-t TYPE*('*a*))
⟨*proof*⟩

lemmas *packed-type-intro-simps* =
td-fafu-idem-intro-simps td-fa-hi-intro-simps align-td-wo-align-array′ *size-td-simps-3 size-td-array*

lemma *access-ti-append*′:
⋀*list*.
access-ti-list (*xs* @ *ys*) *t list* =
access-ti-list xs t (*take* (*size-td-list xs*) *list*) @
access-ti-list ys t (*drop* (*size-td-list xs*) *list*)
⟨*proof*⟩

13.5 Instances

Words (of multiple of 8 size) are packed

instantiation *word* :: (*len8*) *packed-type*
begin
instance
⟨*proof*⟩
end

Pointers are always packed

instantiation *ptr* :: (*c-type*)*packed-type*
begin
instance
⟨*proof*⟩
end

Arrays of packed types are in turn packed

class *array-outer-packed* = *packed-type* + *array-outer-max-size*
class *array-inner-packed* = *array-outer-packed* + *array-inner-max-size*

instance *word* :: (*len8*)*array-outer-packed* ⟨*proof*⟩
instance *word* :: (*len8*)*array-inner-packed* ⟨*proof*⟩

instance *array* :: (*array-outer-packed*, *array-max-count*) *packed-type*
⟨*proof*⟩

instance *array* :: (*array-inner-packed*, *array-max-count*) *array-outer-packed* ⟨*proof*⟩

13.6 Theorems about packed types

13.6.1 *td-fa-hi*

lemma *heap-independence*:
⟦*length h* = *size-of TYPE*('*a* :: *packed-type*); *length h*′ = *size-of TYPE*('*a*) ⟧

$\Longrightarrow$ *access-ti* (*typ-info-t TYPE*('*a*)) *v h* = *access-ti* (*typ-info-t TYPE*('*a*)) *v h*′
⟨*proof*⟩

theorem *packed-heap-update-collapse*:
fixes *u*::'*a*::*packed-type*
fixes *v*::'*a*
shows *heap-update p v* (*heap-update p u h*) = *heap-update p v h*
⟨*proof*⟩

lemma *packed-heap-update-collapse-hrs*:
fixes *p* :: '*a* :: *packed-type ptr*
shows *hrs-mem-update* (*heap-update p v*) (*hrs-mem-update* (*heap-update p v*′) *hp*) =
hrs-mem-update (*heap-update p v*) *hp*
⟨*proof*⟩

13.6.2 *td-fafu-idem*

lemma *order-leE*:
fixes *x* :: '*a* :: *order*
shows ⟦ *x* ≤ *y*; *x* = *y* $\Longrightarrow$ *P*; *x* < *y* $\Longrightarrow$ *P* ⟧ $\Longrightarrow$ *P*
⟨*proof*⟩

lemma *of-nat-mono-maybe-le*:
shows ⟦*X* < 2 ^ *len-of TYPE*('*a*); *Y* ≤ *X*⟧ $\Longrightarrow$ (*of-nat Y* :: '*a* :: *len word*) ≤ *of-nat X*
⟨*proof*⟩

lemma *intvl-le-lower*:
fixes *x* :: '*a* :: *len word*
shows ⟦ *x* ∈ {*y*..+*n*}; *y* ≤ *y* + *of-nat* (*n* − 1); *n* < 2 ^ *len-of TYPE*('*a*) ⟧ $\Longrightarrow$ *y* ≤ *x*
⟨*proof*⟩

lemma *intvl-less-upper*:
fixes *x* :: '*a* :: *len word*
shows ⟦ *x* ∈ {*y*..+*n*}; *y* ≤ *y* + *of-nat* (*n* − 1); *n* < 2 ^ *len-of TYPE*('*a*) ⟧ $\Longrightarrow$ *x* ≤ *y* + *of-nat* (*n* − 1)
⟨*proof*⟩

lemma *packed-type-access-ti*:
fixes *v* :: '*a* :: *packed-type*
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
shows *access-ti* (*typ-info-t TYPE*('*a*)) *v bs* = *access-ti*$_0$ (*typ-info-t TYPE*('*a*)) *v*
⟨*proof*⟩

lemma *c-guard-field-lvalue*:
fixes *p* :: '*a* :: *mem-type ptr*
assumes *cg*: *c-guard p*

and *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*, *n*)
and *eu*: *export-uinfo t* = *typ-uinfo-t TYPE*('*b* :: *mem-type*)
shows *c-guard* (*Ptr* &(*p*→*f*) :: '*b* :: *mem-type ptr*)
⟨*proof*⟩

lemma *word-wrap-of-natD*:
fixes *x* :: '*a* :: *len word*
assumes *wraps*: ¬ *x* ≤ *x* + *of-nat n*
shows ∃ *k*. *x* + *of-nat k* = *0* ∧ *k* ≤ *n*
⟨*proof*⟩

theorem *packed-heap-super-field-update*:
fixes *v* :: '*a* :: *packed-type* **and** *p* :: '*b* :: *packed-type ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*b*)) *f 0* = *Some* (*t*, *n*)
and *cgrd*: *c-guard p*
and *eu*: *export-uinfo t* = *typ-uinfo-t TYPE*('*a*)
shows *heap-update* (*Ptr* &(*p*→*f*)) *v hp* = *heap-update p* (*update-ti t* (*to-bytes-p v*) (*h-val hp p*)) *hp*
⟨*proof*⟩

13.6.3 Proof automation for packed types

definition *td-packed* :: ('*a*,'*b*) *typ-info* ⇒ *nat* ⇒ *nat* ⇒ *bool*
where *td-packed t sz al* ⟷
td-fafu-idem t ∧ *td-fa-hi t* ∧ *aggregate t* ∧ *size-td t* = *sz* ∧ *align-td t* = *al*

lemma *packed-type-class-intro*:
td-packed (*typ-info-t TYPE*('*a*::*mem-type*)) *s a*
⟹ *OFCLASS*('*a*::*mem-type*, *packed-type-class*)
⟨*proof*⟩

lemma *td-fa-hi-map-align*[*simp*]:*td-fa-hi* (*map-align f t*) = *td-fa-hi t*
⟨*proof*⟩

lemma *td-packed-final-pad*:
⟦*td-packed t s a*; *2* ^ (*max algn a*) *dvd s*⟧ ⟹ *td-packed* (*final-pad algn t*) *s* (*max algn a*)
⟨*proof*⟩

lemma *td-packed-final-pad'*:
assumes *packed-t*: *td-packed t s a*
assumes *le*: *algn* ≤ *a*
assumes *dvd*: *2* ^ *a dvd s*
shows *td-packed* (*final-pad algn t*) *s a*
⟨*proof*⟩

lemma *td-packed-ti-typ-combine*:
⟦ *td-packed* (*td*::'*a*::*c-type xtyp-info*) *s a*;
align-of TYPE('*b*::*packed-type*) *dvd s*; *fg-cons xf xfu*; *aggregate td* ⟧

$\Longrightarrow$ *td-packed* (*ti-typ-combine TYPE*('*b*) *xf xfu algn nm td*)
(*s* + *size-td* (*typ-info-t TYPE*('*b*)))
(*max a* (*max algn* (*align-td* (*typ-info-t TYPE*('*b*)))))
⟨*proof*⟩

lemma *td-packed-ti-typ-pad-combine*:
⟦ *td-packed* (*td*::'*a*::*c-type xtyp-info*) *s a*;
align-of TYPE('*b*::*packed-type*) *dvd s*; *algn* ≤ *align-td* (*typ-info-t TYPE*('*b*)); *fg-cons xf xfu*; *aggregate td*⟧
$\Longrightarrow$ *td-packed* (*ti-typ-pad-combine TYPE*('*b*) *xf xfu algn nm td*)
(*s* + *size-td* (*typ-info-t TYPE*('*b*)))
(*max a* (*align-td* (*typ-info-t TYPE*('*b*))))
⟨*proof*⟩

lemma *td-packed-ti-typ-combine-array*:
⟦*td-packed* (*td*::'*a*::*c-type xtyp-info*) *s a*;
align-of TYPE('*b*::*packed-type*) *dvd s*; 0 < *CARD*('*n*); *algn* ≤ *align-td* (*typ-info-t TYPE*('*b*)); *fg-cons xf xfu*⟧
$\Longrightarrow$ *td-packed*
(*ti-typ-combine TYPE*('*b* ['*n* :: *finite*]) *xf xfu algn nm td*)
(*s* + *size-td* (*typ-info-t TYPE*('*b*)) ∗ *CARD*('*n*))
(*max a* (*align-td* (*typ-info-t TYPE*('*b*))))
⟨*proof*⟩

lemma *td-packed-ti-typ-pad-combine-array*:
⟦ *td-packed* (*td*::'*a*::*c-type xtyp-info*) *s a*;
align-of TYPE('*b*::*packed-type*) *dvd s*; 0 < *CARD*('*n*); *algn* ≤ *align-td* (*typ-info-t TYPE*('*b*)); *fg-cons xf xfu* ⟧
$\Longrightarrow$ *td-packed* (*ti-typ-pad-combine TYPE*('*b* ['*n* :: *finite*]) *xf xfu algn nm td*)
(*s* + *size-td* (*typ-info-t TYPE*('*b*)) ∗ *CARD*('*n*))
(*max a* (*align-td* (*typ-info-t TYPE*('*b*))))
⟨*proof*⟩

lemma *td-packed-empty-typ-info*:
td-packed (*empty-typ-info 0 fn*) *0 0*
⟨*proof*⟩

lemmas *td-packed-intros* =
td-packed-final-pad
td-packed-empty-typ-info
td-packed-ti-typ-combine
td-packed-ti-typ-pad-combine
td-packed-ti-typ-combine-array
td-packed-ti-typ-pad-combine-array

end

13.7 Prettier Printing for Programs

theory *PrettyProgs*
imports *Simpl.Vcg*
begin

syntax (**output**)
-Assign :: *'b => 'b => ('a,'p,'f) com* (‹(*2- :==/ -*)› [*30, 30*] *23*)

-seq::*('s,'p,'f) com ⇒ ('s,'p,'f) com ⇒ ('s,'p,'f) com* (‹*-;;//-*› [*20, 21*] *20*)

-While-inv :: *'a bexp => 'a assn => bdy => ('a,'p,'f) com*
(‹(*0WHILE (-)//INV (-)//-*)› [*25, 0, 81*] *71*)

-Do :: *('a,'p,'f) com ⇒ bdy* (‹*DO// (-)//OD*› [*0*] *1000*)

-Cond :: *'a bexp => ('a,'p,'f) com => ('a,'p,'f) com => ('a,'p,'f) com*
(‹(*0IF - THEN// (-)//ELSE// (-)//FI*)› [*0, 0, 0*] *71*)
-Cond-no-else:: *'a bexp => ('a,'p,'f) com => ('a,'p,'f) com*
(‹(*0IF - THEN// (-)//FI*)› [*0, 0*] *71*)

-Try-Catch:: *('a,'p,'f) com ⇒('a,'p,'f) com ⇒ ('a,'p,'f) com*
(‹(*0TRY// (-)//CATCH -//END*)› [*0,0*] *71*)

end

theory *StaticFun*
imports
Main
begin

datatype *('a, 'b) Tree = Node 'a 'b ('a, 'b) Tree ('a, 'b) Tree | Leaf*

primrec
lookup-tree :: *('a, 'b) Tree ⇒ ('a ⇒ 'c :: linorder) ⇒ 'a ⇒ 'b option*
where
lookup-tree Leaf fn x = None
| *lookup-tree (Node y v l r) fn x = (if fn x = fn y then Some v*
else if fn x < fn y then lookup-tree l fn x
else lookup-tree r fn x)

definition *optional-strict-range* :: *('a :: linorder) option ⇒ 'a option ⇒ 'a set*
where
optional-strict-range x y = {z. (x = None ∨ the x < z) ∧ (y = None ∨ z < the y)}

lemma *optional-strict-range-split*:
z ∈ optional-strict-range x y

$\Longrightarrow$ *optional-strict-range x* (*Some z*) = ({..< *z*} ∩ *optional-strict-range x y*)
∧ *optional-strict-range* (*Some z*) *y* = ({*z* <..} ∩ *optional-strict-range x y*)
⟨*proof*⟩

lemma *optional-strict-rangeI*:
z ∈ *optional-strict-range None None*
z < *y* $\Longrightarrow$ *z* ∈ *optional-strict-range None* (*Some y*)
x < *z* $\Longrightarrow$ *z* ∈ *optional-strict-range* (*Some x*) *None*
x < *z* $\Longrightarrow$ *z* < *y* $\Longrightarrow$ *z* ∈ *optional-strict-range* (*Some x*) (*Some y*)
⟨*proof*⟩

definition
tree-eq-fun-in-range :: ('*a*, '*b*) *Tree* ⇒ ('*a* ⇒ '*c* :: *linorder*) ⇒ ('*a* ⇀ '*b*) ⇒ '*c set*
⇒ *bool*
where
tree-eq-fun-in-range T ord f S ≡ ∀ *x*. (*ord x* ∈ *S*) ⟶ *f x* = *lookup-tree T ord x*

lemma *tree-eq-fun-in-range-from-def*:
⟦ *f* ≡ *lookup-tree T ord* ⟧
$\Longrightarrow$ *tree-eq-fun-in-range T ord f* (*optional-strict-range None None*)
⟨*proof*⟩

lemma *tree-eq-fun-in-range-split*:
tree-eq-fun-in-range (*Node z v l r*) *ord f* (*optional-strict-range x y*)
$\Longrightarrow$ *ord z* ∈ *optional-strict-range x y*
$\Longrightarrow$ *tree-eq-fun-in-range l ord f* (*optional-strict-range x* (*Some* (*ord z*)))
∧ *f z* = *Some v*
∧ *tree-eq-fun-in-range r ord f* (*optional-strict-range* (*Some* (*ord z*)) *y*)
⟨*proof*⟩

⟨*ML*⟩

end

theory *IndirectCalls*

imports
PrettyProgs

begin

lemma *hoare-indirect-call-known-proc*:
assumes *spec*: Γ ⊢ *P* (*call-exn init q return result-exn c*) *Q*,*A*
shows Γ⊢ ({*s*. *p s* = *q*} ∩ *P*) (*dynCall-exn f UNIV init p return result-exn c*) *Q*,*A*
⟨*proof*⟩

lemma *hoare-indirect-call-guard*:
assumes *conseq*: $P \subseteq g \cap R$
assumes *spec*: $\Gamma \vdash R$ (*dynCall-exn f UNIV init p return result-exn c*) Q,A
shows $\Gamma\vdash P$ (*dynCall-exn f g init p return result-exn c*) Q,A
⟨*proof*⟩

end

theory *ModifiesProofs*
imports *CLanguage*
begin

definition
modifies-inv-refl :: ('*a* ⇒ '*a set*) ⇒ *bool*
where
modifies-inv-refl P ≡ ∀ *x*. *x* ∈ *P x*

definition
modifies-inv-incl :: ('*a* ⇒ '*a set*) ⇒ *bool*
where
modifies-inv-incl P ≡ ∀ *x y*. *y* ∈ *P x* ⟶ *P y* ⊆ *P x*

definition
modifies-inv-prop :: ('*a* ⇒ '*a set*) ⇒ *bool*
where
modifies-inv-prop P ≡ *modifies-inv-refl P* ∧ *modifies-inv-incl P*

lemma *modifies-inv-prop*:
modifies-inv-refl P ⟹ *modifies-inv-incl P* ⟹ *modifies-inv-prop P*
⟨*proof*⟩

named-theorems *modifies-inv-intros*

locale *modifies-assertion* =
fixes *P* :: '*s* ⇒ '*s set*
assumes *p*: *modifies-inv-prop P*
begin

lemmas *modifies-inv-prop'* =
p[*unfolded modifies-inv-prop-def modifies-inv-refl-def modifies-inv-incl-def*]

lemma *modifies-inv-prop-lift*:
assumes *c*: ⋀σ. $\Gamma,\Theta\vdash_{/F}$ {σ} *c* (*P* σ),(*P* σ)
shows $\Gamma,\Theta\vdash_{/F}$ (*P* σ) *c* (*P* σ),(*P* σ)
⟨*proof*⟩

lemma *modifies-inv-prop-lower*:
assumes *c*: $\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F} (P\ \sigma)\ c\ (P\ \sigma),(P\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}\ c\ (P\ \sigma),(P\ \sigma)$
⟨*proof*⟩

Note that the C-Parser associates sequential composition to the right. So the first statement is typically already an 'atomic' statement (or at least no further sequential composition) that can be solved. We place it as the second precondition because the modifies-tactic follows the canonical order of tactical reasoning and solves the subgoals from the back. So *c1* is already solved before further decomposing *c2*. This keeps the number of subgoals (and thus the overall goal-state) small.

lemma *modifies-inv-Seq* [*modifies-inv-intros*]:
assumes *c*: $\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}\ c2\ (P\ \sigma),(P\ \sigma)\ \bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}\ c1\ (P\ \sigma),(P\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}\ c1\ ;;\ c2\ (P\ \sigma),(P\ \sigma)$
⟨*proof*⟩

lemma *modifies-inv-Cond* [*modifies-inv-intros*]:
assumes *c*: $\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}\ c1\ (P\ \sigma),(P\ \sigma)\ \bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}\ c2\ (P\ \sigma),(P\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}\ Cond\ b\ c1\ c2\ (P\ \sigma),(P\ \sigma)$
⟨*proof*⟩

lemma *modifies-inv-Guard-strip* [*modifies-inv-intros*]:
assumes *c*: $\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/UNIV} \{\sigma\}\ c\ (P\ \sigma),(P\ \sigma)$
shows $\Gamma,\Theta\vdash_{/UNIV} \{\sigma\}\ Guard\ f\ b\ c\ (P\ \sigma),(P\ \sigma)$
⟨*proof*⟩

lemma *modifies-inv-Skip* [*modifies-inv-intros*]:
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}\ SKIP\ (P\ \sigma),(P\ \sigma)$
⟨*proof*⟩

lemma *modifies-inv-Skip'* [*modifies-inv-intros*]:
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}\ SKIP\ (P\ \sigma)$
⟨*proof*⟩

lemma *modifies-inv-whileAnno* [*modifies-inv-intros*]:
assumes *c*: $\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}\ c\ (P\ \sigma),(P\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}\ whileAnno\ b\ I\ V\ c\ (P\ \sigma),(P\ \sigma)$
⟨*proof*⟩

lemma *modifies-inv-While* [*modifies-inv-intros*]:
assumes *c*: $\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}\ c\ (P\ \sigma),(P\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}\ While\ b\ c\ (P\ \sigma),(P\ \sigma)$
⟨*proof*⟩

lemma *modifies-inv-Throw* [*modifies-inv-intros*]:
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *THROW* $(P\ \sigma),(P\ \sigma)$
⟨*proof*⟩

lemma *modifies-inv-Catch* [*modifies-inv-intros*]:
assumes *c*: $\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}$ *c1* $(P\ \sigma),(P\ \sigma)$
$\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}$ *c2* $(P\ \sigma),(P\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *TRY c1 CATCH c2 END* $(P\ \sigma),(P\ \sigma)$
⟨*proof*⟩

lemma *modifies-inv-Catch-all* [*modifies-inv-intros*]:
assumes *1*: $\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}$ *c1* $(P\ \sigma),(P\ \sigma)$
assumes *2*: $\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}$ *c2* $(P\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *TRY c1 CATCH c2 END* $(P\ \sigma)$
⟨*proof*⟩

lemma *modifies-inv-switch-Nil* [*modifies-inv-intros*]:
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *switch v* [] $(P\ \sigma),(P\ \sigma)$
⟨*proof*⟩

lemma *modifies-inv-switch-Cons* [*modifies-inv-intros*]:
assumes *c*: $\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}$ *c* $(P\ \sigma),(P\ \sigma)$
$\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}$ *switch p vcs* $(P\ \sigma),(P\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *switch p* $((v,c)\ \#\ vcs)$ $(P\ \sigma),(P\ \sigma)$
⟨*proof*⟩

end

locale *modifies-state-assertion* = *modifies-assertion P* **for**
P :: $('g,\ 'l,\ 'e,\ 'x)$ *state-scheme* ⇒ $('g,\ 'l,\ 'e,\ 'x)$ *state-scheme set* +
assumes *p*: *modifies-inv-prop P*
begin

lemma *modifies-inv-creturn* [*modifies-inv-intros*]:
assumes *c*: $\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}$ *Basic* $(\lambda s.\ xfu\ (\lambda\text{-}.\ v\ s)\ s)$ $(P\ \sigma),(P\ \sigma)$
$\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}$ *Basic* $(rtu\ (\lambda\text{-}.\ Return))$ $(P\ \sigma),(P\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *creturn rtu xfu v* $(P\ \sigma),(P\ \sigma)$
⟨*proof*⟩

lemma *modifies-inv-creturn-void* [*modifies-inv-intros*]:
assumes *c*: $\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}$ *Basic* $(rtu\ (\lambda\text{-}.\ Return))$ $(P\ \sigma),(P\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *creturn-void rtu* $(P\ \sigma),(P\ \sigma)$
⟨*proof*⟩

lemma *modifies-inv-cbreak* [*modifies-inv-intros*]:
assumes *c*: $\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}$ *Basic* $(rtu\ (\lambda\text{-}.\ Break))$ $(P\ \sigma),(P\ \sigma)$

shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *cbreak rtu* $(P\ \sigma),(P\ \sigma)$
⟨*proof*⟩

lemma *modifies-inv-ccatchbrk* [*modifies-inv-intros*]:
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *ccatchbrk rt* $(P\ \sigma),(P\ \sigma)$
⟨*proof*⟩

lemma *modifies-inv-cgoto* [*modifies-inv-intros*]:
assumes *c*: $\bigwedge\sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}$ *Basic* (*rtu* (λ-. *Goto l*)) $(P\ \sigma),(P\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *cgoto l rtu* $(P\ \sigma),(P\ \sigma)$
⟨*proof*⟩

lemma *modifies-inv-ccatchgoto* [*modifies-inv-intros*]:
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *ccatchgoto l rt* $(P\ \sigma),(P\ \sigma)$
⟨*proof*⟩

lemma *modifies-inv-ccatchreturn* [*modifies-inv-intros*]:
shows $\Gamma,\Theta\vdash_{/F} \{\sigma\}$ *ccatchreturn rt* $(P\ \sigma),(P\ \sigma)$
⟨*proof*⟩

end

lemma *On-Exit-wp*:
assumes *cleanup*: $\Gamma,\Theta\vdash_{/F} R$ *cleanup* $Q,\ A$
assumes *cleanup-catch*: $\Gamma,\Theta\vdash_{/F} B$ *cleanup* $A,\ A$
assumes *c*: $\Gamma,\Theta\vdash_{/F} P\ c\ R,\ B$
shows $\Gamma,\Theta\vdash_{/F} P$ *On-Exit c cleanup* $Q,\ A$
⟨*proof*⟩

lemma *DynCom-fix-pre*: $\forall\, s \in P.\ \Gamma,\Theta\vdash_{/F} \{s\}\ (c\ s)\ Q,A$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F} P$ (*DynCom c*) Q,A
⟨*proof*⟩

lemma $\Gamma,\Theta\vdash_{/F} \{s.\ (\forall\, t.\ (s,\ t) \in r \longrightarrow t \in Q) \wedge (\exists\, t.\ (s,\ t) \in r)\}$ *Spec r* Q,A
⟨*proof*⟩

lemma *hoarep-Spec-fixed*: $(\exists\, t.\ (s,\ t) \in r) \Longrightarrow \Gamma,\Theta\vdash_{/F}\{s\}$ *Spec r* $\{t.\ (s,\ t) \in r\},A$
⟨*proof*⟩

context *stack-heap-state*
begin

lemma *With-Fresh-Stack-Ptr-tight*:
assumes *c*: $\bigwedge s\ p\ d\ vs\ bs.\ s \in P \Longrightarrow vs \in init\ s \Longrightarrow length\ vs = n \Longrightarrow length\ bs = n * size\text{-}of\ TYPE('a) \Longrightarrow$

$(p,\ d) \in$ *stack-allocs n* $\mathcal{S}$ *TYPE('a::mem-type) (htd s)* $\Longrightarrow$
$\Gamma,\Theta\vdash_{/F}\{$*hmem-upd (fold* $(\lambda i.$ *heap-update-padding* $(p +_p$ *int i)* $(vs!i)$ *(take (size-of TYPE('a)) (drop (i * size-of TYPE('a)) bs))) [0..<n]) (htd-upd* $(\lambda$*-. d) s)*$\}$
(c p)
$\{t.\ \forall$ *bs. length bs = n * size-of TYPE('a)* $\longrightarrow$ *hmem-upd (heap-update-list (ptr-val p) bs) (htd-upd (stack-releases n p) t)* $\in Q\}$,
$\{t.\ \forall$ *bs. length bs = n * size-of TYPE('a)* $\longrightarrow$ *hmem-upd (heap-update-list (ptr-val p) bs) (htd-upd (stack-releases n p) t)* $\in A\}$
assumes *no-overflow*: *StackOverflow* $\in F$
shows $\Gamma,\Theta\vdash_{/F}$ *P With-Fresh-Stack-Ptr n init c Q, A*
⟨*proof*⟩

lemma *With-Fresh-Stack-Ptr-tight-wp*:
assumes *conseq*: $\bigwedge$*s p d vs bs. s* $\in$ *P* $\Longrightarrow$ *vs* $\in$ *init s* $\Longrightarrow$ *length vs = n* $\Longrightarrow$ *length bs = n * size-of TYPE('a)* $\Longrightarrow$
$(p,\ d) \in$ *stack-allocs n* $\mathcal{S}$ *TYPE('a::mem-type) (htd s)* $\Longrightarrow$
(hmem-upd (fold $(\lambda i.$ *heap-update-padding* $(p +_p$ *int i)* $(vs!i)$ *(take (size-of TYPE('a)) (drop (i * size-of TYPE('a)) bs))) [0..<n]) (htd-upd* $(\lambda$*-. d) s))* $\in$ *R s p d vs*
assumes *c*: $\bigwedge$*s p d vs bs. s* $\in$ *P* $\Longrightarrow$ *vs* $\in$ *init s* $\Longrightarrow$ *length vs = n* $\Longrightarrow$ *length bs = n * size-of TYPE('a)* $\Longrightarrow$
$(p,\ d) \in$ *stack-allocs n* $\mathcal{S}$ *TYPE('a::mem-type) (htd s)* $\Longrightarrow$
(hmem-upd (fold $(\lambda i.$ *heap-update-padding* $(p +_p$ *int i)* $(vs!i)$ *(take (size-of TYPE('a)) (drop (i * size-of TYPE('a)) bs))) [0..<n]) (htd-upd* $(\lambda$*-. d) s))* $\in$ *R s p d vs* $\Longrightarrow$
$\Gamma,\Theta\vdash_{/F}$*(R s p d vs) (c p)*
$\{t.\ \forall$ *bs. length bs = n * size-of TYPE('a)* $\longrightarrow$ *hmem-upd (heap-update-list (ptr-val p) bs) (htd-upd (stack-releases n p) t)* $\in Q\}$,
$\{t.\ \forall$ *bs. length bs = n * size-of TYPE('a)* $\longrightarrow$ *hmem-upd (heap-update-list (ptr-val p) bs) (htd-upd (stack-releases n p) t)* $\in A\}$
assumes *no-overflow*: *StackOverflow* $\in F$
shows $\Gamma,\Theta\vdash_{/F}$ *P With-Fresh-Stack-Ptr n init c Q, A*
⟨*proof*⟩

Caution: this WP-setup was developed to solve the modified clauses. It might not be the best fit for WP-style reasoning in general. Also note that it is currently not invoked by the automatic modifies-proofs triggerd by the C-parser as *With-Fresh-Stack-Ptr* is first decomposed by the (see the rule below).

⟨*ML*⟩

end

locale *modifies-assertion-stack-heap-state* =
modifies-assertion P + stack-heap-state htd htd-upd hmem hmem-upd $\mathcal{S}$
for *P*::*'s* $\Rightarrow$ *'s set* **and**

htd:: *'s* ⇒ *heap-typ-desc* **and**
htd-upd:: (*heap-typ-desc* ⇒ *heap-typ-desc*) ⇒ *'s* ⇒ *'s* **and**
hmem:: *'s* ⇒ *heap-mem* **and**
hmem-upd:: (*heap-mem* ⇒ *heap-mem*) ⇒ *'s* ⇒ *'s* **and**
$\mathcal{S}$::*addr set* +
assumes *hmem-upd-inv*: ⋀σ *m*. *hmem-upd m* σ ∈ (*P* σ)
assumes *htd-upd-inv*: ⋀σ *d*. *htd-upd d* σ ∈ (*P* σ)
begin

lemma *modifies-With-Fresh-Stack-Ptr* [*modifies-inv-intros*]:
assumes *no-overflow*: *StackOverflow* ∈ *F*
assumes *c*: ⋀σ *p*. Γ,Θ⊢$_{/F}$ {σ} (*c* (*p*::*'a*::*mem-type ptr*)) (*P* σ),(*P* σ)
shows Γ,Θ⊢$_{/F}$ {σ} *With-Fresh-Stack-Ptr n init c* (*P* σ), (*P* σ)
⟨*proof*⟩

end

locale *modifies-assertion-stack-heap-raw-state* =
modifies-assertion P + *stack-heap-raw-state t-hrs t-hrs-update* $\mathcal{S}$
for *P*::*'s* ⇒ *'s set* **and**
t-hrs:: *'s* ⇒ *heap-raw-state* **and**
t-hrs-update:: (*heap-raw-state* ⇒ *heap-raw-state*) ⇒ *'s* ⇒ *'s* **and**
$\mathcal{S}$::*addr set* +
assumes *hrs-upd-inv*: ⋀σ *m*. *t-hrs-update m* σ ∈ (*P* σ)
begin

sublocale *modifies-assertion-stack-heap-state*
P
λ*s*. *hrs-htd* (*t-hrs s*) λ*upd*. *t-hrs-update* (*hrs-htd-update upd*)
λ*s*. *hrs-mem* (*t-hrs s*) λ*upd*. *t-hrs-update* (*hrs-mem-update upd*)
$\mathcal{S}$
⟨*proof*⟩

end

end

13.8 Modelling Local Variables

theory *CLocals*
imports *UMM*
HOL−Library.Code-Binary-Nat
ML-Record-Antiquotation
begin

⟨*ML*⟩
type-synonym *locals* = *nat* ⇒ *byte list*

definition *clookup* :: *nat* ⇒ *locals* ⇒ *'a*::*mem-type* **where**
clookup n l = *from-bytes* (*l n*)

definition *cupdate* :: *nat* ⇒ (*'a*::*mem-type* ⇒ *'a*) ⇒ *locals* ⇒ *locals* **where**
cupdate n f l = *l*(*n* := *to-bytes* (*f* (*from-bytes* (*l* (*n*)))) (*replicate* (*size-of TYPE*(*'a*)) *0*))

lemma *clookup-cupdate-same*[*simp*, *state-simp*]: *clookup n* (*cupdate n f l*) = *f* (*clookup n l*)
⟨*proof*⟩

lemma *clookup-cupdate-same-cond*[*code-simproc state-simp*]:
n = *0* ⟹ *clookup n* (*cupdate 0 f l*) = *f* (*clookup n l*)
n = *0* ⟹ *clookup 0* (*cupdate n f l*) = *f* (*clookup n l*)
n = *1* ⟹ *clookup n* (*cupdate 1 f l*) = *f* (*clookup n l*)
n = *Suc 0* ⟹ *clookup n* (*cupdate* (*Suc 0*) *f l*) = *f* (*clookup n l*)
n = *1* ⟹ *clookup 1* (*cupdate n f l*) = *f* (*clookup n l*)
n = *Suc 0* ⟹ *clookup* (*Suc 0*) (*cupdate n f l*) = *f* (*clookup n l*)
n = *numeral m* ⟹ *clookup n* (*cupdate* (*numeral m*) *f l*) = *f* (*clookup n l*)
n = *numeral m* ⟹ *clookup* (*numeral m*) (*cupdate n f l*) = *f* (*clookup n l*)
⟨*proof*⟩

lemma *clookup-refl-cond*[*code-simproc state-simp*]:
n = *m* ⟹ *clookup n l* = *clookup m l* ⟷ *True*
⟨*proof*⟩

lemma *clookup-cupdate-other*[*code-simproc state-simp*]: *n* ≠ *m* ⟹ *clookup n* (*cupdate m f l*) = (*clookup n l*)
⟨*proof*⟩

lemma *cupdate-compose*[*simp*, *state-simp*]: *cupdate n f* (*cupdate n g l*) = *cupdate n* (*f o g*) *l*
⟨*proof*⟩

lemma *cupdate-compose-cond*[*code-simproc state-simp*]:
n = *0* ⟹ *cupdate n f* (*cupdate 0 g l*) = *cupdate n* (*f o g*) *l*
n = *0* ⟹ *cupdate 0 f* (*cupdate n g l*) = *cupdate n* (*f o g*) *l*
n = *1* ⟹ *cupdate n f* (*cupdate 1 g l*) = *cupdate n* (*f o g*) *l*
n = *Suc 0* ⟹ *cupdate n f* (*cupdate* (*Suc 0*) *g l*) = *cupdate n* (*f o g*) *l*
n = *Suc 0* ⟹ *cupdate* (*Suc 0*) *f* (*cupdate n g l*) = *cupdate n* (*f o g*) *l*
n = *numeral m* ⟹ *cupdate n f* (*cupdate* (*numeral m*) *g l*) = *cupdate n* (*f o g*) *l*
n = *numeral m* ⟹ *cupdate* (*numeral m*) *f* (*cupdate n g l*) = *cupdate n* (*f o g*) *l*
⟨*proof*⟩

lemma *clookup-cupdate-other-numeral*[*simplified*, *simp*, *state-simp*]:

clookup 0 (cupdate 1 f l) = (clookup 0 l)
clookup 0 (cupdate (numeral m) f l) = (clookup 0 l)
clookup 1 (cupdate 0 f l) = (clookup 1 l)
numeral m ≠ (1::nat) ⟹
clookup 1 (cupdate (numeral m) f l) = (clookup 1 l)

clookup (numeral n) (cupdate 0 f l) = (clookup (numeral n) l)
numeral n ≠ (1::nat) ⟹
clookup (numeral n) (cupdate 1 f l) = (clookup (numeral n) l)

n ≠ m ⟹
clookup (numeral n) (cupdate (numeral m) f l) = (clookup (numeral n) l)
⟨*proof*⟩

lemma *cupdate-commute*: *n ≠ m ⟹ cupdate n f (cupdate m g l) = cupdate m g (cupdate n f l)*
⟨*proof*⟩

lemma *cupdate-commute-ordered*[*code-simproc state-simp*]: *n < m ⟹ cupdate n f (cupdate m g l) = cupdate m g (cupdate n f l)*
⟨*proof*⟩

lemma *cupdate-commute-numeral-simp*[*simplified*, *simp*, *state-simp*]:
cupdate 0 f (cupdate 1 g l) = cupdate 1 g (cupdate 0 f l)
cupdate 0 f (cupdate (numeral m) g l) = cupdate (numeral m) g (cupdate 0 f l)

numeral m ≠ (1::nat) ⟹
cupdate 1 f (cupdate (numeral m) g l) = cupdate (numeral m) g (cupdate 1 f l)

n < m ⟹ cupdate (numeral n) f (cupdate (numeral m) g l) = cupdate (numeral m) g (cupdate (numeral n) f l)
⟨*proof*⟩

lemma *const-compose* [*simp*, *state-simp*]:
cupdate n ((λ-. x) ∘ f) = cupdate n (λ-. x)
cupdate n (f ∘ (λ-. x)) = cupdate n (λ-. f x)
⟨*proof*⟩

lemma *K-eq-cong*: *((λ-. x) = (λ-. y)) ⟷ x = y*
⟨*proof*⟩

⟨*ML*⟩

named-theorems *locals*

consts *clocals-string-embedding* :: *string ⇒ nat*
consts *exit-'*::*nat*

definition *global-exn-var-clocal* = *clocals-string-embedding ''global-exn-var''*

bundle *clocals-string-embedding*
begin
notation *clocals-string-embedding* (‹$\mathcal{S}$›)
end

⟨*ML*⟩

nonterminal *localsupdbinds* **and** *localsupdbind*

syntax
-localsupdbind :: *'a* ⇒ *'a* ⇒ *localsupdbind*
(‹(‹*indent=2 notation=‹infix localsupdbind›*›- :=$_{\mathcal{L}}$/ -)›)
:: *localsupdbind* ⇒ *localsupdbinds* (‹-›)
-localsupdbinds:: *localsupdbind* ⇒ *localsupdbinds* ⇒ *localsupdbinds*
(‹(‹*open-block notation=‹mixfix localsupdbinds›*›-,/ -)›)

syntax
-statespace-lookup :: *locals* ⇒ *'name* ⇒ *'c*
(‹(‹*open-block notation=‹infix statespace-lookup›*›- · -)› [*60*, *60*] *60*)
-statespace-locals-lookup :: (*'g*, *locals*, *'e*, *'x*) *state-scheme* ⇒ *'name* ⇒ *'c*
(‹(‹*open-block notation=‹infix statespace-locals-lookup›*›- ·$_{\mathcal{L}}$ -)› [*60*, *60*] *60*)

-statespace-update :: *locals* ⇒ *'name* ⇒ (*'c* ⇒ *'c*) ⇒ *locals*
-statespace-updates :: *locals* ⇒ *updbinds* ⇒ *locals*
(‹(‹*open-block notation=‹mixfix statespace-updates›*›-⟨-⟩)› [*900*, *0*] *900*)

-statespace-locals-update :: (*'g*, *locals*, *'e*, *'x*) *state-scheme* ⇒ *'name* ⇒ (*'c* ⇒ *'c*) ⇒ (*'g*, *locals*, *'e*, *'x*) *state-scheme*
-statespace-locals-updates :: *locals* ⇒ *localsupdbinds* ⇒ *locals*
(‹(‹*open-block notation=‹mixfix statespace-locals-updates›*›-⟨-⟩)› [*900*, *0*] *900*)

-statespace-locals-map ::
'name ⇒ (*'c* ⇒ *'c*) ⇒ (*'g*, *locals*, *'e*, *'x*) *state-scheme* ⇒ (*'g*, *locals*, *'e*, *'x*) *state-scheme*
(‹(‹*indent=2 notation=‹infix statespace-locals-map›*›-:=$_{\mathcal{L}}$/ -)› [*1000*, *1000*] *1000*)

syntax-consts
-statespace-updates ⇌ *fun-upd* **and**
-statespace-locals-updates ⇌ *locals-update*

translations
-statespace-updates f (*-updbinds b bs*) ==

-statespace-updates (*-statespace-updates f b*) *bs*
$s\langle x := y\rangle$ == *-statespace-update s x y*

-statespace-locals-updates f (*-localsupdbinds b bs*) ==
-statespace-locals-updates (*-statespace-locals-updates f b*) *bs*
$s\langle x :=_{\mathcal{L}} y\rangle$ == *-statespace-locals-update s x y*

$\langle ML \rangle$

end

Chapter 14

Setup Lex / Yacc and Translation from C to Simpl

theory *CTranslationSetup*
imports
UMM
PackedTypes
PrettyProgs
StaticFun
IndirectCalls
ModifiesProofs
HOL−Eisbach.Eisbach
ML-Record-Antiquotation
Option-Scanner
Misc-Antiquotation
MkTermAntiquote
TermPatternAntiquote
CLocals
keywords
cond-sorry-modifies-proofs :: *thy-decl*
and
mllex
mlyacc:: *thy-load*
begin

⟨*ML*⟩

definition *coerce*::*'a*::*mem-type* ⇒ *'b*::*mem-type* **where**
coerce v = *from-bytes* (*to-bytes-p v*)

syntax
-coerce :: *type* ⇒ *type* ⇒ *logic*
(‹(‹*indent=1 notation=‹mixfix COERCE›› COERCE*/(‹*indent=1 notation=‹infix coerce›› '*(- → -')))›)

syntax-consts
-coerce == *coerce*
translations
COERCE('*a* → '*b*) => *CONST coerce* :: ('*a* ⇒ '*b*)
⟨*ML*⟩

lemma *coerce-id*[*simp*]:
shows *coerce v* = *v*
⟨*proof*⟩

lemma *coerce-cancel-packed*[*simp*]:
fixes *v*::'*a*::*packed-type*
assumes *sz-eq*: *size-of* (*TYPE*('*a*)) = *size-of* (*TYPE*('*b*))
shows *coerce* ((*coerce v*)::'*b*::*packed-type*) = *v*
⟨*proof*⟩

definition *coerce-map*:: ('*a*::*mem-type* ⇒ '*a*) ⇒ ('*b*::*mem-type* ⇒ '*b*) **where**
coerce-map f v = *coerce* (*f* (*coerce v*))

lemma *coerce-map-id*[*simp*]: *coerce-map f* (*coerce v*) = *f v*
⟨*proof*⟩

lemma *coerce-coerce-map-cancel-packed*[*simp*]:
fixes *f*::'*a*::*packed-type* ⇒ '*a*
fixes *v*::'*b*::*packed-type*
assumes *sz-eq*[*simp*]: *size-of* (*TYPE*('*a*)) = *size-of* (*TYPE*('*b*))
shows ((*coerce* (*coerce-map f v*))::'*a*) = *f* (*coerce v*)
⟨*proof*⟩

named-theorems *global-const-defs* **and**
global-const-array-selectors **and**
global-const-non-array-selectors **and**
global-const-selectors

named-theorems *fun-ptr-simps*
named-theorems *fun-ptr-intros*
named-theorems *fun-ptr-distinct*
named-theorems *fun-ptr-subtree*
named-theorems *disjoint-*$\mathcal{G}$*-*$\mathcal{S}$

We integrate mllex and mlyacc directly into Isabelle:

- We compile the SML files according to the description in `tools/mlyacc/src/FILES`
- We export the necessary signatures and structures to the Isabelle/ML environment.

- As mllex / mlyacc operate directly on files we invoke them on temporary files and redirect stdout / stderr to display the messages within PIDE. We wrap this in Isabelle commands **mllex**, **mlyacc**.

⟨*ML*⟩

primrec *map-of-default* ::
  (*'p* ⇒ *'a*) ⇒ (*'p* ∗ *'a*) *list* ⇒ *'p* ⇒ *'a*
where
  map-of-default d [] *x* = *d x*
| *map-of-default d* (*x* # *xs*) *x'* = (*if fst x* = *x' then snd x else map-of-default d xs x'*)

lemma *map-of-default-append*: ‹*map-of-default d* (*xs* @ *ys*) = *map-of-default* (*map-of-default d ys*) *xs*›
  ⟨*proof*⟩

lemma *map-of-default-map-of-conv*:
  ‹*map-of-default d xs p* = (*case map-of xs p of Some f* ⇒ *f* | *None* ⇒ *d p*)›
  ⟨*proof*⟩

lemma *map-of-default-fallthrough*:
  p ∉ *set* (*map fst xs*) ⟹ *map-of-default d xs p* = *d p*
  ⟨*proof*⟩

lemma *map-of-default-distinct*:
  assumes *distinct* (*map fst xs*)
  shows *list-all* (λ(*p*, *f*). *map-of-default d xs p* = *f*) *xs*
  ⟨*proof*⟩

lemma *map-of-default-default-conv*:
  assumes *list-all* (λ(*p*, *f*). *d p* = *f*) *xs*
  shows *map-of-default d xs* = *d*
  ⟨*proof*⟩

lemma *map-of-default-monotone-cons*[*partial-function-mono*]:
  assumes *f1* [*partial-function-mono*]: *monotone R X f1*
  assumes [*partial-function-mono*]: *monotone R X* (λ*f*. *map-of-default d* (*xs f*) *p*)
  shows *monotone R X* (λ*f*. *map-of-default d* ((*p1*, *f1 f*)#*xs f*) *p*)
  ⟨*proof*⟩

hide-const (**open**) *StaticFun.Node*

primrec *tree-of* :: *'a list* ⇒ *'a tree*
where
  tree-of [] = *Tip*
| *tree-of* (*x*#*xs*) = *Node* (*Tip*) *x False* (*tree-of xs*)

lemma *set-of-tree-of*: *set-of* (*tree-of xs*) = *set xs*
⟨*proof*⟩

lemma *all-distinct-tree-of*:
assumes *all-distinct* (*tree-of xs*)
shows *distinct xs*
⟨*proof*⟩

lemma *all-distinct-tree-of′*:
all-distinct t ⟹ *tree-of xs* ≡ *t* ⟹ *distinct xs*
⟨*proof*⟩

lemma *map-of-default-fallthrough′*:
map fst xs ≡ *ps* ⟹ *tree-of ps* ≡ *t* ⟹ *p* ∉ *set-of t* ⟹ *map-of-default d xs p* = *d p*
⟨*proof*⟩

primrec *list-of* :: *'a tree* ⇒ *'a list*
where
list-of Tip = []
| *list-of* (*Node l x d r*) = *list-of l* @ (*if d then* [] *else* [*x*]) @ *list-of r*

lemma *list-of-tree-of-conv* [*simp*]: *list-of* (*tree-of xs*) = *xs*
⟨*proof*⟩

lemma *set-list-of-set-of-conv*: *set* (*list-of t*) = *set-of t*
⟨*proof*⟩

lemma *all-distinct-list-of*:
assumes *all-distinct t*
shows *distinct* (*list-of t*)
⟨*proof*⟩

lemma *map-of-default-distinct-lookup-list-all*:
distinct (*map fst xs*) ⟹ *list-all* (λ(*p*, *f*). *map-of-default d xs p* = *f*) *xs*
⟨*proof*⟩

lemma *map-of-default-distinct-lookup-list-all′*:
assumes *ps*: *map fst xs* ≡ *ps*
assumes *t*: *tree-of ps* ≡ *t*
assumes *dist*: *all-distinct t*
shows *list-all* (λ(*p*, *f*). *map-of-default d xs p* = *f*) *xs*
⟨*proof*⟩

lemma *map-of-default-distinct-lookup-list-all″*:
assumes *t*: *list-of t* ≡ *ps*
assumes *ps*: *map fst xs* ≡ *ps*
assumes *dist*: *all-distinct t*

shows *list-all* (λ(*p*, *f*). *map-of-default d xs p* = *f*) *xs*
⟨*proof*⟩

lemma *map-of-default-other-lookup-list-all*:
set ps ∩ *set* (*map fst xs*) = {} ⟹ *list-all* (λ*p*. *map-of-default d xs p* = *d p*) *ps*
⟨*proof*⟩

lemma *delete-Some-subset*: *DistinctTreeProver.delete x t* = *Some t'* ⟹ *set-of t* ⊆ {*x*} ∪ *set-of t'*
⟨*proof*⟩

lemma *delete-Some-set-of-union*:
assumes *del*: *DistinctTreeProver.delete x t* = *Some t'* **shows** *set-of t* = {*x*} ∪ *set-of t'*
⟨*proof*⟩

primrec *undeleted* :: *'a tree* ⇒ *bool*
where
undeleted Tip = *True*
| *undeleted* (*Node l y d r*) = (¬ *d* ∧ *undeleted l* ∧ *undeleted r*)

lemma *undeleted-tree-of*[*simp*]: *undeleted* (*tree-of xs*)
⟨*proof*⟩

lemma *subtract-union-subset*:
subtract t_1 t_2 = *Some t* ⟹ *undeleted* t_1 ⟹ *set-of* t_2 ⊆ *set-of* t_1 ∪ *set-of t*
⟨*proof*⟩

lemma *subtract-union-eq*:
assumes *sub*: *subtract* t_1 t_2 = *Some t*
assumes *und*: *undeleted* t_1
shows *set-of* t_2 = *set-of* t_1 ∪ *set-of t*
⟨*proof*⟩

lemma *subtract-empty*:
assumes *sub*: *subtract* t_1 t_2 = *Some t*
assumes *und*: *undeleted* t_1
assumes *empty*: *set-of t* = {}
shows *set-of* t_1 = *set-of* t_2
⟨*proof*⟩

lemma *map-of-default-other-lookup-Ball*:
assumes *ps*: *list-of t* ≡ *ps*

assumes *map-fst*: *map fst xs* ≡ *ps*
assumes *sub*: *subtract t t-all* = *Some t-sub*
shows ∀ *p* ∈ *set-of t-sub*. *map-of-default d xs p* = *d p*
⟨*proof*⟩

lemma *subtract-set-of-exchange-first*:
assumes *sub1*: *subtract* t_1 *t* = *Some* t'
assumes *sub2*: *subtract* t_2 *t* = *Some* t''
assumes *und1*: *undeleted* t_1
assumes *und2*: *undeleted* t_2
assumes *seq*: *set-of* t_1 = *set-of* t_2
shows *set-of* t' = *set-of* t''
⟨*proof*⟩

lemma *TWO*: *Suc* (*Suc 0*) = *2* ⟨*proof*⟩

definition
fun-addr-of :: *int* ⇒ *unit ptr* **where**
fun-addr-of i ≡ *Ptr* (*word-of-int i*)

definition
ptr-range :: *'a*::*c-type ptr* ⇒ *addr set* **where**
ptr-range p ≡ {*ptr-val* (*p*::*'a ptr*) ..< *ptr-val p* + *word-of-int*(*int*(*size-of* (*TYPE*(*'a*))))
}

lemma *guarded-spec-body-wp* [*vcg-hoare*]:
P ⊆ {*s*. (∀ *t*. (*s*,*t*) ∈ *R* ⟶ *t* ∈ *Q*) ∧ (*Ft* ∉ *F* ⟶ (∃ *t*. (*s*,*t*) ∈ *R*))}
⟹ Γ,Θ⊢$_{/F}$ *P* (*guarded-spec-body Ft R*) *Q*, *A*
⟨*proof*⟩

named-theorems *recursive-records-fold-congs* **and** *recursive-records-split-all-eqs*

⟨*ML*⟩

mllex *StrictC.lex*
mlyacc *StrictC.grm*

⟨*ML*⟩
term *word-of-int*
⟨*ML*⟩

context
begin

⟨*ML*⟩
end

⟨*ML*⟩

named-theorems *enum-defs*

⟨*ML*⟩

declare *typ-info-word* [*simp del*]
declare *typ-info-ptr* [*simp del*]

lemma *valid-call-Spec-eq-subset*:
Γ' *procname* = *Some* (*Spec R*) ⟹
HoarePartialDef.valid Γ' *NF P* (*Call procname*) *Q A* = (*P* ⊆ *fst* ‘ *R* ∧ (*R* ⊆ (− *P*) × *UNIV* ∪ *UNIV* × *Q*))
⟨*proof*⟩

lemma *creturn-wp* [*vcg-hoare*]:
assumes *P* ⊆ {*s*. (*exnupd* (λ-. *Return*)) (*rvupd* (λ-. *v s*) *s*) ∈ *A*}
shows Γ,Θ⊢$_{/F}$ *P creturn exnupd rvupd v Q*, *A*
⟨*proof*⟩

lemma *creturn-wp-total* [*vcg-hoare*]:
assumes *P* ⊆ {*s*. (*exnupd* (λ-. *Return*)) (*rvupd* (λ-. *v s*) *s*) ∈ *A*}
shows Γ,Θ⊢$_{t/F}$ *P creturn exnupd rvupd v Q*, *A*
⟨*proof*⟩

lemma *creturn-void-wp* [*vcg-hoare*]:
assumes *P* ⊆ {*s*. (*exnupd* (λ-. *Return*)) *s* ∈ *A*}
shows Γ,Θ⊢$_{/F}$ *P creturn-void exnupd Q*, *A*
⟨*proof*⟩

lemma *creturn-void-wp-total* [*vcg-hoare*]:
assumes *P* ⊆ {*s*. (*exnupd* (λ-. *Return*)) *s* ∈ *A*}
shows Γ,Θ⊢$_{t/F}$ *P creturn-void exnupd Q*, *A*
⟨*proof*⟩

lemma *cbreak-wp* [*vcg-hoare*]:
assumes *P* ⊆ {*s*. (*exnupd* (λ-. *Break*)) *s* ∈ *A*}
shows Γ,Θ⊢$_{/F}$ *P cbreak exnupd Q*, *A*
⟨*proof*⟩

lemma *cbreak-wp-totoal* [*vcg-hoare*]:
assumes *P* ⊆ {*s*. (*exnupd* (λ-. *Break*)) *s* ∈ *A*}

shows $\Gamma,\Theta\vdash_{t/F} P$ *cbreak exnupd* Q, A
⟨*proof*⟩

lemma *ccatchbrk-wp* [*vcg-hoare*]:
assumes $P \subseteq \{s.\ (exnupd\ s = Break \longrightarrow s \in Q) \wedge$
$(exnupd\ s \neq Break \longrightarrow s \in A)\}$
shows $\Gamma,\Theta\vdash_{/F} P$ *ccatchbrk exnupd* Q, A
⟨*proof*⟩

lemma *ccatchbrk-wp-total* [*vcg-hoare*]:
assumes $P \subseteq \{s.\ (exnupd\ s = Break \longrightarrow s \in Q) \wedge$
$(exnupd\ s \neq Break \longrightarrow s \in A)\}$
shows $\Gamma,\Theta\vdash_{t/F} P$ *ccatchbrk exnupd* Q, A
⟨*proof*⟩

lemma *cgoto-wp* [*vcg-hoare*]:
assumes $P \subseteq \{s.\ (exnupd\ (\lambda\text{-}.\ Goto\ l))\ s \in A\}$
shows $\Gamma,\Theta\vdash_{/F} P$ *cgoto l exnupd* Q, A
⟨*proof*⟩

lemma *cgoto-wp-total* [*vcg-hoare*]:
assumes $P \subseteq \{s.\ (exnupd\ (\lambda\text{-}.\ Goto\ l))\ s \in A\}$
shows $\Gamma,\Theta\vdash_{t/F} P$ *cgoto l exnupd* Q, A
⟨*proof*⟩

lemma *ccatchgoto-wp* [*vcg-hoare*]:
assumes $P \subseteq \{s.\ (exnupd\ s = Goto\ l \longrightarrow s \in Q) \wedge$
$(exnupd\ s \neq Goto\ l \longrightarrow s \in A)\}$
shows $\Gamma,\Theta\vdash_{/F} P$ *ccatchgoto l exnupd* Q, A
⟨*proof*⟩

lemma *ccatchgoto-wp-total* [*vcg-hoare*]:
assumes $P \subseteq \{s.\ (exnupd\ s = Goto\ l \longrightarrow s \in Q) \wedge$
$(exnupd\ s \neq Goto\ l \longrightarrow s \in A)\}$
shows $\Gamma,\Theta\vdash_{t/F} P$ *ccatchgoto l exnupd* Q, A
⟨*proof*⟩

lemma *ccatchreturn-wp* [*vcg-hoare*]:
assumes $P \subseteq \{s.\ (is\text{-}local\ (exnupd\ s) \longrightarrow s \in Q) \wedge$
$(\neg\ is\text{-}local\ (exnupd\ s) \longrightarrow s \in A)\}$
shows $\Gamma,\Theta\vdash_{/F} P$ *ccatchreturn exnupd* Q, A
⟨*proof*⟩

lemma *ccatchreturn-wp-total* [*vcg-hoare*]:
assumes $P \subseteq \{s.\ (is\text{-}local\ (exnupd\ s) \longrightarrow s \in Q) \wedge$
$(\neg\ is\text{-}local\ (exnupd\ s) \longrightarrow s \in A)\}$
shows $\Gamma,\Theta\vdash_{t/F} P$ *ccatchreturn exnupd* Q, A
⟨*proof*⟩

lemma *cexit-wp* [*vcg-hoare*]:
assumes $P \subseteq \{s.\ exnupd\ s \in A\}$
shows $\Gamma,\Theta\vdash_{/F} P\ cexit\ exnupd\ Q,\ A$
⟨*proof*⟩

lemma *cexit-wp-total* [*vcg-hoare*]:
assumes $P \subseteq \{s.\ exnupd\ s \in A\}$
shows $\Gamma,\Theta\vdash_{t/F} P\ cexit\ exnupd\ Q,\ A$
⟨*proof*⟩

lemma *cchaos-wp* [*vcg-hoare*]:
assumes $P \subseteq \ \{s.\ \forall x.\ (v\ x\ s) \in Q\ \}$
shows $\Gamma,\Theta\vdash_{/F} P\ cchaos\ v\ Q,\ A$
⟨*proof*⟩

lemma *cchaos-wp-total* [*vcg-hoare*]:
assumes $P \subseteq \ \{s.\ \forall x.\ (v\ x\ s) \in Q\ \}$
shows $\Gamma,\Theta\vdash_{t/F} P\ cchaos\ v\ Q,\ A$
⟨*proof*⟩

lemma *lvar-nondet-init-wp* [*vcg-hoare*]:
$P \subseteq \{s.\ \forall v.\ (upd\ (\lambda\text{-}.\ v))\ s \in Q\} \Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ lvar\text{-}nondet\text{-}init\ upd\ Q,\ A$
⟨*proof*⟩

lemma *lvar-nondet-init-wp-total* [*vcg-hoare*]:
$P \subseteq \{s.\ \forall v.\ (upd\ (\lambda\text{-}.\ v))\ s \in Q\} \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P\ lvar\text{-}nondet\text{-}init\ upd\ Q,\ A$
⟨*proof*⟩

lemma *Seq-propagate-precond*:
$[\![\Gamma,\Theta\vdash_{/F} P\ c_1\ P,A;\ \Gamma,\Theta\vdash_{/F} P\ c_2\ Q,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{/F} P\ (Seq\ c_1\ c_2)\ Q,A$
⟨*proof*⟩

lemma *Seq-propagate-precond-total*:
$[\![\Gamma,\Theta\vdash_{t/F} P\ c_1\ P,A;\ \Gamma,\Theta\vdash_{t/F} P\ c_2\ Q,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{t/F} P\ (Seq\ c_1\ c_2)\ Q,A$
⟨*proof*⟩

lemma *mem-safe-lvar-init* [*simp*,*intro*]:
assumes *upd*: $\bigwedge g\ v\ s.\ globals\text{-}update\ g\ (upd\ (\lambda\text{-}.\ v)\ s) = upd\ (\lambda\text{-}.\ v)\ (globals\text{-}update\ g\ s)$
assumes *acc*: $\bigwedge v\ s.\ globals\ (upd\ (\lambda\text{-}.\ v)\ s) = globals\ s$
shows *mem-safe* (*lvar-nondet-init upd*) *x*
⟨*proof*⟩

lemma *intra-safe-lvar-nondet-init* [*simp*]:
intra-safe (*lvar-nondet-init upd* :: (('*a*::*heap-state-type*','*l*,'*e*,'*x*) *state-scheme*,'*p*,'*f*) *com*) =
(∀Γ. *mem-safe* (*lvar-nondet-init upd* :: (('*a*::*heap-state-type*','*l*,'*e*,'*x*) *state-scheme*,'*p*,'*f*)

com)
(Γ :: ((*'a*,*'l*,*'e*,*'x*) *state-scheme*,*'p*,*'f*) *body*))
⟨*proof*⟩

lemma *proc-deps-lvar-nondet-init* [*simp*]:
proc-deps (*lvar-nondet-init upd*) Γ = {}
⟨*proof*⟩

declare *word-neq-0-conv*[*simp*]

declare [[*hoare-use-generalise*=*true*]]
end

theory *Array-Selectors*
imports
CTranslationSetup
keywords *array-selectors* :: *thy-defn*
begin

named-theorems *array-selectors-simps*

lemmas [*array-selectors-simps*] =— numerals
Arrays.arr-fupdate-same
Arrays.arr-fupdate-other
Numeral-Type.card-num0
Numeral-Type.card-num1
Numeral-Type.card-bit0
Numeral-Type.card-bit1
Nat.One-nat-def
Nat.mult-Suc-right
Nat.mult-0-right
Nat.Suc-not-Zero
Num.Suc-numeral
Num.eq-numeral-Suc
Num.Suc-eq-numeral
Num.less-Suc-numeral
Num.mult-num-simps
Num.add-num-simps
Num.pred-numeral-simps
Num.numeral-times-numeral
Num.num.distinct
Num.num.inject
Nat.add-0-right
Num.arith-simps
Num.more-arith-simps
Num.rel-simps

Nat.Zero-not-Suc

lemmas [*array-selectors-simps*] =
comp-apply
fcp-beta

⟨*ML*⟩

experiment
begin

definition *my-array* ≡ *fupdate 3*
(*apfst* (λ-. *0x30*) *o apsnd* (λ-. *43*))
(*fupdate 2* (*apfst* (λ-. *0x20*) *o apsnd* (λ-. *42*))
(*fupdate 1* (*apfst* (λ-. *0x10*) *o apsnd* (λ-. *41*))
(*fupdate 0* (*apfst* (λ-. *0x0*) *o apsnd* (λ-. *40*))
((*ARRAY* -. (*0*, *0*))::(*32 word* × *nat*)[*4*]))))

lemmas [*array-selectors-simps*] =
apfst-conv
apsnd-conv
— in applications, the update functions are from the recursive record package, therefore the recursive record package simpset is included by default

array-selectors (*no-recursive-record-simpset*)— does not make a difference here
my-array-sels **is** *my-array-def*

lemma *my-array*.[*0*] ≡ (*0*, *40*)
my-array.[*1*] ≡ (*0x10*, *41*)
my-array.[*Suc 0*] ≡ (*0x10*, *41*)
my-array.[*2*] ≡ (*0x20*, *42*)
my-array.[*3*] ≡ (*0x30*, *43*)
⟨*proof*⟩

end

end

theory *CTranslation*
imports
CTranslationSetup
Array-Selectors
keywords
new-C-include-dir:: *thy-decl*
and
include-C-file
install-C-types
install-C-file :: *thy-load*

begin
⟨*ML*⟩

end

Chapter 15

Misc. Lemmas

theory *TypHeapLib*
imports *CTranslation*
begin

15.1 Abbreviations and helpers

definition *is-an-abbreviation* ≡ *True*

abbreviation
  clift ≡ *lift-t c-guard*

lemma *clift-def*: *is-an-abbreviation* ⟨*proof*⟩

15.2 Basic operations

15.2.1 clift

lemma *c-guard-clift*:
  clift hp p = *Some x* ⟹ *c-guard p*
  ⟨*proof*⟩

lemma *clift-heap-update*:
  fixes *p* :: *'a* :: *mem-type ptr*
  shows *hrs-htd hp* ⊨$_t$ *p* ⟹ *clift* (*hrs-mem-update* (*heap-update p v*) *hp*) = (*clift hp*)(*p* ↦ *v*)
  ⟨*proof*⟩

lemma *clift-heap-update-same*:
  fixes *p* :: *'a* :: *mem-type ptr*
  shows ⟦ *hrs-htd hp* ⊨$_t$ *p*; *typ-uinfo-t TYPE('a)* ⊥$_t$ *typ-uinfo-t TYPE('b)* ⟧
  ⟹ *clift* (*hrs-mem-update* (*heap-update p v*) *hp*) = (*clift hp* :: *'b* :: *mem-type typ-heap*)
  ⟨*proof*⟩

lemmas *clift-heap-update-same-td-name* = *clift-heap-update-same* [*OF* - *tag-disj-via-td-name*, *unfolded pad-typ-name-def*]

15.2.2 *h-val*

lemmas *h-val-clift* = *lift-t-lift* [**where** *g* = *c-guard*, *unfolded CTypesDefs.lift-def*, *simplified*]
lemma *h-val-clift'*:
clift hp p = *Some v* ⟹ *h-val* (*hrs-mem hp*) *p* = *v*
⟨*proof*⟩

15.2.3 *h-t-valid*

lemma *clift-Some-eq-valid*:
(∃ *v*. *clift hp p* = *Some v*) = (*hrs-htd hp* ⊨$_t$ *p*)
⟨*proof*⟩

lemma *h-t-valid-clift-Some-iff*:
(*hrs-htd hp* ⊨$_t$ *p*) = (∃ *v*. *clift hp p* = *Some v*)
⟨*proof*⟩

lemma *h-t-valid-clift*:
clift hp p = *Some v* ⟹ *hrs-htd hp* ⊨$_t$ *p*
⟨*proof*⟩

lemma *c-guard-h-t-valid*:
hrs-htd hp ⊨$_t$ *p* ⟹ *c-guard p*
⟨*proof*⟩

15.3 *field-lvalue*

15.3.1 *heap-update*

lemma *heap-update-field*:
⟦*field-ti TYPE*('*a* :: *packed-type*) *f* = *Some t*; *c-guard p*;
export-uinfo t = *export-uinfo* (*typ-info-t TYPE*('*b* :: *packed-type*))⟧
⟹ *heap-update* (*Ptr* &(*p*→*f*) :: '*b ptr*) *v hp* =
heap-update p (*update-ti t* (*to-bytes-p v*) (*h-val hp p*)) *hp*
⟨*proof*⟩

lemma *heap-update-field'*:
⟦*field-ti TYPE*('*a* :: *packed-type*) *f* = *Some t*; *c-guard p*;
export-uinfo t = *export-uinfo* (*typ-info-t TYPE*('*b* :: *packed-type*))⟧
⟹ *heap-update* (*Ptr* &(*p*→*f*) :: '*b ptr*) *v hp* =
heap-update p (*update-ti-t t* (*to-bytes-p v*) (*h-val hp p*)) *hp*
⟨*proof*⟩

lemma *heap-update-field-hrs*:

fixes $p :: 'a :: packed\text{-}type\ ptr$ **and** $v :: 'b :: packed\text{-}type$
shows $[\![field\text{-}ti\ TYPE('a)\ f = Some\ t;\ c\text{-}guard\ p;$
$export\text{-}uinfo\ t = export\text{-}uinfo\ (typ\text{-}info\text{-}t\ TYPE('b))]\!]$
$\Longrightarrow hrs\text{-}mem\text{-}update\ (heap\text{-}update\ (Ptr\ \&(p\rightarrow f))\ v)\ hp =$
$hrs\text{-}mem\text{-}update\ (heap\text{-}update\ p\ (update\text{-}ti\text{-}t\ t\ (to\text{-}bytes\text{-}p\ v)\ (h\text{-}val\ (hrs\text{-}mem\ hp)\ p)))\ hp$
⟨*proof*⟩

lemma *heap-update-field-ext*:
$[\![field\text{-}ti\ TYPE('a :: packed\text{-}type)\ f = Some\ t;\ c\text{-}guard\ p;$
$export\text{-}uinfo\ t = export\text{-}uinfo\ (typ\text{-}info\text{-}t\ TYPE('b :: packed\text{-}type))]\!]$
$\Longrightarrow heap\text{-}update\ (Ptr\ \&(p\rightarrow f) :: 'b\ ptr) =$
$(\lambda v\ hp.\ heap\text{-}update\ p\ (update\text{-}ti\ t\ (to\text{-}bytes\text{-}p\ v)\ (h\text{-}val\ hp\ p))\ hp)$
⟨*proof*⟩

15.3.2 *c-guard*

lemma *c-guard-field*:
$[\![c\text{-}guard\ (p :: 'a :: mem\text{-}type\ ptr);\ field\text{-}ti\ TYPE('a :: mem\text{-}type)\ f = Some\ t;$
$export\text{-}uinfo\ t = export\text{-}uinfo\ (typ\text{-}info\text{-}t\ TYPE('b :: mem\text{-}type))]\!]$
$\Longrightarrow c\text{-}guard\ (Ptr\ \&(p\rightarrow f) :: 'b :: mem\text{-}type\ ptr)$
⟨*proof*⟩

15.3.3 clift

lemma *clift-field*:
fixes $v :: 'a :: mem\text{-}type$ **and** $p :: 'a :: mem\text{-}type\ ptr$
assumes *lf*: $clift\ hp\ p = Some\ v$
and *fl*: $field\text{-}ti\ TYPE('a)\ f = Some\ t$
and *eu*: $export\text{-}uinfo\ t = export\text{-}uinfo\ (typ\text{-}info\text{-}t\ TYPE('b :: mem\text{-}type))$
shows $clift\ hp\ (Ptr\ \&(p\rightarrow f) :: 'b :: mem\text{-}type\ ptr) = Some\ (from\text{-}bytes\ (access\text{-}ti_0\ t\ v))$
⟨*proof*⟩

Updates

lemma *clift-field-update*:
fixes $val :: 'b :: mem\text{-}type$ **and** $ptr :: 'a :: mem\text{-}type\ ptr$
assumes *fl*: $field\text{-}ti\ TYPE('a)\ f = Some\ t$
and *eu*: $export\text{-}uinfo\ t = export\text{-}uinfo\ (typ\text{-}info\text{-}t\ TYPE('b))$
and *cl*: $clift\ hp\ ptr = Some\ z$
shows $(clift\ (hrs\text{-}mem\text{-}update\ (heap\text{-}update\ (Ptr\ \&(ptr\rightarrow f))\ val)\ hp)) =$
$(clift\ hp)(ptr \mapsto field\text{-}update\ (field\text{-}desc\ t)\ (to\text{-}bytes\text{-}p\ val)\ z)$
(**is** *?LHS = ?RHS*)
⟨*proof*⟩

lemma *clift-field-update-padding*:
fixes $val :: 'b :: mem\text{-}type$ **and** $ptr :: 'a :: mem\text{-}type\ ptr$
assumes *fl*: $field\text{-}ti\ TYPE('a)\ f = Some\ t$
and *eu*: $export\text{-}uinfo\ t = export\text{-}uinfo\ (typ\text{-}info\text{-}t\ TYPE('b))$

and *cl*: *clift hp ptr* = *Some z*
and *lbs*: *length bs* = *size-of TYPE('b)*
shows (*clift* (*hrs-mem-update* (*heap-update-padding* (*Ptr* &(*ptr*→*f*)) *val bs*) *hp*))
=
(*clift hp*)(*ptr* ↦ *field-update* (*field-desc t*) (*to-bytes-p val*) *z*)
(**is** *?LHS* = *?RHS*)
⟨*proof*⟩

15.3.4 cparent

definition
cparent :: ('*a* :: *c-type*) *ptr* ⇒ *string list* ⇒ ('*b* :: *c-type*) *ptr*
where
cparent p fs ≡ *THE p'*. *p* = *Ptr* &(*p'*→*fs*)

lemma *cparent-field* [*simp*]:
cparent (*Ptr* &(*p*→*fs*)) *fs* = *p*
⟨*proof*⟩

lemma *cparent-def2*:
fixes *p* :: '*b* :: *c-type ptr*
shows *cparent p f* ≡ (*Ptr* (*ptr-val p* − *of-nat* (*field-offset TYPE*('*a* :: *c-type*) *f*))
:: '*a* :: *c-type ptr*)
(**is** *cparent p f* ≡ *?p'*)
⟨*proof*⟩

lemma *field-cparent* [*simp*]:
fixes *p* :: '*a* :: *c-type ptr*
shows (*Ptr* &(*cparent p f* :: '*b* :: *c-type ptr* →*f*)) = *p*
⟨*proof*⟩

lemma *clift-cparentE*:
fixes *v* :: '*a* :: *mem-type* **and** *p* :: '*b* :: *mem-type ptr*
assumes *lf*: *clift hp* (*cparent p fs* :: '*a ptr*) = *Some v*
and *fl*: *field-ti TYPE*('*a*) *fs* = *Some t*
and *eu*: *export-uinfo t* = *export-uinfo* (*typ-info-t TYPE*('*b*))
shows *clift hp p* = *Some* (*from-bytes* (*access-ti*$_0$ *t v*))
⟨*proof*⟩

lemma *heap-update-to-cparent*:
fixes *p* :: '*b* :: *packed-type ptr* **and** *fs* :: *char list list*
defines *cp* ≡ *cparent p fs* :: '*a* :: *packed-type ptr*
assumes *fl*: *field-ti TYPE*('*a* :: *packed-type*) *fs* = *Some t*
and *cg*: *c-guard cp*
and *eu*: *export-uinfo t* = *export-uinfo* (*typ-info-t TYPE*('*b*))
shows *heap-update p v hp* = *heap-update cp* (*update-ti t* (*to-bytes-p v*) (*h-val hp cp*)) *hp*
(**is** *?LHS* = *?RHS*)

⟨*proof*⟩

lemma *c-guard-cparent*:
⟦ *c-guard* ((*cparent p f*)::*'a*::*mem-type ptr*);
field-ti TYPE(*'a*) *f* = *Some t*;
export-uinfo t = *typ-uinfo-t TYPE*(*'b*) ⟧ ⟹
c-guard (*p*::*'b*::*mem-type ptr*)
⟨*proof*⟩

lemma *parent-update-child*:
fixes *p*::*'b*::*packed-type ptr*
shows
⟦ *c-guard* ((*cparent p f*)::*'a*::*packed-type ptr*); *field-ti TYPE*(*'a*) *f* = *Some t*;
export-uinfo t = *export-uinfo* (*typ-info-t TYPE*(*'b*))⟧
⟹ *hrs-mem-update* (*heap-update p v*) *hp* =
hrs-mem-update
(*heap-update* ((*cparent p f*)::*'a ptr*)
(*update-ti-t t* (*to-bytes-p v*) (*h-val* (*hrs-mem hp*) (*cparent p f*))))
hp
⟨*proof*⟩

15.3.5 *h-val*

lemma *h-val-field-clift*:
fixes *pa* :: *'a* :: *mem-type ptr*
assumes *cl*: *clift* (*h*, *d*) *pa* = *Some v*
and *fl*: *field-ti TYPE*(*'a*) *f* = *Some t*
and *eu*: *export-uinfo t* = *export-uinfo* (*typ-info-t TYPE*(*'b* :: *mem-type*))
shows *h-val h* (*Ptr* &(*pa*→*f*) :: *'b* :: *mem-type ptr*) = *from-bytes* (*access-ti*$_0$ *t v*)
⟨*proof*⟩

lemma *h-val-field-clift'*:
fixes *pa* :: *'a* :: *mem-type ptr*
assumes *cl*: *clift hp pa* = *Some v*
and *fl*: *field-ti TYPE*(*'a*) *f* = *Some t*
and *eu*: *export-uinfo t* = *typ-uinfo-t TYPE*(*'b* :: *mem-type*)
shows *h-val* (*hrs-mem hp*) (*Ptr* &(*pa*→*f*) :: *'b* :: *mem-type ptr*) = *from-bytes* (*access-ti*$_0$ *t v*)
⟨*proof*⟩

lemma *clift-subtype*:
⟦ *clift hp* ((*cparent p f*)::*'a*::*mem-type ptr*) = *Some v*;
field-ti TYPE(*'a*) *f* = *Some t*;
export-uinfo t = *export-uinfo* (*typ-info-t TYPE*(*'b*::*mem-type*)) ⟧ ⟹
clift hp (*p*::*'b ptr*) = *Some* (*from-bytes* (*access-ti*$_0$ *t v*))
⟨*proof*⟩

15.3.6 *h-t-valid*

lemma *h-t-valid-field*:
fixes $p :: 'a :: mem\text{-}type\ ptr$
assumes *htv*: $d \models_t p$
and *fti*: *field-ti TYPE*('a :: *mem-type*) f = *Some t*
and *eu*: *export-uinfo t* = *export-uinfo* (*typ-info-t TYPE*('b :: *mem-type*))
shows $d \models_t (Ptr\ \&(p\rightarrow f) :: 'b :: mem\text{-}type\ ptr)$
⟨*proof*⟩

lemma *h-t-valid-field'*:
fixes $p::'a::mem\text{-}type\ ptr$
shows
$[\![$ *field-ti TYPE*('a) f = *Some t*;
export-uinfo t = *typ-uinfo-t TYPE*('b);
$d,g \models_t p;\ g'\ ((Ptr\ \&(p\rightarrow f))::'b::mem\text{-}type\ ptr)\ ]\!] \Longrightarrow d,g' \models_t Ptr\ \&(p\rightarrow f)$
⟨*proof*⟩

lemma *h-t-valid-c-guard-field*:
fixes $p::'a::mem\text{-}type\ ptr$
shows
$[\![\ d \models_t p;$
field-ti TYPE('a) f = *Some t*;
export-uinfo t = *typ-uinfo-t TYPE*('b) $]\!] \Longrightarrow$
$d \models_t ((Ptr\ \&(p\rightarrow f))::'b::mem\text{-}type\ ptr)$
⟨*proof*⟩

lemma *h-t-valid-cparent*:
$[\![$ *field-ti TYPE*('a) f = *Some t*;
export-uinfo t = *typ-uinfo-t TYPE*('b);
$d,g \models_t ((cparent\ p\ f)::'a::mem\text{-}type\ ptr);\ g'\ (p::'b::mem\text{-}type\ ptr)\ ]\!] \Longrightarrow$
$d,g' \models_t p$
⟨*proof*⟩

lemma *h-t-valid-c-guard-cparent*:
fixes $p::'b::mem\text{-}type\ ptr$
shows
$[\![\ d \models_t ((cparent\ p\ f)::'a::mem\text{-}type\ ptr);$
field-ti TYPE('a) f = *Some t*;
export-uinfo t = *typ-uinfo-t TYPE*('b) $]\!] \Longrightarrow$
$d \models_t p$
⟨*proof*⟩

lemma *c-guard-array-c-guard*:
c-guard (*ptr-coerce p* :: ('b :: *c-type*, 'a :: *finite*) *array ptr*) ⟹ *c-guard* (*p* :: 'b *ptr*)
⟨*proof*⟩

lemma *c-guard-array-field*:
assumes *parent-cguard*: *c-guard* (*p* :: 'a :: *mem-type ptr*)

and *subfield*: *field-ti TYPE*('*a* :: *mem-type*) *f* = *Some t*
and *type-match*: *export-uinfo t* = *export-uinfo* (*typ-info-t TYPE*((*'b* :: *array-outer-max-size*, *'c* :: *array-max-count*) *array*))
shows *c-guard* (*Ptr* &(*p*→*f*) :: *'b ptr*)
⟨*proof*⟩

instantiation *ptr* :: (*type*) *enum*
begin

definition *enum-ptr* ≡ *map Ptr enum-class.enum*
definition *enum-all-ptr P* ≡ *enum-class.enum-all* (λ*v*. *P* (*Ptr v*))
definition *enum-ex-ptr P* ≡ *enum-class.enum-ex* (λ*v*. *P* (*Ptr v*))

instance
⟨*proof*⟩
end

15.3.7 Type Combinators and Padding

lemma *ti-typ-pad-combine-empty-ti*:
fixes *tp* :: *'b* :: *c-type itself*
shows *ti-typ-pad-combine tp lu upd algn fld* (*empty-typ-info algn' n*) =
TypDesc (*max algn'* (*max algn* (*align-td* (*typ-info-t TYPE*(*'b*))))))
(*TypAggregate* [*DTuple* (*adjust-ti* (*typ-info-t TYPE*(*'b*)) *lu upd*) *fld*
(|*field-access* = *xto-bytes* ∘ *lu*,
field-update = *upd* ∘ *xfrom-bytes*,
field-sz = *size-of TYPE*(*'b*)|)]) *n*
⟨*proof*⟩

lemma *ti-typ-combine-empty-ti*:
fixes *tp* :: *'b* :: *c-type itself*
shows *ti-typ-combine tp lu upd algn fld* (*empty-typ-info algn' n*) =
TypDesc (*max algn'* (*max algn* (*align-td* (*typ-info-t TYPE*(*'b*))))))
(*TypAggregate* [*DTuple* (*adjust-ti* (*typ-info-t TYPE*(*'b*)) *lu upd*) *fld*
(|*field-access* = *xto-bytes* ∘ *lu*,
field-update = *upd* ∘ *xfrom-bytes*,
field-sz = *size-of TYPE*(*'b*)|)]) *n*
⟨*proof*⟩

lemma *ti-typ-pad-combine-td*:
fixes *tp* :: *'b* :: *c-type itself*
shows *padup* (*max* (2 ^ *algn*) (*align-of TYPE*(*'b*))) (*size-td-struct st*) = *0* ⟹
ti-typ-pad-combine tp lu upd algn fld (*TypDesc algn' st n*) =
TypDesc (*max algn'* (*max algn* (*align-td* (*typ-info-t TYPE*(*'b*))))))
(*extend-ti-struct st* (*adjust-ti* (*typ-info-t TYPE*(*'b*)) *lu upd*) *fld*
(|*field-access* = *xto-bytes* ∘ *lu*,
field-update = *upd* ∘ *xfrom-bytes*,
field-sz = *size-of TYPE*(*'b*)|)) *n*
⟨*proof*⟩

lemma *ti-typ-combine-td*:
fixes *tp* :: *'b* :: *c-type itself*
shows *padup* (*align-of TYPE*('*b*)) (*size-td-struct st*) = *0* ⟹
ti-typ-combine tp lu upd algn fld (*TypDesc algn' st n*) =
TypDesc (*max algn'* (*max algn* (*align-td* (*typ-info-t TYPE*('*b*))))))
(*extend-ti-struct st* (*adjust-ti* (*typ-info-t TYPE*('*b*)) *lu upd*) *fld*
(|*field-access* = *xto-bytes* ∘ *lu*,
field-update = *upd* ∘ *xfrom-bytes*,
field-sz = *size-of TYPE*('*b*)|)) *n*
⟨*proof*⟩

lemma *update-ti-t-pad-combine*:
assumes *std*: *size-td td' mod 2* ^ (*max algn* (*align-td* (*typ-info-t TYPE*('*a* :: *c-type*)))) = *0*
shows *update-ti-t* (*ti-typ-pad-combine TYPE*('*a* :: *c-type*) *lu upd algn fld td'*) *bs v* =
update-ti-t (*ti-typ-combine TYPE*('*a* :: *c-type*) *lu upd algn fld td'*) *bs v*
⟨*proof*⟩

15.3.8 The orphanage: miscellaneous lemmas pulled up to (roughly) where they belong.

lemma *uinfo-array-tag-n-m-not-le-typ-name*:
typ-name (*typ-info-t TYPE*('*b*)) @ ''*-array-*'' @ *nat-to-bin-string m*
∉ *td-names* (*typ-info-t TYPE*('*a*))
⟹ ¬ *uinfo-array-tag-n-m TYPE*('*b* :: *c-type*) *n m* ≤ *typ-uinfo-t TYPE*('*a* :: *c-type*)
⟨*proof*⟩

lemma *tag-not-le-via-td-name*:
typ-name (*typ-info-t TYPE*('*a*)) ∉ *td-names* (*typ-info-t TYPE*('*b*))
⟹ *typ-name* (*typ-info-t TYPE*('*a*)) ≠ *pad-typ-name*
⟹ ¬ *typ-uinfo-t TYPE*('*a* :: *c-type*) ≤ *typ-uinfo-t TYPE* ('*b* :: *c-type*)
⟨*proof*⟩

lemmas *typ-heap-simps* =
— *c-guard*
c-guard-field
c-guard-h-t-valid
— *h-t-valid*
h-t-valid-field
h-t-valid-clift
— *h-val*
h-val-field-clift'

h-val-clift'
— *clift*
clift-field
clift-field-update
heap-update-field-hrs
heap-update-field'
clift-heap-update
clift-heap-update-same-td-name — Try this last (is expensive)

end

theory *LemmaBucket-C*
imports
More-Lib
WordSetup
TypHeapLib
ArrayAssertion
begin

declare *word-neq-0-conv* [*simp del*]

lemma *Ptr-not-null-pointer-not-zero*: $(Ptr\ p \neq NULL) = (p \neq 0)$
⟨*proof*⟩

lemma *hrs-mem-f*: $f\ (hrs\text{-}mem\ s) = hrs\text{-}mem\ (hrs\text{-}mem\text{-}update\ f\ s)$
⟨*proof*⟩

lemma *hrs-mem-heap-update*:
$heap\text{-}update\ p\ v\ (hrs\text{-}mem\ s) = hrs\text{-}mem\ (hrs\text{-}mem\text{-}update\ (heap\text{-}update\ p\ v)\ s)$
⟨*proof*⟩

lemma *surj-Ptr* [*simp*]:
surj Ptr
⟨*proof*⟩

lemma *inj-Ptr* [*simp*]:
inj Ptr
⟨*proof*⟩

lemma *bij-Ptr* :
bij Ptr
⟨*proof*⟩

lemma *exec-Guard*:
$(G \vdash \langle Guard\ Err\ S\ c,\ Normal\ s\rangle \Rightarrow s') = (if\ s \in S\ then\ G \vdash \langle c,\ Normal\ s\rangle \Rightarrow s'\ else\ s' = Fault\ Err)$

⟨*proof*⟩

lemma *byte-ptr-guarded:ptr-val* (*x*::*8 word ptr*) ≠ *0* ⟹ *c-guard x*
⟨*proof*⟩

lemma *intvl-aligned-bottom-eq*:
fixes *p* :: *'a*::*len word*
assumes *al1*: *is-aligned x n*
and *al2*: *is-aligned p bits*
and *nb*: ¬ *n* < *bits*
and *off*: *off* ≤ *2* ^ *bits off* ≠ *0*
shows (*x* ∈ {*p* ..+ *off*}) = (*x* = *p*)
⟨*proof*⟩

lemma *intvl-mem-weaken*: *x* ∈ {*p*..+*a* − *n*} ⟹ *x* ∈ {*p*..+*a*}
⟨*proof*⟩

lemma *upto-intvl-eq*:
fixes *x* :: *'a*::*len word*
assumes *al*: *is-aligned x n*
shows {*x*..+*2* ^ *n*} = {*x* .. *x* + *2* ^ *n* − *1*}
⟨*proof*⟩

lemma *upto-intvl-eq'*:
fixes *x* :: *'a* :: *len word*
shows ⟦ *x* ≤ *x* + (*of-nat b* − *1*); *b* ≠ *0*; *b* ≤ *2* ^ *len-of TYPE*(*'a*)⟧ ⟹ {*x*..+*b*}
= {*x* .. *x* + *of-nat b* − *1*}
⟨*proof*⟩

lemma *intvl-aligned-top*:
fixes *x* :: *'a*::*len word*
assumes *al1*: *is-aligned x n*
and *al2*: *is-aligned p bits*
and *nb*: *n* ≤ *bits*
and *offn*: *off* < *2* ^ *n*
and *wb*: *bits* < *len-of TYPE*(*'a*)
shows (*x* ∈ {*p* ..+ *2* ^ *bits* − *off*}) = (*x* ∈ {*p* ..+ *2* ^ *bits*})
⟨*proof*⟩

lemma *heap-update-list-update*:
fixes *v* :: *word8*
shows *x* ≠ *y* ⟹ *heap-update-list s xs* (*hp*(*y* := *v*)) *x* = *heap-update-list s xs hp x*
⟨*proof*⟩

lemma *heap-update-list-append2*:
length xs + length ys < 2 ^ word-bits ⟹
heap-update-list s (xs @ ys) hp
= heap-update-list s xs (heap-update-list (s + of-nat (length xs)) ys hp)
⟨*proof*⟩

lemma *heap-update-word8*:
heap-update p (v :: word8) hp = hp(ptr-val p := v)
⟨*proof*⟩

lemma *index-foldr-update2*:
⟦ n ≤ i; i < CARD('b::finite) ⟧ ⟹ index (foldr (λn arr. Arrays.update arr n m) [0..<n] (x :: ('a,'b) array)) i = index x i
⟨*proof*⟩

lemma *index-foldr-update*:
⟦ i < n; n ≤ CARD('b::finite) ⟧ ⟹ index (foldr (λn arr. Arrays.update arr n m) [0..<n] (x :: ('a,'b) array)) i = m
⟨*proof*⟩

lemma *intvl-disjoint1*:
fixes *a :: 'a :: len word*
assumes *abc*: *a + of-nat b ≤ c*
and *alb*: *a ≤ a + of-nat b*
and *cld*: *c ≤ c + of-nat d*
and *blt*: *b < 2 ^ len-of TYPE('a)*
and *dlt*: *d < 2 ^ len-of TYPE('a)*
shows *{a..+b} ∩ {c..+d} = {}*
⟨*proof*⟩

lemma *intvl-disjoint2*:
fixes *a :: 'a :: len word*
assumes *abc*: *a + of-nat b ≤ c*
and *alb*: *a ≤ a + of-nat b*
and *cld*: *c ≤ c + of-nat d*
and *blt*: *b < 2 ^ len-of TYPE('a)*
and *dlt*: *d < 2 ^ len-of TYPE('a)*
shows *{c..+d} ∩ {a..+b} = {}*
⟨*proof*⟩

lemma *typ-slice-t-self*:
td ∈ fst ` set (typ-slice-t td m)
⟨*proof*⟩

lemma *index-fold-update*:
⟦ distinct xs; set xs ⊆ {..< card (UNIV :: 'b set)}; n < card (UNIV :: 'b set) ⟧
⟹
index (foldr (λn (arr :: 'a['b :: finite]). Arrays.update arr n (f n (index arr n))))

xs v) *n*
= (*if n* ∈ *set xs then f n* (*index v n*) *else index v n*)
⟨*proof*⟩

lemma *hrs-mem-update-cong*:
⟦ ⋀*x*. *f x* = *f′ x* ⟧ ⟹ *hrs-mem-update f* = *hrs-mem-update f′*
⟨*proof*⟩

lemma *Guard-no-cong*:
⟦ *A*=*A′*; *c*=*c′* ⟧ ⟹ *Guard A P c* = *Guard A′ P c′*
⟨*proof*⟩

lemma *coerce-heap-update-to-heap-updates*:
assumes *n*: *n* = *chunk* ∗ *m* **and** *len*: *length xs* = *n*
shows *heap-update-list x xs*
= (λ*s*. *foldl* (λ*s n*. *heap-update-list* (*x* + (*of-nat n* ∗ *of-nat chunk*))
(*take chunk* (*drop* (*n* ∗ *chunk*) *xs*)) *s*)
s [*0* ..< *m*])
⟨*proof*⟩

lemma *update-ti-list-array′*:
⟦ *update-ti-list-t* (*map f* [*0* ..< *n*]) *xs v* = *y*;
∀ *n*. *size-td-tuple* (*f n*) = *v3*; *length xs* = *v3* ∗ *n*;
∀ *m xs v′*. *length xs* = *v3* ∧ *m* < *n* ⟶
update-ti-tuple-t (*f m*) *xs v′* = *Arrays.update v′ m* (*update-ti-t* (*g m*) *xs* (*index v′ m*)) ⟧
⟹ *y* = *foldr* (λ*n arr*. *Arrays.update arr n* (*update-ti-t* (*g n*) (*take v3* (*drop* (*v3* ∗ *n*) *xs*)) (*index arr n*))) [*0* ..< *n*] *v*
⟨*proof*⟩

lemma *update-ti-list-array*:
⟦ *update-ti-list-t* (*map f* [*0* ..< *n*]) *xs v* = (*y* :: *′a*[*′b* :: *finite*]);
∀ *n*. *size-td-tuple* (*f n*) = *v3*; *length xs* = *v3* ∗ *n*;
∀ *m xs v′*. *length xs* = *v3* ∧ *m* < *n* ⟶
update-ti-tuple-t (*f m*) *xs v′* = *Arrays.update v′ m* (*update-ti-t* (*g m*) *xs* (*index v′ m*));
n ≤ *card* (*UNIV* :: *′b set*) ⟧
⟹ ∀ *m* < *n*. *update-ti-t* (*g m*) (*take v3* (*drop* (*v3* ∗ *m*) *xs*)) (*index v m*) = *index y m*
⟨*proof*⟩

lemma *access-in-array*:
fixes *y* :: (*′a* :: *c-type*)[*′b* :: *finite*]
assumes *assms*: *h-val hp x* = *y*
n < *card* (*UNIV* :: *′b set*)
and *subst*: ∀ *xs v*. *length xs* = *size-of TYPE*(*′a*)

$\longrightarrow$ *update-ti-t* (*typ-info-t TYPE*('*a*)) *xs v* = *f xs*
shows *h-val hp*
(*Ptr* (*ptr-val x* + *of-nat* (*n* ∗ *size-of TYPE*('*a*)))) = *index y n*
⟨*proof*⟩

lemma *access-ti-list-array*:
⟦ ∀ *n*. *size-td-tuple* (*f n*) = *v3*; *length xs* = *v3* ∗ *n*;
∀ *m*. *m* < *n* ∧ *v3* ≤ *length* (*drop* (*v3* ∗ *m*) *xs*)
$\longrightarrow$ *access-ti-tuple* (*f m*) (*FCP g*) (*take v3* (*drop* (*v3* ∗ *m*) *xs*)) = (*h m*)
⟧ ⟹
access-ti-list (*map f* [*0* ..< *n*]) (*FCP g*) *xs*
= *foldl* (@) [] (*map h* [*0* ..< *n*])
⟨*proof*⟩

lemma *take-drop-foldl-concat*:
⟦ ⋀*y*. *y* < *m* ⟹ *length* (*f y*) = *n*; *x* < *m* ⟧
⟹ *take n* (*drop* (*x* ∗ *n*) (*foldl* (@) [] (*map f* [*0* ..< *m*]))) = *f x*
⟨*proof*⟩

lemma *heap-update-Array*:
heap-update (*p* ::('*a*::*packed-type*['*b*::*finite*]) *ptr*) *arr*
= (λ*s*. *foldl* (λ*s n*. *heap-update* (*array-ptr-index p False n*)
(*Arrays.index arr n*) *s*) *s* [*0* ..< *card* (*UNIV* :: '*b set*)])
⟨*proof*⟩

lemma *from-bytes-Array-element*:
fixes *p* :: ('*a*::*mem-type*['*b*::*finite*]) *ptr*
assumes *less*: *of-nat n* < *card* (*UNIV* :: '*b set*)
assumes *len*: *length bs* = *size-of TYPE*('*a*) ∗ *CARD*('*b*)
shows
index (*from-bytes bs* :: '*a*['*b*]) *n*
= *from-bytes* (*take* (*size-of TYPE*('*a*)) (*drop* (*n* ∗ *size-of TYPE*('*a*)) *bs*))
⟨*proof*⟩

lemma *heap-access-Array-element'*:
fixes *p* :: ('*a*::*mem-type*['*b*::*finite*]) *ptr*
assumes *less*: *of-nat n* < *card* (*UNIV* :: '*b set*)
shows
index (*h-val hp p*) *n*
= *h-val hp* (*array-ptr-index p False n*)
⟨*proof*⟩

lemmas *heap-access-Array-element*
= *heap-access-Array-element'*[*simplified array-ptr-index-simps*]

lemma *heap-update-id*:
h-val hp ptr = (*v* :: '*a* :: *packed-type*)

$\Longrightarrow$ *heap-update ptr v hp = hp*
⟨*proof*⟩

lemma *heap-update-Array-update*:
assumes *n*: *n < CARD('b :: finite)*
assumes *size*: *CARD('b) * size-of TYPE('a :: packed-type) < 2 ^ addr-bitsize*
shows *heap-update p (Arrays.update (arr :: 'a['b]) n v) hp*
= *heap-update (array-ptr-index p False n) v (heap-update p arr hp)*
⟨*proof*⟩

lemma *heap-update-id-Array*:
fixes *arr :: ('a :: packed-type)['b :: finite]*
shows *arr = h-val hp p*
$\Longrightarrow$ *heap-update p arr hp = hp*
⟨*proof*⟩

lemma *heap-update-Array-element''*:
fixes *p' :: (('a :: packed-type)['b::finite]) ptr*
fixes *p :: ('a :: packed-type) ptr*
fixes *hp w*
assumes *p*: *p = array-ptr-index p' False n*
assumes *n*: *n < CARD('b)*
assumes *size*: *CARD('b) * size-of TYPE('a) < 2 ^ addr-bitsize*
shows *heap-update p' (Arrays.update (h-val hp p') n w) hp*
= *heap-update p w hp*
⟨*proof*⟩

lemmas *heap-update-Array-element'*
= *heap-update-Array-element''[simplified array-ptr-index-simps]*

lemma *array-count-size*:
*CARD('b :: array-max-count) * size-of TYPE('a :: array-outer-max-size) < 2 ^ addr-bitsize*
⟨*proof*⟩

lemmas *heap-update-Array-element*
= *heap-update-Array-element'[OF refl - array-count-size]*

lemma *typ-slice-list-cut*:
⟦ *(∀ x ∈ set xs. size-td (dt-fst x) = m); m ≠ 0; n < (length xs * m)* ⟧
$\Longrightarrow$ *typ-slice-list xs n =*
typ-slice-tuple (xs ! (n div m)) (n mod m)
⟨*proof*⟩

lemma *typ-slice-t-array*:
⟦ *n < CARD('b); y < size-of TYPE('a)* ⟧
$\Longrightarrow$ *typ-slice-t (export-uinfo (typ-info-t TYPE('a))) y ≤*
typ-slice-t (export-uinfo (array-tag TYPE('a['b :: finite])))

$(y + \mathit{size\text{-}of}\ \mathit{TYPE}('a :: \mathit{mem\text{-}type}) * n)$
⟨*proof*⟩

lemma *h-t-valid-Array-element'*:
⟦ $\mathit{htd} \models_t (p :: (('a :: \mathit{mem\text{-}type})['b :: \mathit{finite}])\ \mathit{ptr});\ \mathit{coerce} \vee n < \mathit{CARD}('b)$ ⟧
$\Longrightarrow \mathit{htd} \models_t \mathit{array\text{-}ptr\text{-}index}\ p\ \mathit{coerce}\ n$ **for** *coerce*
⟨*proof*⟩

lemma *h-t-valid-Array-element*:
⟦ $\mathit{htd} \models_t (p :: (('a :: \mathit{mem\text{-}type})['b :: \mathit{finite}])\ \mathit{ptr});\ 0 \leq n;\ n < \mathit{int}\ \mathit{CARD}('b)$ ⟧
$\Longrightarrow \mathit{htd} \models_t ((\mathit{ptr\text{-}coerce}\ p :: 'a\ \mathit{ptr}) +_p n)$
⟨*proof*⟩

lemma *ptr-safe-Array-element*:
⟦ $\mathit{ptr\text{-}safe}\ (p :: (('a :: \mathit{mem\text{-}type})['b :: \mathit{finite}])\ \mathit{ptr})\ \mathit{htd};\ \mathit{coerce} \vee n < \mathit{CARD}('b)$ ⟧
$\Longrightarrow \mathit{ptr\text{-}safe}\ (\mathit{array\text{-}ptr\text{-}index}\ p\ \mathit{coerce}\ n)\ \mathit{htd}$ **for** *coerce*
⟨*proof*⟩

lemma *from-bytes-eq*:
$\mathit{from\text{-}bytes}\ [x] = x$
⟨*proof*⟩

lemma *bytes-disjoint*:$(x::('a::\mathit{c\text{-}type})\ \mathit{ptr}) \neq y \Longrightarrow \{\mathit{ptr\text{-}val}\ x + a\ ..+\ 1\} \cap \{\mathit{ptr\text{-}val}\ y + a\ ..+\ 1\} = \{\}$
⟨*proof*⟩

lemma *byte-ptrs-disjoint*:$(x::('a::\mathit{c\text{-}type})\ \mathit{ptr}) \neq y \Longrightarrow \forall\, i < \mathit{of\text{-}nat}\ (\mathit{size\text{-}of}\ \mathit{TYPE}('a)).\ \mathit{ptr\text{-}val}\ x + i \neq \mathit{ptr\text{-}val}\ y + i$
⟨*proof*⟩

lemma *le-step*:⟦$(x::('a::\mathit{len})\ \mathit{word}) < y + 1;\ x \neq y$⟧ $\Longrightarrow x < y$
⟨*proof*⟩

lemma *ptr-add-disjoint*:
⟦ $\mathit{ptr\text{-}val}\ y \notin \{\mathit{ptr\text{-}val}\ x\ ..+\ \mathit{size\text{-}of}\ \mathit{TYPE}('a)\};$
$\mathit{ptr\text{-}val}\ (x::('a::\mathit{c\text{-}type})\ \mathit{ptr}) < \mathit{ptr\text{-}val}\ (y::('b::\mathit{c\text{-}type})\ \mathit{ptr});$
$a < \mathit{of\text{-}nat}\ (\mathit{size\text{-}of}\ \mathit{TYPE}('a))$ ⟧ $\Longrightarrow$
$\mathit{ptr\text{-}val}\ x + a < \mathit{ptr\text{-}val}\ y$
⟨*proof*⟩

lemma *ptr-add-disjoint2*:
⟦ $\mathit{ptr\text{-}val}\ x \notin \{\mathit{ptr\text{-}val}\ y\ ..+\ \mathit{size\text{-}of}\ \mathit{TYPE}('a)\};$
$\mathit{ptr\text{-}val}\ (y::('b::\mathit{c\text{-}type})\ \mathit{ptr}) < \mathit{ptr\text{-}val}\ (x::('a::\mathit{c\text{-}type})\ \mathit{ptr});$
$a < \mathit{of\text{-}nat}\ (\mathit{size\text{-}of}\ \mathit{TYPE}('a))$ ⟧ $\Longrightarrow$
$\mathit{ptr\text{-}val}\ y + a < \mathit{ptr\text{-}val}\ x$
⟨*proof*⟩

lemma *ptr-aligned-is-aligned*:⟦$\mathit{ptr\text{-}aligned}\ (x::('a::\mathit{c\text{-}type})\ \mathit{ptr});\ \mathit{align\text{-}of}\ \mathit{TYPE}('a) = 2\ \hat{}\ n$⟧ $\Longrightarrow \mathit{is\text{-}aligned}\ (\mathit{ptr\text{-}val}\ x)\ n$

⟨*proof*⟩

lemma *intvl-no-overflow*:
assumes *no-overflow*: *unat a + b < 2 ^ len-of TYPE('a::len)*
shows *(x ∈ {(a :: 'a word) ..+ b}) = (a ≤ x ∧ x < (a + of-nat b))*
⟨*proof*⟩

lemma *FCP-arg-cong*:*f = g ⟹ FCP f = FCP g*
⟨*proof*⟩

lemma *h-val-id*:
h-val (hrs-mem (hrs-mem-update (heap-update x y) s)) x = (y::'a::mem-type)
⟨*proof*⟩

lemma *h-val-id-padding*:
length bs = size-of TYPE('a) ⟹ h-val (hrs-mem (hrs-mem-update (heap-update-padding x y bs) s)) x = (y::'a::mem-type)
⟨*proof*⟩

lemma *heap-update-id2*:
hrs-mem-update (heap-update p ((h-val (hrs-mem s) p)::'a::packed-type)) s = s
⟨*proof*⟩

lemma *intvlI-unat*:*unat b < unat c ⟹ a + b ∈ {a ..+ unat c}*
⟨*proof*⟩

lemma *neq-imp-bytes-disjoint*:
⟦ c-guard (x::'a::c-type ptr); c-guard y; unat j < align-of TYPE('a);
unat i < align-of TYPE('a); x ≠ y; 2 ^ n = align-of TYPE('a); n < 32⟧
⟹
ptr-val x + j ≠ ptr-val y + i
⟨*proof*⟩

lemma *heap-update-list-base'*:*heap-update-list p [] = id*
⟨*proof*⟩

lemma *hrs-mem-update-id3*: *hrs-mem-update id = id*
⟨*proof*⟩

lemma *h-t-valid-ptr-retyp-inside-eq*:
fixes *p :: 'a :: mem-type ptr* **and** *p' :: 'a :: mem-type ptr*
assumes *inside*: *ptr-val p' ∈ {ptr-val p ..+ size-of TYPE('a)}*
and *ht*: *ptr-retyp p td, g ⊨ₜ p'*
shows *p = p'*
⟨*proof*⟩

lemma *typ-slice-t-self-nth*:
$\exists\, n < \mathit{length}\ (\mathit{typ\text{-}slice\text{-}t}\ \mathit{td}\ m).\ \exists\, b.\ \mathit{typ\text{-}slice\text{-}t}\ \mathit{td}\ m\ !\ n = (\mathit{td},\ b)$
⟨*proof*⟩

lemma *ptr-retyp-other-cleared-region*:
fixes $p :: 'a :: \mathit{mem\text{-}type}\ \mathit{ptr}$ **and** $p' :: 'b :: \mathit{mem\text{-}type}\ \mathit{ptr}$
assumes *ht*: $\mathit{ptr\text{-}retyp}\ p\ \mathit{td},\ g \models_t p'$
and *tdisj*: $\mathit{typ\text{-}uinfo\text{-}t}\ \mathit{TYPE}('a) \perp_t \mathit{typ\text{-}uinfo\text{-}t}\ \mathit{TYPE}('b :: \mathit{mem\text{-}type})$
and *clear*: $\forall\, x \in \{\mathit{ptr\text{-}val}\ p\ ..+\ \mathit{size\text{-}of}\ \mathit{TYPE}('a)\}.\ \forall\, n\ b.\ \mathit{snd}\ (\mathit{td}\ x)\ n \neq \mathit{Some}\ (\mathit{typ\text{-}uinfo\text{-}t}\ \mathit{TYPE}('b),\ b)$
shows $\{\mathit{ptr\text{-}val}\ p'..+\ \mathit{size\text{-}of}\ \mathit{TYPE}('b)\} \cap \{\mathit{ptr\text{-}val}\ p\ ..+\ \mathit{size\text{-}of}\ \mathit{TYPE}('a)\} = \{\}$
⟨*proof*⟩

end

theory *Cong-Tactic*
imports
Main
HOL−Eisbach.Eisbach-Tools

begin

Simple congruence prover
Replaces a goal of the shape:
$f\ x\ (g\ y)\ (\lambda x.\ x + 1) = f\ x'\ (i\ y')\ (\lambda z.\ z + \mathit{Suc}\ 0)$
by
$x = x'\ g\ y = i\ y'\ 1 = \mathit{Suc}\ 0$
The tactic essentially applies $[\![?f = ?g;\ ?x = ?y]\!] \Longrightarrow ?f\ ?x = ?g\ ?y$, but using first-order matching.

⟨*ML*⟩

method *cong-step* = (*app-cong* | *abs-cong* | *rule refl* | *rule eq-reflection*)
method *cong* = (*cong-step* ; *cong?*)

Preserve context information from [*cong*] rules

⟨*ML*⟩

method *cong-context-step* = (*const-cong* | *cong-step* | *rule simp-impliesI*)
method *cong-context* = (*cong-context-step* ; *cong-context?*)

end

Part III

AutoCorres

Chapter 16

Spec-Monad

theory *Spec-Monad*
imports
Basic-Runs-To-VCG
HOL−Library.Complete-Partial-Order2
HOL−Library.Monad-Syntax
AutoCorres-Utils
begin

16.1 *rel-map* and *rel-project*

definition *rel-map* :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$ **where**
rel-map f r x = $(x = f\ r)$

lemma *rel-map-direct*[*simp*]: *rel-map f a* $(f\ a)$
⟨*proof*⟩

abbreviation *rel-project* ≡ *rel-map*

lemmas *rel-project-def* = *rel-map-def*

lemma *rel-project-id*: *rel-project id* = $(=)$
rel-project $(\lambda v.\ v)$ = $(=)$
⟨*proof*⟩

lemma *rel-project-unit*: *rel-project* $(\lambda\text{-}.\ ())$ *x y* = *True*
⟨*proof*⟩

lemma *rel-projectI*: *y* = *prj x* ⟹ *rel-project prj x y*
⟨*proof*⟩

lemma *rel-project-conv*: *rel-project prj x y* = $(y = prj\ x)$
⟨*proof*⟩

16.2 Misc Theorems

declare *case-unit-Unity* [*simp*] — without this rule simplifier seems loops in unexpected ways

lemma *abs-const-unit*: $(\lambda(v::unit).\ f) = (\lambda().\ f)$
⟨*proof*⟩

lemma *SUP-mono''*: $(\bigwedge x.\ x{\in}A \Longrightarrow f\ x \leq g\ x) \Longrightarrow (\bigsqcup x{\in}A.\ f\ x) \leq (\bigsqcup x{\in}A.\ g\ x ::$ *-::complete-lattice*)
⟨*proof*⟩

lemma *wf-nat-bound*: *wf* $\{(a,\ b).\ b < a \wedge b \leq (n::nat)\}$
⟨*proof*⟩

lemma (**in** *complete-lattice*) *admissible-le*:
ccpo.admissible Inf $(\leq)$ $(\lambda x.\ (y \leq x))$
⟨*proof*⟩

lemma *mono-const*: *mono* $(\lambda x.\ c)$
⟨*proof*⟩

lemma *mono-lam*: $(\bigwedge a.\ mono\ (\lambda x.\ F\ x\ a)) \Longrightarrow mono\ F$
⟨*proof*⟩

lemma *mono-app*: *mono* $(\lambda x.\ x\ a)$
⟨*proof*⟩

lemma *all-cong-map*:
assumes *f-f'*: $\bigwedge y.\ f\ (f'\ y) = y$ **and** *P-Q*: $\bigwedge x.\ P\ x \longleftrightarrow Q\ (f\ x)$
shows $(\forall x.\ P\ x) \longleftrightarrow (\forall y.\ Q\ y)$
⟨*proof*⟩

lemma *rel-set-refl*: $(\bigwedge x.\ x \in A \Longrightarrow R\ x\ x) \Longrightarrow rel\text{-}set\ R\ A\ A$
⟨*proof*⟩

lemma *rel-set-converse-iff*: *rel-set R X Y* $\longleftrightarrow$ *rel-set* R^{-1-1} *Y X*
⟨*proof*⟩

lemma *rel-set-weaken*:
$(\bigwedge x\ y.\ x \in A \Longrightarrow y \in B \Longrightarrow P\ x\ y \Longrightarrow Q\ x\ y) \Longrightarrow rel\text{-}set\ P\ A\ B \Longrightarrow rel\text{-}set\ Q\ A\ B$
⟨*proof*⟩

lemma *sim-set-refl*: $(\bigwedge x.\ x \in X \Longrightarrow R\ x\ x) \Longrightarrow sim\text{-}set\ R\ X\ X$
⟨*proof*⟩

16.3 Galois Connections

lemma *mono-of-Sup-cont*:
fixes f :: *'a*::*complete-lattice* ⇒ *'b*::*complete-lattice*
assumes *cont*: $\bigwedge X.\ f\ (Sup\ X) = (SUP\ x{\in}X.\ f\ x)$
assumes *xy*: $x \leq y$
shows $f\ x \leq f\ y$
⟨*proof*⟩

lemma *gc-of-Sup-cont*:
fixes f :: *'a*::*complete-lattice* ⇒ *'b*::*complete-lattice*
assumes *cont*: $\bigwedge X.\ f\ (Sup\ X) = (SUP\ x{\in}X.\ f\ x)$
shows $f\ x \leq y \longleftrightarrow x \leq Sup\ \{x.\ f\ x \leq y\}$
⟨*proof*⟩

lemma *mono-of-Inf-cont*:
fixes f :: *'a*::*complete-lattice* ⇒ *'b*::*complete-lattice*
assumes *cont*: $\bigwedge X.\ f\ (Inf\ X) = (INF\ x{\in}X.\ f\ x)$
assumes *xy*: $x \leq y$
shows $f\ x \leq f\ y$
⟨*proof*⟩

lemma *gc-of-Inf-cont*:
fixes f :: *'a*::*complete-lattice* ⇒ *'b*::*complete-lattice*
assumes *cont*: $\bigwedge X.\ f\ (Inf\ X) = (INF\ x{\in}X.\ f\ x)$
shows $Inf\ \{y.\ x \leq f\ y\} \leq y \longleftrightarrow x \leq f\ y$
⟨*proof*⟩

lemma *gfp-fusion*:
assumes *f-g*: $\bigwedge x\ y.\ g\ x \leq y \longleftrightarrow x \leq f\ y$
assumes *a*: *mono a*
assumes *b*: *mono b*
assumes ∗: $\bigwedge x.\ f\ (a\ x) = b\ (f\ x)$
shows $f\ (gfp\ a) = gfp\ b$
⟨*proof*⟩

lemma *lfp-fusion*:
assumes *f-g*: $\bigwedge x\ y.\ g\ x \leq y \longleftrightarrow x \leq f\ y$
assumes *a*: *mono a*
assumes *b*: *mono b*
assumes ∗: $\bigwedge x.\ g\ (a\ x) = b\ (g\ x)$
shows $g\ (lfp\ a) = lfp\ b$
⟨*proof*⟩

16.4 *post-state* type

datatype *'r post-state* = *Failure* | *Success 'r set*

Failure is supposed to model things like undefined behaviour in C. We

usually have to show the absence of *Failure* for all possible executions of the program. Moreover, it is used to model the 'result' of a non terminating computation.

lemma *split-post-state*:
  $x = (\mathit{case}\ x\ \mathit{of}\ \mathit{Success}\ X \Rightarrow \mathit{Success}\ X \mid \mathit{Failure} \Rightarrow \mathit{Failure})$
  for x::'a *post-state*
  ⟨*proof*⟩

instantiation *post-state* :: (*type*) *order*
begin

inductive *less-eq-post-state* :: 'a *post-state* ⇒ 'a *post-state* ⇒ *bool* **where**
  Failure-le[*simp*, *intro*]: *less-eq-post-state p Failure*
| *Success-le-Success*[*intro*]: $r \subseteq q \Longrightarrow \mathit{less\text{-}eq\text{-}post\text{-}state}\ (\mathit{Success}\ r)\ (\mathit{Success}\ q)$

definition *less-post-state* :: 'a *post-state* ⇒ 'a *post-state* ⇒ *bool* **where**
  $\mathit{less\text{-}post\text{-}state}\ p\ q \longleftrightarrow p \leq q \wedge \neg\ q \leq p$

instance
⟨*proof*⟩
end

lemma *Success-le-Success-iff*[*simp*]: $\mathit{Success}\ r \leq \mathit{Success}\ q \longleftrightarrow r \subseteq q$
  ⟨*proof*⟩

lemma *Failure-le-iff*[*simp*]: $\mathit{Failure} \leq q \longleftrightarrow q = \mathit{Failure}$
  ⟨*proof*⟩

instantiation *post-state* :: (*type*) *complete-lattice*
begin

definition *top-post-state* :: 'a *post-state* **where**
  top-post-state = *Failure*

definition *bot-post-state* :: 'a *post-state* **where**
  bot-post-state = *Success* {}

definition *inf-post-state* :: 'a *post-state* ⇒ 'a *post-state* ⇒ 'a *post-state* **where**
  inf-post-state =
    (λ*Failure* ⇒ *id* | *Success res1* ⇒
      (λ*Failure* ⇒ *Success res1* | *Success res2* ⇒ *Success* (*res1* ∩ *res2*)))

definition *sup-post-state* :: 'a *post-state* ⇒ 'a *post-state* ⇒ 'a *post-state* **where**
  sup-post-state =
    (λ*Failure* ⇒ (λ-. *Failure*) | *Success res1* ⇒
      (λ*Failure* ⇒ *Failure* | *Success res2* ⇒ *Success* (*res1* ∪ *res2*)))

definition *Inf-post-state* :: 'a *post-state set* ⇒ 'a *post-state* **where**
  Inf-post-state s = (*if Success* −‘ *s* = {} *then Failure else Success* (⋂ (*Success* −‘

$s)))$

definition *Sup-post-state* :: *'a post-state set* ⇒ *'a post-state* **where**
Sup-post-state s = (*if Failure* ∈ *s then Failure else Success* (⋃ (*Success* −‘ *s*)))

instance
⟨*proof*⟩
end

primrec *holds-post-state* :: (*'a* ⇒ *bool*) ⇒ *'a post-state* ⇒ *bool* **where**
holds-post-state P Failure ⟷ *False*
| *holds-post-state P* (*Success X*) ⟷ (∀ *x*∈*X*. *P x*)

primrec *holds-partial-post-state* :: (*'a* ⇒ *bool*) ⇒ *'a post-state* ⇒ *bool* **where**
holds-partial-post-state P Failure ⟷ *True*
| *holds-partial-post-state P* (*Success X*) ⟷ (∀ *x*∈*X*. *P x*)

inductive
sim-post-state :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a post-state* ⇒ *'b post-state* ⇒ *bool*
for *R*
where
[*simp*]: ⋀*p*. *sim-post-state R p Failure*
| ⋀*A B*. *sim-set R A B* ⟹ *sim-post-state R* (*Success A*) (*Success B*)

inductive
rel-post-state :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a post-state* ⇒ *'b post-state* ⇒ *bool*
for *R*
where
[*simp*]: *rel-post-state R Failure Failure*
| ⋀*A B*. *rel-set R A B* ⟹ *rel-post-state R* (*Success A*) (*Success B*)

primrec *lift-post-state* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'b post-state* ⇒ *'a post-state* **where**
lift-post-state R Failure = *Failure*
| *lift-post-state R* (*Success Y*) = *Success* {*x*. ∃ *y*∈*Y*. *R x y*}

primrec *unlift-post-state* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a post-state* ⇒ *'b post-state* **where**
unlift-post-state R Failure = *Failure*
| *unlift-post-state R* (*Success X*) = *Success* {*y*. ∀ *x*. *R x y* ⟶ *x* ∈ *X*}

primrec *map-post-state* :: (*'a* ⇒ *'b*) ⇒ *'a post-state* ⇒ *'b post-state* **where**
map-post-state f Failure = *Failure*
| *map-post-state f* (*Success r*) = *Success* (*f* ‘ *r*)

primrec *vmap-post-state* :: (*'a* ⇒ *'b*) ⇒ *'b post-state* ⇒ *'a post-state* **where**
vmap-post-state f Failure = *Failure*
| *vmap-post-state f* (*Success r*) = *Success* (*f* −‘ *r*)

definition *pure-post-state* :: *'a* ⇒ *'a post-state* **where**
pure-post-state v = *Success* {*v*}

primrec *bind-post-state* :: *'a post-state* ⇒ (*'a* ⇒ *'b post-state*) ⇒ *'b post-state*
where
  bind-post-state Failure p = *Failure*
| *bind-post-state* (*Success res*) *p* = ⨆ (*p* ‘ *res*)

16.4.1 Order Properties

lemma *top-ne-bot*[*simp*]: (⊥:: - *post-state*) ≠ ⊤
  and *bot-ne-top*[*simp*]: (⊤:: - *post-state*) ≠ ⊥
  ⟨*proof*⟩

lemma *Success-ne-top*[*simp*]:
  Success X ≠ ⊤ ⊤ ≠ *Success X*
  ⟨*proof*⟩

lemma *Success-eq-bot-iff*[*simp*]:
  Success X = ⊥ ⟷ *X* = {} ⊥ = *Success X* ⟷ *X* = {}
  ⟨*proof*⟩

lemma *Sup-Success*: (⨆ *x*∈*X*. *Success* (*f x*)) = *Success* (⋃ *x*∈*X*. *f x*)
  ⟨*proof*⟩

lemma *Sup-Success-pair*: (⨆ (*x*, *y*)∈*X*. *Success* (*f x y*)) = *Success* (⋃ (*x*, *y*)∈*X*. *f x y*)
  ⟨*proof*⟩

16.4.2 *holds-post-state*

lemma *holds-post-state-iff*:
  holds-post-state P p ⟷ (∃ *X*. *p* = *Success X* ∧ (∀ *x*∈*X*. *P x*))
  ⟨*proof*⟩

lemma *holds-post-state-weaken*:
  holds-post-state P p ⟹ (⋀*x*. *P x* ⟹ *Q x*) ⟹ *holds-post-state Q p*
  ⟨*proof*⟩

lemma *holds-post-state-combine*:
  holds-post-state P p ⟹ *holds-post-state Q p* ⟹
    (⋀*x*. *P x* ⟹ *Q x* ⟹ *R x*) ⟹ *holds-post-state R p*
  ⟨*proof*⟩

lemma *holds-post-state-Ball*:
  Y ≠ {} ∨ *p* ≠ *Failure* ⟹
    holds-post-state (λ*x*. ∀ *y*∈*Y*. *P x y*) *p* ⟷ (∀ *y*∈*Y*. *holds-post-state* (λ*x*. *P x y*) *p*)
  ⟨*proof*⟩

lemma *holds-post-state-All*:
  holds-post-state (λ*x*. ∀ *y*. *P x y*) *p* ⟷ (∀ *y*. *holds-post-state* (λ*x*. *P x y*) *p*)

⟨*proof*⟩

lemma *holds-post-state-BexI*:
$y \in Y \Longrightarrow$ *holds-post-state* $(\lambda x.\ P\ x\ y)\ p \Longrightarrow$ *holds-post-state* $(\lambda x.\ \exists\, y{\in}Y.\ P\ x\ y)\ p$
⟨*proof*⟩

lemma *holds-post-state-conj*:
holds-post-state $(\lambda x.\ P\ x \wedge Q\ x)\ p \longleftrightarrow$ *holds-post-state* $P\ p \wedge$ *holds-post-state* $Q\ p$
⟨*proof*⟩

lemma *sim-post-state-iff*:
sim-post-state $R\ p\ q \longleftrightarrow$
$(\forall\, P.$ *holds-post-state* $(\lambda y.\ \forall\, x.\ R\ x\ y \longrightarrow P\ x)\ q \longrightarrow$ *holds-post-state* $P\ p)$
⟨*proof*⟩

lemma *post-state-le-iff*:
$p \leq q \longleftrightarrow (\forall\, P.$ *holds-post-state* $P\ q \longrightarrow$ *holds-post-state* $P\ p)$
⟨*proof*⟩

lemma *post-state-eq-iff*:
$p = q \longleftrightarrow (\forall\, P.$ *holds-post-state* $P\ p \longleftrightarrow$ *holds-post-state* $P\ q)$
⟨*proof*⟩

lemma *holds-top-post-state*[*simp*]: $\neg$ *holds-post-state* $P\ \top$
⟨*proof*⟩

lemma *holds-bot-post-state*[*simp*]: *holds-post-state* $P\ \bot$
⟨*proof*⟩

lemma *holds-Sup-post-state*[*simp*]: *holds-post-state* P $(Sup\ F) \longleftrightarrow (\forall\, f{\in}F.$ *holds-post-state* $P\ f)$
⟨*proof*⟩

lemma *holds-post-state-gfp*:
holds-post-state P $(gfp\ f) \longleftrightarrow (\forall\, p.\ p \leq f\ p \longrightarrow$ *holds-post-state* $P\ p)$
⟨*proof*⟩

lemma *holds-post-state-gfp-apply*:
holds-post-state P $(gfp\ f\ x) \longleftrightarrow (\forall\, p.\ p \leq f\ p \longrightarrow$ *holds-post-state* P $(p\ x))$
⟨*proof*⟩

lemma *holds-lift-post-state*[*simp*]:
holds-post-state P $(lift\text{-}post\text{-}state\ R\ x) \longleftrightarrow$ *holds-post-state* $(\lambda y.\ \forall\, x.\ R\ x\ y \longrightarrow P\ x)\ x$
⟨*proof*⟩

lemma *holds-map-post-state*[*simp*]:

holds-post-state P (map-post-state f x) ⟷ holds-post-state (λx. P (f x)) x
⟨*proof*⟩

lemma *holds-vmap-post-state*[*simp*]:
holds-post-state P (vmap-post-state f x) ⟷ holds-post-state (λx. ∀ y. f y = x ⟶ P y) x
⟨*proof*⟩

lemma *holds-pure-post-state*[*simp*]: *holds-post-state P (pure-post-state x) ⟷ P x*
⟨*proof*⟩

lemma *holds-bind-post-state*[*simp*]:
holds-post-state P (bind-post-state f g) ⟷ holds-post-state (λx. holds-post-state P (g x)) f
⟨*proof*⟩

lemma *holds-post-state-False*: *holds-post-state (λx. False) f ⟷ f = ⊥*
⟨*proof*⟩

16.4.3 *holds-post-state-partial*

lemma *holds-partial-post-state-of-holds*:
holds-post-state P p ⟹ holds-partial-post-state P p
⟨*proof*⟩

lemma *holds-partial-post-state-iff*:
holds-partial-post-state P p ⟷ (∀ X. p = Success X ⟶ (∀ x∈X. P x))
⟨*proof*⟩

lemma *holds-partial-post-state-True*[*simp*]: *holds-partial-post-state (λx. True) p*
⟨*proof*⟩

lemma *holds-partial-post-state-weaken*:
holds-partial-post-state P p ⟹ (⋀x. P x ⟹ Q x) ⟹ holds-partial-post-state Q p
⟨*proof*⟩

lemma *holds-partial-post-state-Ball*:
holds-partial-post-state (λx. ∀ y∈Y. P x y) p ⟷ (∀ y∈Y. holds-partial-post-state (λx. P x y) p)
⟨*proof*⟩

lemma *holds-partial-post-state-All*:
holds-partial-post-state (λx. ∀ y. P x y) p ⟷ (∀ y. holds-partial-post-state (λx. P x y) p)
⟨*proof*⟩

lemma *holds-partial-post-state-conj*:
holds-partial-post-state (λx. P x ∧ Q x) p ⟷

holds-partial-post-state P p ∧ holds-partial-post-state Q p
⟨*proof*⟩

lemma *holds-partial-top-post-state*[*simp*]: *holds-partial-post-state P ⊤*
⟨*proof*⟩

lemma *holds-partial-bot-post-state*[*simp*]: *holds-partial-post-state P ⊥*
⟨*proof*⟩

lemma *holds-partial-pure-post-state*[*simp*]: *holds-partial-post-state P* (*pure-post-state x*) ⟷ *P x*
⟨*proof*⟩

lemma *holds-partial-Sup-post-stateI*:
(⋀*x*. *x* ∈ *X* ⟹ *holds-partial-post-state P x*) ⟹ *holds-partial-post-state P* (*Sup X*)
⟨*proof*⟩

lemma *holds-partial-bind-post-state*:
holds-partial-post-state (λ*x*. *holds-partial-post-state P* (*g x*)) *f* ⟹
holds-partial-post-state P (*bind-post-state f g*)
⟨*proof*⟩

lemma *holds-partial-map-post-state*[*simp*]:
holds-partial-post-state P (*map-post-state f x*) ⟷ *holds-partial-post-state* (λ*x*. *P* (*f x*)) *x*
⟨*proof*⟩

lemma *holds-partial-vmap-post-state*[*simp*]:
holds-partial-post-state P (*vmap-post-state f x*) ⟷
holds-partial-post-state (λ*x*. ∀ *y*. *f y* = *x* ⟶ *P y*) *x*
⟨*proof*⟩

lemma *holds-partial-post-state-rel*:
rel-post-state (*R* :: *'a* ⇒ *'b* ⇒ *bool*) *p q* ⟹
holds-partial-post-state (λ*b*. ∀ *a*. *R a b* ⟶ *P a*) *q* ⟹
holds-partial-post-state P p
⟨*proof*⟩

16.4.4 *sim-post-state*

lemma *sim-post-state-eq-iff-le*: *sim-post-state* (=) *p q* ⟷ *p* ≤ *q*
⟨*proof*⟩

lemma *sim-post-state-Success-Success-iff*[*simp*]:
sim-post-state R (*Success r*) (*Success q*) ⟷ (∀ *a*∈*r*. ∃ *b*∈*q*. *R a b*)
⟨*proof*⟩

lemma *sim-post-state-Success2*:

sim-post-state R f (Success q) $\longleftrightarrow$ *holds-post-state* $(\lambda a.\ \exists\, b{\in}q.\ R\ a\ b)$ *f*
⟨*proof*⟩

lemma *sim-post-state-Failure1*[*simp*]: *sim-post-state R Failure q* $\longleftrightarrow$ $q = Failure$
⟨*proof*⟩

lemma *sim-post-state-top2*[*simp*, *intro*]: *sim-post-state R p* $\top$
⟨*proof*⟩

lemma *sim-post-state-top1*[*simp*]: *sim-post-state R* $\top$ *q* $\longleftrightarrow$ $q = \top$
⟨*proof*⟩

lemma *sim-post-state-bot2*[*simp*, *intro*]: *sim-post-state R p* $\bot$ $\longleftrightarrow$ $p = \bot$
⟨*proof*⟩

lemma *sim-post-state-bot1*[*simp*, *intro*]: *sim-post-state R* $\bot$ *q*
⟨*proof*⟩

lemma *sim-post-state-le1*: *sim-post-state R f' g* $\Longrightarrow$ $f \leq f'$ $\Longrightarrow$ *sim-post-state R f g*
⟨*proof*⟩

lemma *sim-post-state-le2*: *sim-post-state R f g* $\Longrightarrow$ $g \leq g'$ $\Longrightarrow$ *sim-post-state R f g'*
⟨*proof*⟩

lemma *sim-post-state-Sup1*:
sim-post-state R (Sup A) f $\longleftrightarrow$ $(\forall\, a{\in}A.$ *sim-post-state R a f*$)$
⟨*proof*⟩

lemma *sim-post-state-Sup2*:
$a \in A$ $\Longrightarrow$ *sim-post-state R f a* $\Longrightarrow$ *sim-post-state R f (Sup A)*
⟨*proof*⟩

lemma *sim-post-state-Sup*:
$\forall\, a{\in}A.\ \exists\, b{\in}B.$ *sim-post-state R a b* $\Longrightarrow$ *sim-post-state R (Sup A) (Sup B)*
⟨*proof*⟩

lemma *sim-post-state-weaken*:
sim-post-state R f g $\Longrightarrow$ $(\bigwedge x\ y.\ R\ x\ y \Longrightarrow Q\ x\ y)$ $\Longrightarrow$ *sim-post-state Q f g*
⟨*proof*⟩

lemma *sim-post-state-trans*:
sim-post-state R f g $\Longrightarrow$ *sim-post-state Q g h* $\Longrightarrow$ $(\bigwedge x\ y\ z.\ R\ x\ y \Longrightarrow Q\ y\ z \Longrightarrow S\ x\ z)$ $\Longrightarrow$
sim-post-state S f h
⟨*proof*⟩

lemma *sim-post-state-refl'*: *holds-partial-post-state* $(\lambda x.\ R\ x\ x)$ *f* $\Longrightarrow$ *sim-post-state*

R f f
⟨*proof*⟩

lemma *sim-post-state-refl*: $(\bigwedge x.\ R\ x\ x) \Longrightarrow$ *sim-post-state R f f*
⟨*proof*⟩

lemma *sim-post-state-pure-post-state2*:
sim-post-state F f (*pure-post-state x*) $\longleftrightarrow$ *holds-post-state* $(\lambda y.\ F\ y\ x)$ *f*
⟨*proof*⟩

16.4.5 *rel-post-state*

lemma *rel-post-state-top*[*simp*, *intro*!]: *rel-post-state R* $\top$ $\top$
⟨*proof*⟩

lemma *rel-post-state-top-iff*[*simp*]:
rel-post-state R $\top$ *p* $\longleftrightarrow$ *p* $= \top$
rel-post-state R p $\top$ $\longleftrightarrow$ *p* $= \top$
⟨*proof*⟩

lemma *rel-post-state-Success-iff*[*simp*]:
rel-post-state R (*Success A*) (*Success B*) $\longleftrightarrow$ *rel-set R A B*
⟨*proof*⟩

lemma *rel-post-state-bot*[*simp*, *intro*!]: *rel-post-state R* $\bot$ $\bot$
⟨*proof*⟩

lemma *rel-post-state-eq-sim-post-state*:
rel-post-state R p q $\longleftrightarrow$ *sim-post-state R p q* $\wedge$ *sim-post-state* R^{-1-1} *q p*
⟨*proof*⟩

lemma *rel-post-state-weaken*:
rel-post-state R f g $\Longrightarrow$ $(\bigwedge x\ y.\ R\ x\ y \Longrightarrow Q\ x\ y) \Longrightarrow$ *rel-post-state Q f g*
⟨*proof*⟩

lemma *rel-post-state-eq*[*relator-eq*]: *rel-post-state* $(=) = (=)$
⟨*proof*⟩

lemma *rel-post-state-mono*[*relator-mono*]:
$A \leq B \Longrightarrow$ *rel-post-state* $A \leq$ *rel-post-state B*
⟨*proof*⟩

lemma *rel-post-state-refl'*: *holds-partial-post-state* $(\lambda x.\ R\ x\ x)$ *f* $\Longrightarrow$ *rel-post-state R f f*
⟨*proof*⟩

lemma *rel-post-state-refl*: $(\bigwedge x.\ R\ x\ x) \Longrightarrow$ *rel-post-state R f f*
⟨*proof*⟩

16.4.6 *lift-post-state*

lemma *lift-post-state-top*: *lift-post-state R* $\top$ = $\top$
⟨*proof*⟩

lemma *lift-post-state-unlift-post-state*: *lift-post-state R p* ≤ *q* ⟷ *p* ≤ *unlift-post-state R q*
⟨*proof*⟩

lemma *lift-post-state-eq-Sup*: *lift-post-state R p* = *Sup* {*q*. *sim-post-state R q p*}
⟨*proof*⟩

lemma *le-lift-post-state-iff*: *q* ≤ *lift-post-state R p* ⟷ *sim-post-state R q p*
⟨*proof*⟩

lemma *lift-post-state-eq*[*simp*]: *lift-post-state* (=) *p* = *p*
⟨*proof*⟩

lemma *lift-post-state-comp*:
lift-post-state R (*lift-post-state Q p*) = *lift-post-state* (*R OO Q*) *p*
⟨*proof*⟩

lemma *sim-post-state-lift*:
sim-post-state Q q (*lift-post-state R p*) ⟷ *sim-post-state* (*Q OO R*) *q p*
⟨*proof*⟩

lemma *lift-post-state-Sup*: *lift-post-state R* (*Sup F*) = (*SUP f*∈*F*. *lift-post-state R f*)
⟨*proof*⟩

lemma *lift-post-state-mono*: *p* ≤ *q* ⟹ *lift-post-state R p* ≤ *lift-post-state R q*
⟨*proof*⟩

16.4.7 *map-post-state*

lemma *map-post-state-top*: *map-post-state f* $\top$ = $\top$
⟨*proof*⟩

lemma *mono-map-post-state*: *s1* ≤ *s2* ⟹ *map-post-state f s1* ≤ *map-post-state f s2*
⟨*proof*⟩

lemma *map-post-state-eq-lift-post-state*: *map-post-state f p* = *lift-post-state* (λ*a b*. *a* = *f b*) *p*
⟨*proof*⟩

lemma *map-post-state-Sup*: *map-post-state f* (*Sup X*) = (*SUP x*∈*X*. *map-post-state f x*)
⟨*proof*⟩

lemma *map-post-state-comp*: *map-post-state f* (*map-post-state g p*) = *map-post-state* (*f* ∘ *g*) *p*
⟨*proof*⟩

lemma *map-post-state-id*[*simp*]: *map-post-state id p* = *p*
⟨*proof*⟩

lemma *map-post-state-pure*[*simp*]: *map-post-state f* (*pure-post-state x*) = *pure-post-state* (*f x*)
⟨*proof*⟩

lemma *sim-post-state-map-post-state1*:
sim-post-state R (*map-post-state f p*) *q* ⟷ *sim-post-state* (λ*x y*. *R* (*f x*) *y*) *p q*
⟨*proof*⟩

lemma *sim-post-state-map-post-state2*:
sim-post-state R p (*map-post-state f q*) ⟷ *sim-post-state* (λ*x y*. *R x* (*f y*)) *p q*
⟨*proof*⟩

lemma *map-post-state-eq-top*[*simp*]: *map-post-state f p* = ⊤ ⟷ *p* = ⊤
⟨*proof*⟩

lemma *map-post-state-eq-bot*[*simp*]: *map-post-state f p* = ⊥ ⟷ *p* = ⊥
⟨*proof*⟩

16.4.8 *vmap-post-state*

lemma *vmap-post-state-top*: *vmap-post-state f* ⊤ = ⊤
⟨*proof*⟩

lemma *vmap-post-state-Sup*:
vmap-post-state f (*Sup X*) = (*SUP x*∈*X*. *vmap-post-state f x*)
⟨*proof*⟩

lemma *vmap-post-state-le-iff*: (*vmap-post-state f p* ≤ *q*) = (*p* ≤ ⨆ {*p*. *vmap-post-state f p* ≤ *q*})
⟨*proof*⟩

lemma *vmap-post-state-eq-lift-post-state*: *vmap-post-state f p* = *lift-post-state* (λ*a b*. *f a* = *b*) *p*
⟨*proof*⟩

lemma *vmap-post-state-comp*: *vmap-post-state f* (*vmap-post-state g p*) = *vmap-post-state* (*g* ∘ *f*) *p*
⟨*proof*⟩

lemma *vmap-post-state-id*: *vmap-post-state id p* = *p*
⟨*proof*⟩

lemma *sim-post-state-vmap-post-state2*:
sim-post-state R p (*vmap-post-state f q*) ⟷
sim-post-state (λ*x y*. ∃ *y'*. *f y'* = *y* ∧ *R x y'*) *p q*
⟨*proof*⟩

lemma *vmap-post-state-le-iff-le-map-post-state*:
map-post-state f p ≤ *q* ⟷ *p* ≤ *vmap-post-state f q*
⟨*proof*⟩

16.4.9 *pure-post-state*

lemma *pure-post-state-Failure*[*simp*]: *pure-post-state v* ≠ *Failure*
⟨*proof*⟩

lemma *pure-post-state-top*[*simp*]: *pure-post-state v* ≠ ⊤
⟨*proof*⟩

lemma *pure-post-state-bot*[*simp*]: *pure-post-state v* ≠ ⊥
⟨*proof*⟩

lemma *pure-post-state-inj*[*simp*]: *pure-post-state v* = *pure-post-state w* ⟷ *v* = *w*
⟨*proof*⟩

lemma *sim-pure-post-state-iff*[*simp*]:
sim-post-state R (*pure-post-state a*) (*pure-post-state b*) ⟷ *R a b*
⟨*proof*⟩

lemma *rel-pure-post-state-iff*[*simp*]:
rel-post-state R (*pure-post-state a*) (*pure-post-state b*) ⟷ *R a b*
⟨*proof*⟩

lemma *pure-post-state-le*[*simp*]: *pure-post-state v* ≤ *pure-post-state w* ⟷ *v* = *w*
⟨*proof*⟩

lemma *Success-eq-pure-post-state*[*simp*]: *Success X* = *pure-post-state x* ⟷ *X* = {*x*}
⟨*proof*⟩

lemma *pure-post-state-eq-Success*[*simp*]: *pure-post-state x* = *Success X* ⟷ *X* = {*x*}
⟨*proof*⟩

lemma *pure-post-state-le-Success*[*simp*]: *pure-post-state x* ≤ *Success X* ⟷ *x* ∈ *X*
⟨*proof*⟩

lemma *Success-le-pure-post-state*[*simp*]: *Success X* ≤ *pure-post-state x* ⟷ *X* ⊆ {*x*}
⟨*proof*⟩

lemma *case-pure-post-state*[*simp*]:
(*case pure-post-state x of Failure* ⇒ *f* | *Success X* ⇒ *S X*) = *S* {*x*}
⟨*proof*⟩

lemma *sim-post-state-Success-pure-post-state*[*simp*]:
sim-post-state R (*Success X*) (*pure-post-state y*) ⟷ (∀ *x*∈*X*. *R x y*)
⟨*proof*⟩

16.4.10 *bind-post-state*

lemma *bind-post-state-top*: *bind-post-state* ⊤ *g* = ⊤
⟨*proof*⟩

lemma *bind-post-state-Sup1*[*simp*]:
bind-post-state (*Sup F*) *g* = (*SUP f*∈*F*. *bind-post-state f g*)
⟨*proof*⟩

lemma *bind-post-state-Sup2*[*simp*]:
G ≠ {} ∨ *f* ≠ *Failure* ⟹ *bind-post-state f* (*Sup G*) = (*SUP g*∈*G*. *bind-post-state f g*)
⟨*proof*⟩

lemma *bind-post-state-sup1*[*simp*]:
bind-post-state (*sup f1 f2*) *g* = *sup* (*bind-post-state f1 g*) (*bind-post-state f2 g*)
⟨*proof*⟩

lemma *bind-post-state-sup2*[*simp*]:
bind-post-state f (*sup g1 g2*) = *sup* (*bind-post-state f g1*) (*bind-post-state f g2*)
⟨*proof*⟩

lemma *bind-post-state-top1*[*simp*]: *bind-post-state* ⊤ *f* = ⊤
⟨*proof*⟩

lemma *bind-post-state-bot1*[*simp*]: *bind-post-state* ⊥ *f* = ⊥
⟨*proof*⟩

lemma *bind-post-state-eq-top*:
bind-post-state f g = ⊤ ⟷ ¬ *holds-post-state* (λ*x*. *g x* ≠ ⊤) *f*
⟨*proof*⟩

lemma *bind-post-state-eq-bot*:
bind-post-state f g = ⊥ ⟷ *holds-post-state* (λ*x*. *g x* = ⊥) *f*
⟨*proof*⟩

lemma *lift-post-state-bind-post-state*:
lift-post-state R (*bind-post-state x g*) = *bind-post-state x* (λ*v*. *lift-post-state R* (*g v*))
⟨*proof*⟩

lemma *vmap-post-state-bind-post-state*:
vmap-post-state f (*bind-post-state p g*) = *bind-post-state p* (λ*v*. *vmap-post-state f* (*g v*))
⟨*proof*⟩

lemma *map-post-state-bind-post-state*:
map-post-state f (*bind-post-state x g*) = *bind-post-state x* (λ*v*. *map-post-state f* (*g v*))
⟨*proof*⟩

lemma *bind-post-state-pure-post-state1*[*simp*]:
bind-post-state (*pure-post-state v*) *f* = *f v*
⟨*proof*⟩

lemma *bind-post-state-pure-post-state2*:
bind-post-state p (λ*x*. *pure-post-state* (*f x*)) = *map-post-state f p*
⟨*proof*⟩

lemma *bind-post-state-map-post-state*:
bind-post-state (*map-post-state f p*) *g* = *bind-post-state p* (λ*x*. *g* (*f x*))
⟨*proof*⟩

lemma *bind-post-state-assoc*[*simp*]:
bind-post-state (*bind-post-state f g*) *h* =
bind-post-state f (λ*v*. *bind-post-state* (*g v*) *h*)
⟨*proof*⟩

lemma *sim-bind-post-state′*:
sim-post-state (λ*x y*. *sim-post-state R* (*g x*) (*k y*)) *f h* ⟹
sim-post-state R (*bind-post-state f g*) (*bind-post-state h k*)
⟨*proof*⟩

lemma *sim-bind-post-state-left-iff*:
sim-post-state R (*bind-post-state f g*) *h* ⟷
h = *Failure* ∨ *holds-post-state* (λ*x*. *sim-post-state R* (*g x*) *h*) *f*
⟨*proof*⟩

lemma *sim-bind-post-state-left*:
holds-post-state (λ*x*. *sim-post-state R* (*g x*) *h*) *f* ⟹
sim-post-state R (*bind-post-state f g*) *h*
⟨*proof*⟩

lemma *sim-bind-post-state-right*:
g ≠ ⊥ ⟹ *holds-partial-post-state* (λ*x*. *sim-post-state R f* (*h x*)) *g* ⟹
sim-post-state R f (*bind-post-state g h*)
⟨*proof*⟩

lemma *sim-bind-post-state*:
sim-post-state Q f h ⟹ (⋀*x y*. *Q x y* ⟹ *sim-post-state R* (*g x*) (*k y*)) ⟹

sim-post-state R (bind-post-state f g) (bind-post-state h k)
⟨*proof*⟩

lemma *rel-bind-post-state'*:
rel-post-state (λ*a b*. *rel-post-state R* (*g1 a*) (*g2 b*)) *f1 f2* ⟹
rel-post-state R (*bind-post-state f1 g1*) (*bind-post-state f2 g2*)
⟨*proof*⟩

lemma *rel-bind-post-state*:
rel-post-state Q f1 f2 ⟹ (⋀*a b*. *Q a b* ⟹ *rel-post-state R* (*g1 a*) (*g2 b*)) ⟹
rel-post-state R (*bind-post-state f1 g1*) (*bind-post-state f2 g2*)
⟨*proof*⟩

lemma *mono-bind-post-state*: *f1* ≤ *f2* ⟹ *g1* ≤ *g2* ⟹ *bind-post-state f1 g1* ≤ *bind-post-state f2 g2*
⟨*proof*⟩

lemma *Sup-eq-Failure*[*simp*]: *Sup X* = *Failure* ⟷ *Failure* ∈ *X*
⟨*proof*⟩

lemma *Failure-inf-iff*: (*Failure* = *x* ⊓ *y*) ⟷ (*x* = *Failure* ∧ *y* = *Failure*)
⟨*proof*⟩

lemma *Success-vimage-singleton-cancel*: *Success* −‘ {*Success X*} = {*X*}
⟨*proof*⟩

lemma *Success-vimage-image-cancel*: *Success* −‘ (λ*x*. *Success* (*f x*)) ‘ *X* = *f* ‘ *X*
⟨*proof*⟩

lemma *Success-image-comp*: (λ*x*. *Success* (*g x*)) ‘ *X* = *Success* ‘ (*g* ‘ *X*)
⟨*proof*⟩

lemma *case-top-post-state*[*simp*]:
(*case* ⊤ *of Failure* ⇒ *f* | *Success X* ⇒ *S X*) = *f*
⟨*proof*⟩

lemma *case-bot-post-state*[*simp*]:
(*case* ⊥ *of Failure* ⇒ *f* | *Success X* ⇒ *S X*) = *S* {}
⟨*proof*⟩

16.5 *exception-or-result* type

instantiation *option* :: (*type*) *default*
begin

definition *default-option* = *None*

instance ⟨*proof*⟩

end

lemma *Some-ne-default*[*simp*]:
Some x ≠ *default default* ≠ *Some x*
default = *None* ⟷ *True None* = *default* ⟷ *True*
⟨*proof*⟩

typedef (**overloaded**) ('*a*::*default*, '*b*) *exception-or-result* =
Inl ‘ (*UNIV* − {*default*}) ∪ *Inr* ‘ *UNIV* :: ('*a* + '*b*) *set*
⟨*proof*⟩

setup-lifting *type-definition-exception-or-result*

context assumes *SORT-CONSTRAINT*('*e*::*default*)
begin

lift-definition *Exception* :: '*e* ⇒ ('*e*, '*v*) *exception-or-result* **is**
λ*e*. *if e* = *default then Inr undefined else Inl e*
⟨*proof*⟩

lift-definition *Result* :: '*v* ⇒ ('*e*, '*v*) *exception-or-result* **is**
λ*v*. *Inr v*
⟨*proof*⟩

end

lift-definition
case-exception-or-result :: ('*e*::*default* ⇒ '*a*) ⇒ ('*v* ⇒ '*a*) ⇒ ('*e*, '*v*) *exception-or-result* ⇒ '*a*
is *case-sum* ⟨*proof*⟩

declare [[*case-translation case-exception-or-result Exception Result*]]

lemma *Result-eq-Result*[*simp*]: *Result a* = *Result b* ⟷ *a* = *b*
⟨*proof*⟩

lemma *Exception-eq-Exception*[*simp*]: *Exception a* = *Exception b* ⟷ *a* = *b*
⟨*proof*⟩

lemma *Result-eq-Exception*[*simp*]: *Result a* = *Exception e* ⟷ (*e* = *default* ∧ *a* = *undefined*)
⟨*proof*⟩

lemma *Exception-eq-Result*[*simp*]: *Exception e* = *Result a* ⟷ (*e* = *default* ∧ *a* = *undefined*)
⟨*proof*⟩

lemma *exception-or-result-cases*[*case-names Exception Result*, *cases type*: *exception-or-result*]:

$(\bigwedge e.\ e \neq default \Longrightarrow x = Exception\ e \Longrightarrow P) \Longrightarrow (\bigwedge v.\ x = Result\ v \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *case-exception-or-result-Result*[*simp*]:
$(case\ Result\ v\ of\ Exception\ e \Rightarrow f\ e \mid Result\ w \Rightarrow g\ w) = g\ v$
⟨*proof*⟩

lemma *case-exception-or-result-Exception*[*simp*]:
$(case\ Exception\ e\ of\ Exception\ e \Rightarrow f\ e \mid Result\ w \Rightarrow g\ w) = (if\ e = default\ then\ g\ undefined\ else\ f\ e)$
⟨*proof*⟩

Caution: for split rules don't use syntax *case r of Exception e* ⇒ *f e* | *Result w* ⇒ *g w*, as this introduces non eta-contracted *f* and *g*, which don't work with the splitter.

lemma *exception-or-result-split*:
$P\ (case\text{-}exception\text{-}or\text{-}result\ f\ g\ r) \longleftrightarrow$
$(\forall e.\ e \neq default \longrightarrow r = Exception\ e \longrightarrow P\ (f\ e)) \wedge$
$(\forall v.\ r = Result\ v \longrightarrow P\ (g\ v))$
⟨*proof*⟩

lemma *exception-or-result-split-asm*:
$P\ (case\text{-}exception\text{-}or\text{-}result\ f\ g\ r) \longleftrightarrow$
$\neg\ ((\exists e.\ r = Exception\ e \wedge e \neq default \wedge \neg\ P\ (f\ e)) \vee$
$(\exists v.\ r = Result\ v \wedge \neg\ P\ (g\ v)))$
⟨*proof*⟩

lemmas *exception-or-result-splits* = *exception-or-result-split exception-or-result-split-asm*

lemma *split-exception-or-result*:
$r = (case\ r\ of\ Exception\ e \Rightarrow Exception\ e \mid Result\ v \Rightarrow Result\ v)$
⟨*proof*⟩

lemma *exception-or-result-nchotomy*:
$\neg\ ((\forall e.\ e \neq default \longrightarrow x \neq Exception\ e) \wedge (\forall v.\ x \neq Result\ v))$ ⟨*proof*⟩

lemma *val-split*:
fixes *r*::(*unit*, *'a*) *exception-or-result*
shows
$P\ (case\text{-}exception\text{-}or\text{-}result\ f\ g\ r) \longleftrightarrow$
$(\forall v.\ r = Result\ v \longrightarrow P\ (g\ v))$
⟨*proof*⟩

lemma *val-split-asm*:
fixes *r*::(*unit*, *'a*) *exception-or-result*
shows $P\ (case\text{-}exception\text{-}or\text{-}result\ f\ g\ r) \longleftrightarrow$
$\neg\ (\exists v.\ r = Result\ v \wedge \neg\ P\ (g\ v))$
⟨*proof*⟩

lemmas *val-splits*[*split*] = *val-split val-split-asm*

instantiation *exception-or-result* :: ({*equal*, *default*}, *equal*) *equal* **begin**

definition *equal-exception-or-result a b* =
(*case a of*
Exception e ⇒ (*case b of Exception e'* ⇒ *HOL.equal e e'* | *Result s* ⇒ *False*)
| *Result r* ⇒ (*case b of Exception e* ⇒ *False* | *Result s* ⇒ *HOL.equal r s*)
)
instance ⟨*proof*⟩

end

definition *is-Exception*:: ('*e*::*default*, '*a*) *exception-or-result* ⇒ *bool* **where**
is-Exception x ≡ (*case x of Exception e* ⇒ *e* ≠ *default* | *Result* - ⇒ *False*)

definition *is-Result*:: ('*e*::*default*, '*a*) *exception-or-result* ⇒ *bool* **where**
is-Result x ≡ (*case x of Exception e* ⇒ *e* = *default* | *Result* - ⇒ *True*)

definition *the-Exception*:: ('*e*::*default*, '*a*) *exception-or-result* ⇒ '*e* **where**
the-Exception x ≡ (*case x of Exception e* ⇒ *e* | *Result* - ⇒ *default*)

definition *the-Result*:: ('*e*::*default*, '*a*) *exception-or-result* ⇒ '*a* **where**
the-Result x ≡ (*case x of Exception e* ⇒ *undefined* | *Result v* ⇒ *v*)

lemma *is-Exception-simps*[*simp*]:
is-Exception (*Exception e*) = (*e* ≠ *default*)
is-Exception (*Result v*) = *False*
⟨*proof*⟩

lemma *is-Result-simps*[*simp*]:
is-Result (*Exception e*) = (*e* = *default*)
is-Result (*Result v*) = *True*
⟨*proof*⟩

lemma *the-Exception-simp*[*simp*]:
the-Exception (*Exception e*) = *e*
⟨*proof*⟩

lemma *the-Exception-Result*:
the-Exception (*Result v*) = *default*
⟨*proof*⟩

definition *undefined-unit*::*unit* ⇒ '*b* **where** *undefined-unit x* ≡ *undefined*

syntax -*Res* :: *pttrn* ⇒ *pttrn* (‹(‹*open-block notation*=‹*prefix Res*››*Res* -)›)
syntax-consts -*Res* ⇌ *case-exception-or-result*
translations λ*Res x*. *b* ⇌ *CONST case-exception-or-result* (*CONST undefined-unit*)

$(\lambda x.\ b)$

term $\lambda Res\ x.\ f\ x$
term $\lambda Res\ (x,\ y,\ z).\ f\ x\ y\ z$
term $\lambda Res\ (x,\ y,\ z)\ s.\ P\ x\ y\ s\ z$

lifting-update *exception-or-result.lifting*
lifting-forget *exception-or-result.lifting*

inductive *rel-exception-or-result*:: $('e::default \Rightarrow 'f::default \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow$
$('e,\ 'a)\ exception\text{-}or\text{-}result \Rightarrow ('f,\ 'b)\ exception\text{-}or\text{-}result \Rightarrow bool$
where
Exception:
$E\ e\ f \Longrightarrow e \neq default \Longrightarrow f \neq default \Longrightarrow$
$rel\text{-}exception\text{-}or\text{-}result\ E\ R\ (Exception\ e)\ (Exception\ f)$ |
Result:
$R\ a\ b \Longrightarrow rel\text{-}exception\text{-}or\text{-}result\ E\ R\ (Result\ a)\ (Result\ b)$

lemma *All-exception-or-result-cases*:
$(\forall x.\ P\ (x::(\text{-},\ \text{-})\ exception\text{-}or\text{-}result)) \longleftrightarrow$
$(\forall err.\ err \neq default \longrightarrow P\ (Exception\ err)) \wedge\ (\forall v.\ P\ (Result\ v))$
⟨*proof*⟩

lemma *Ball-exception-or-result-cases*:
$(\forall x{\in}s.\ P\ (x::(\text{-},\ \text{-})\ exception\text{-}or\text{-}result)) \longleftrightarrow$
$(\forall err.\ err \neq default \longrightarrow Exception\ err \in s \longrightarrow P\ (Exception\ err)) \wedge$
$(\forall v.\ Result\ v \in s \longrightarrow P\ (Result\ v))$
⟨*proof*⟩

lemma *Bex-exception-or-result-cases*:
$(\exists x{\in}s.\ P\ (x::(\text{-},\ \text{-})\ exception\text{-}or\text{-}result)) \longleftrightarrow$
$(\exists err.\ err \neq default \wedge Exception\ err \in s \wedge P\ (Exception\ err)) \vee$
$(\exists v.\ Result\ v \in s \wedge P\ (Result\ v))$
⟨*proof*⟩

lemma *Ex-exception-or-result-cases*:
$(\exists x.\ P\ (x::(\text{-},\ \text{-})\ exception\text{-}or\text{-}result)) \longleftrightarrow$
$(\exists err.\ err \neq default \wedge P\ (Exception\ err)) \vee$
$(\exists v.\ P\ (Result\ v))$
⟨*proof*⟩

lemma *case-distrib-exception-or-result*:
$f\ (case\ x\ of\ Exception\ e \Rightarrow E\ e \mid Result\ v \Rightarrow R\ v) = (case\ x\ of\ Exception\ e \Rightarrow f\ (E\ e) \mid\ Result\ v \Rightarrow f\ (R\ v))$
⟨*proof*⟩

lemma *rel-exception-or-result-Results*[*simp*]:

rel-exception-or-result E R (Result a) (Result b) ⟷ R a b
⟨*proof*⟩

lemma *rel-exception-or-result-Exception*[*simp*]:
e ≠ default ⟹ f ≠ default ⟹
rel-exception-or-result E R (Exception e) (Exception f) ⟷ E e f
⟨*proof*⟩

lemma *rel-exception-or-result-Result-Exception*[*simp*]:
e ≠ default ⟹ ¬ rel-exception-or-result E R (Exception e) (Result b)
f ≠ default ⟹ ¬ rel-exception-or-result E R (Result a) (Exception f)
⟨*proof*⟩

lemma *is-Exception-iff*: *is-Exception x ⟷ (∃ e. e ≠ default ∧ x = Exception e)*
⟨*proof*⟩

lemma *rel-exception-or-result-converse*:
(rel-exception-or-result E R)⁻¹⁻¹ = rel-exception-or-result E⁻¹⁻¹ R⁻¹⁻¹
⟨*proof*⟩

lemma *rel-eception-or-result-Result*[*simp*]:
rel-exception-or-result E R (Result x) (Result y) = R x y
⟨*proof*⟩

lemma *rel-exception-or-result-eq-conv*: *rel-exception-or-result (=) (=) = (=)*
⟨*proof*⟩

lemma *rel-exception-or-result-sum-eq*: *rel-exception-or-result (=) (=) = (=)*
⟨*proof*⟩

definition
map-exception-or-result::
('e::default ⇒ 'f::default) ⇒ ('a ⇒ 'b) ⇒ ('e, 'a) exception-or-result ⇒
('f, 'b) exception-or-result
where
map-exception-or-result E F x =
(case x of Exception e ⇒ Exception (E e) | Result v ⇒ Result (F v))

lemma *map-exception-or-result-Exception*[*simp*]:
e ≠ default ⟹ map-exception-or-result E F (Exception e) = Exception (E e)
⟨*proof*⟩

lemma *map-exception-or-result-Result*[*simp*]:
map-exception-or-result E F (Result v) = Result (F v)
⟨*proof*⟩

lemma *map-exception-or-result-id*: *map-exception-or-result id id x = x*
⟨*proof*⟩

lemma *map-exception-or-result-comp*:
assumes *E2*: $\bigwedge x.\ x \neq default \Longrightarrow E2\ x \neq default$
shows *map-exception-or-result E1 F1* (*map-exception-or-result E2 F2 x*) =
map-exception-or-result (*E1* ∘ *E2*) (*F1* ∘ *F2*) *x*
⟨*proof*⟩

lemma *le-bind-post-state-exception-or-result-cases*[*case-names Exception Result*]:
assumes
holds-partial-post-state ($\lambda(x,\ t).\ \forall e.\ e \neq default \longrightarrow x = Exception\ e \longrightarrow X\ e\ t \leq X'\ e\ t$) *x*
assumes *holds-partial-post-state* ($\lambda(x,\ t).\ \forall v.\ x = Result\ v \longrightarrow V\ v\ t \leq V'\ v\ t$) *x*
shows *bind-post-state x* ($\lambda(r,\ t).$ *case r of Exception e* ⇒ *X e t* | *Result v* ⇒ *V v t*) ≤
bind-post-state x ($\lambda(r,\ t).$ *case r of Exception e* ⇒ *X' e t*| *Result v* ⇒ *V' v t*)
⟨*proof*⟩

16.6 *spec-monad* type

typedef (**overloaded**) ('*e*::*default*, '*a*, '*s*) *spec-monad* =
UNIV :: ('*s* ⇒ (('*e*::*default*, '*a*) *exception-or-result* × '*s*) *post-state*) *set*
morphisms *run Spec*
⟨*proof*⟩

lemma *run-case-prod-distrib*[*simp*]:
run (*case x of* (*r*, *s*) ⇒ *f r s*) *t* = (*case x of* (*r*, *s*) ⇒ *run* (*f r s*) *t*)
⟨*proof*⟩

lemma *image-case-prod-distrib*[*simp*]:
($\lambda(r,\ t).$ *f r t*) ‘ ($\lambda v.$ (*g v*, *h v*)) ‘ *R* = ($\lambda v.$ (*f* (*g v*) (*h v*))) ‘ *R*
⟨*proof*⟩

lemma *run-Spec*[*simp*]: *run* (*Spec p*) *s* = *p s*
⟨*proof*⟩

setup-lifting *type-definition-spec-monad*

lemma *spec-monad-ext*: ($\bigwedge s.$ *run f s* = *run g s*) ⟹ *f* = *g*
⟨*proof*⟩

lemma *spec-monad-ext-iff*: *f* = *g* ⟷ ($\forall s.$ *run f s* = *run g s*)
⟨*proof*⟩

instantiation *spec-monad* :: (*default*, *type*, *type*) *complete-lattice*
begin

lift-definition
less-eq-spec-monad :: ('*e*::*default*, '*r*, '*s*) *spec-monad* ⇒ ('*e*, '*r*, '*s*) *spec-monad* ⇒ *bool*

is $(\leq)$ ⟨*proof*⟩

lift-definition
less-spec-monad :: ('*e*::*default*, '*r*, '*s*) *spec-monad* ⇒ ('*e*, '*r*, '*s*) *spec-monad* ⇒ *bool*
is $(<)$ ⟨*proof*⟩

lift-definition *bot-spec-monad* :: ('*e*::*default*, '*r*, '*s*) *spec-monad* **is** *bot* ⟨*proof*⟩

lift-definition *top-spec-monad* :: ('*e*::*default*, '*r*, '*s*) *spec-monad* **is** *top* ⟨*proof*⟩

lift-definition *Inf-spec-monad* :: ('*e*::*default*, '*r*, '*s*) *spec-monad set* ⇒ ('*e*, '*r*, '*s*) *spec-monad*
is *Inf* ⟨*proof*⟩

lift-definition *Sup-spec-monad* :: ('*e*::*default*, '*r*, '*s*) *spec-monad set* ⇒ ('*e*, '*r*, '*s*) *spec-monad*
is *Sup* ⟨*proof*⟩

lift-definition
sup-spec-monad ::
('*e*::*default*, '*r*, '*s*) *spec-monad* ⇒ ('*e*, '*r*, '*s*) *spec-monad* ⇒ ('*e*, '*r*, '*s*) *spec-monad*
is *sup* ⟨*proof*⟩

lift-definition
inf-spec-monad ::
('*e*::*default*, '*r*, '*s*) *spec-monad* ⇒ ('*e*, '*r*, '*s*) *spec-monad* ⇒ ('*e*, '*r*, '*s*) *spec-monad*
is *inf* ⟨*proof*⟩

instance
⟨*proof*⟩
end

lift-definition *fail* :: ('*e*::*default*, '*a*, '*s*) *spec-monad*
is λ*s*. *Failure* ⟨*proof*⟩

lift-definition
bind-exception-or-result ::
('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒
(('*e*, '*a*) *exception-or-result* ⇒ ('*f*::*default*, '*b*, '*s*) *spec-monad*) ⇒ ('*f*, '*b*, '*s*) *spec-monad*
is
λ*f h s*. *bind-post-state* (*f s*) (λ(*v*, *t*). *h v t*) ⟨*proof*⟩

lift-definition *bind-handle* ::
('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒
('*a* ⇒ ('*f*, '*b*, '*s*) *spec-monad*) ⇒ ('*e* ⇒ ('*f*, '*b*, '*s*) *spec-monad*) ⇒
('*f*::*default*, '*b*, '*s*) *spec-monad*
is λ*f g h s*. *bind-post-state* (*f s*) (λ(*r*, *t*). *case r of Exception e* ⇒ *h e t* | *Result v*

$\Rightarrow$ *g v t*) ⟨*proof*⟩

lift-definition *yield* :: (*'e*, *'a*) *exception-or-result* $\Rightarrow$ (*'e*::*default*, *'a*, *'s*) *spec-monad*
  is λ*r s*. *pure-post-state* (*r*, *s*) ⟨*proof*⟩

abbreviation *return r* $\equiv$ *yield* (*Result r*)

abbreviation *skip* $\equiv$ *return* ()

abbreviation *throw-exception-or-result e* $\equiv$ *yield* (*Exception e*)

lift-definition *get-state* :: (*'e*::*default*, *'s*, *'s*) *spec-monad*
  is λ*s*. *pure-post-state* (*Result s*, *s*) ⟨*proof*⟩

lift-definition *set-state* :: *'s* $\Rightarrow$ (*'e*::*default*, *unit*, *'s*) *spec-monad*
  is λ*t s*. *pure-post-state* (*Result* (), *t*) ⟨*proof*⟩

lift-definition *map-value* ::
    ((*'e*::*default*, *'a*) *exception-or-result* $\Rightarrow$ (*'f*::*default*, *'b*) *exception-or-result*) $\Rightarrow$
      (*'e*, *'a*, *'s*) *spec-monad* $\Rightarrow$ (*'f*, *'b*, *'s*) *spec-monad*
  is λ*f g s*. *map-post-state* (*apfst f*) (*g s*) ⟨*proof*⟩

lift-definition *vmap-value* ::
    ((*'e*::*default*, *'a*) *exception-or-result* $\Rightarrow$ (*'f*::*default*, *'b*) *exception-or-result*) $\Rightarrow$
      (*'f*, *'b*, *'s*) *spec-monad* $\Rightarrow$ (*'e*, *'a*, *'s*) *spec-monad*
  is λ*f g s*. *vmap-post-state* (*apfst f*) (*g s*) ⟨*proof*⟩

definition *bind* ::
    (*'e*::*default*, *'a*, *'s*) *spec-monad* $\Rightarrow$ (*'a* $\Rightarrow$ (*'e*, *'b*, *'s*) *spec-monad*) $\Rightarrow$ (*'e*, *'b*, *'s*)
spec-monad
where
  bind f g = *bind-handle f g throw-exception-or-result*

adhoc-overloading
  Monad-Syntax.bind $\rightleftharpoons$ *bind*

lift-definition *lift-state* ::
    (*'s* $\Rightarrow$ *'t* $\Rightarrow$ *bool*) $\Rightarrow$ (*'e*::*default*, *'a*, *'t*) *spec-monad* $\Rightarrow$ (*'e*, *'a*, *'s*) *spec-monad*
  is λ*R p s*. *lift-post-state* (*rel-prod* (=) *R*) (*SUP t*$\in${*t*. *R s t*}. *p t*) ⟨*proof*⟩

definition *exec-concrete* ::
  (*'s* $\Rightarrow$ *'t*) $\Rightarrow$ (*'e*::*default*, *'a*, *'s*) *spec-monad* $\Rightarrow$ (*'e*, *'a*, *'t*) *spec-monad*
where
  exec-concrete st = *lift-state* (λ*t s*. *t* = *st s*)

definition *exec-abstract* ::
  (*'s* $\Rightarrow$ *'t*) $\Rightarrow$ (*'e*::*default*, *'a*, *'t*) *spec-monad* $\Rightarrow$ (*'e*, *'a*, *'s*) *spec-monad*
where
  exec-abstract st = *lift-state* (λ*s t*. *t* = *st s*)

definition *select-value* :: (*'e*, *'a*) *exception-or-result set* ⇒ (*'e*::*default*, *'a*, *'s*) *spec-monad*
where
select-value R = (*SUP r*∈*R*. *yield r*)

definition *select* :: *'a set* ⇒ (*'e*::*default*, *'a*, *'s*) *spec-monad* **where**
select S = (*SUP a*∈*S*. *return a*)

definition *unknown* :: (*'e*::*default*, *'a*, *'s*) *spec-monad* **where**
unknown ≡ *select UNIV*

definition *gets* :: (*'s* ⇒ *'a*) ⇒ (*'e*::*default*, *'a*, *'s*) *spec-monad* **where**
gets f = *bind get-state* (λ*s*. *return* (*f s*))

definition *assert-opt* :: *'a option* ⇒ (*'e*::*default*, *'a*, *'s*) *spec-monad* **where**
assert-opt v = (*case v of None* ⇒ *fail* | *Some v* ⇒ *return v*)

definition *gets-the* :: (*'s* ⇒ *'a option*) ⇒ (*'e*::*default*, *'a*, *'s*) *spec-monad* **where**
gets-the f = *bind* (*gets f*) *assert-opt*

definition *modify* :: (*'s* ⇒ *'s*) ⇒ (*'e*::*default*, *unit*, *'s*) *spec-monad* **where**
modify f = *bind get-state* (λ*s*. *set-state* (*f s*))

definition *assume* :: *bool* ⇒ (*'e*::*default*, *unit*, *'s*) *spec-monad* **where**
assume P = (*if P then return* () *else bot*)

definition *assert* :: *bool* ⇒ (*'e*::*default*, *unit*, *'s*) *spec-monad* **where**
assert P = (*if P then return* () *else top*)

definition *assuming* :: (*'s* ⇒ *bool*) ⇒ (*'e*::*default*, *unit*, *'s*) *spec-monad* **where**
assuming P = *do* {*s* ← *get-state*; *assume* (*P s*)}

definition *guard*:: (*'s* ⇒ *bool*) ⇒ (*'e*::*default*, *unit*, *'s*) *spec-monad* **where**
guard P = *do* {*s* ← *get-state*; *assert* (*P s*)}

definition *assume-result-and-state* :: (*'s* ⇒ (*'a* × *'s*) *set*) ⇒ (*'e*::*default*, *'a*, *'s*)
spec-monad **where**
assume-result-and-state f = *do* {*s* ← *get-state*; (*v*, *t*) ← *select* (*f s*); *set-state t*;
return v}

lift-definition *assume-outcome* ::
(*'s* ⇒ ((*'e*, *'a*) *exception-or-result* × *'s*) *set*) ⇒ (*'e*::*default*, *'a*, *'s*) *spec-monad*
is λ*f s*. *Success* (*f s*) ⟨*proof*⟩

definition *assert-result-and-state* ::
(*'s* ⇒ (*'a* × *'s*) *set*) ⇒ (*'e*::*default*, *'a*, *'s*) *spec-monad*
where
assert-result-and-state f =

do {*s* ← *get-state*; *assert* (*f s* ≠ {}); (*v*, *t*) ← *select* (*f s*); *set-state t*; *return v*}

abbreviation *state-select* :: ('*s* × '*s*) *set* ⇒ ('*e*::*default*, *unit*, '*s*) *spec-monad*
where
state-select r ≡ *assert-result-and-state* (λ*s*. {((), *s*'). (*s*, *s*') ∈ *r*})

definition *condition* ::
('*s* ⇒ *bool*) ⇒ ('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒ ('*e*, '*a*, '*s*) *spec-monad* ⇒
('*e*, '*a*, '*s*) *spec-monad*
where
condition C T F =
bind get-state (λ*s*. *if C s then T else F*)

notation (**output**)
condition (‹(‹*notation*=‹*prefix condition*››*condition* (-)// (-)// (-))› [*1000*,*1000*,*1000*] *999*)

definition *when* ::*bool* ⇒ ('*e*::*default*, *unit*, '*s*) *spec-monad* ⇒ ('*e*, *unit*, '*s*) *spec-monad*
where *when c f* ≡ *condition* (λ-. *c*) *f skip*

abbreviation *unless* ::*bool* ⇒ ('*e*::*default*, *unit*, '*s*) *spec-monad* ⇒ ('*e*, *unit*, '*s*) *spec-monad*
where *unless c* ≡ *when* (¬*c*)

definition *on-exit'* ::
('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒ ('*e*, *unit*, '*s*) *spec-monad* ⇒ ('*e*, '*a*, '*s*) *spec-monad*
where
on-exit' f c ≡
bind-exception-or-result f (λ*r*. *do* { *c*; *yield r* })

definition *on-exit* ::
('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒ ('*s* × '*s*) *set* ⇒ ('*e*, '*a*, '*s*) *spec-monad*
where
on-exit f cleanup ≡ *on-exit' f* (*state-select cleanup*)

abbreviation *guard-on-exit* ::
('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒ ('*s* ⇒ *bool*) ⇒ ('*s* × '*s*) *set* ⇒ ('*e*, '*a*, '*s*) *spec-monad*
where
guard-on-exit f grd cleanup ≡ *on-exit' f* (*bind* (*guard grd*) (λ-. *state-select cleanup*))

abbreviation *assume-on-exit* ::
('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒ ('*s* ⇒ *bool*) ⇒ ('*s* × '*s*) *set* ⇒ ('*e*, '*a*, '*s*) *spec-monad*
where
assume-on-exit f grd cleanup ≡
on-exit' f (*bind* (*assuming grd*) (λ-. *state-select cleanup*))

lift-definition *run-bind* ::
  ('e::*default*, 'a, 't) *spec-monad* ⇒ 't ⇒
  (('e, 'a) *exception-or-result* ⇒ 't ⇒ ('f::*default*, 'b, 's) *spec-monad*) ⇒
  ('f::*default*, 'b, 's) *spec-monad*
is λf t g s. *bind-post-state* (f t) (λ(r, t). g r t s) ⟨*proof*⟩

type-synonym ('e, 'a, 's) *predicate* = ('e, 'a) *exception-or-result* ⇒ 's ⇒ *bool*

lift-definition *runs-to* ::
  ('e::*default*, 'a, 's) *spec-monad* ⇒ 's ⇒ ('e, 'a, 's) *predicate* ⇒ *bool*
  (‹(‹*open-block notation*=‹*mixfix runs-to*››-/ · - ⦃ - ⦄)› [*61*, *1000*, *0*] *30*) — syntax -*do-block* has 62
is λf s Q. *holds-post-state* (λ(r, t). Q r t) (f s) ⟨*proof*⟩

lift-definition *runs-to-partial* ::
  ('e::*default*, 'a, 's) *spec-monad* ⇒ 's ⇒ ('e, 'a, 's) *predicate* ⇒ *bool*
  (‹(‹*open-block notation*=‹*mixfix runs-to-partial*››-/ · - *?*⦃ - ⦄)› [*61*, *1000*, *0*] *30*)
is λf s Q. *holds-partial-post-state* (λ(r, t). Q r t) (f s) ⟨*proof*⟩

lift-definition *refines* ::
  ('e::*default*, 'a, 's) *spec-monad* ⇒ ('f::*default*, 'b, 't) *spec-monad* ⇒ 's ⇒ 't ⇒
  ((('e, 'a) *exception-or-result* × 's) ⇒ (('f, 'b) *exception-or-result* × 't) ⇒ *bool*) ⇒ *bool*
is λf g s t R. *sim-post-state* R (f s) (g t) ⟨*proof*⟩

lift-definition *rel-spec* ::
  ('e::*default*, 'a, 's) *spec-monad* ⇒ ('f::*default*, 'b, 't) *spec-monad* ⇒ 's ⇒ 't ⇒
  ((('e, 'a) *exception-or-result* × 's) ⇒ (('f, 'b) *exception-or-result* × 't) ⇒ *bool*) ⇒ *bool*
is
  λf g s t R. *rel-post-state* R (f s) (g t) ⟨*proof*⟩

definition *rel-spec-monad* ::
  ('s ⇒ 't ⇒ *bool*) ⇒ (('e, 'a) *exception-or-result* ⇒ ('f, 'b) *exception-or-result* ⇒ *bool*) ⇒
    ('e::*default*, 'a, 's) *spec-monad* ⇒ ('f::*default*, 'b, 't) *spec-monad* ⇒ *bool*
where
  rel-spec-monad R Q f g =
    (∀ s t. R s t ⟶ *rel-post-state* (*rel-prod* Q R) (*run* f s) (*run* g t))

lift-definition *always-progress* :: ('e::*default*, 'a, 's) *spec-monad* ⇒ *bool* **is**
  λp. ∀ s. p s ≠ *bot* ⟨*proof*⟩

named-theorems *run-spec-monad Simplification rules to run a Spec monad*
named-theorems *runs-to-iff Equivalence theorems for runs to predicate*
named-theorems *always-progress-intros intro rules for always-progress predicate*

lemma *runs-to-partialI*: (⋀r x. P r x) ⟹ f · s *?*⦃ P ⦄

⟨*proof*⟩

lemma *runs-to-partial-True*: $f \cdot s \ ?\{\!|\ \lambda r\ s.\ True\ |\!\}$
⟨*proof*⟩

lemma *runs-to-partial-conj*:
$f \cdot s \ ?\{\!|\ P\ |\!\} \Longrightarrow f \cdot s \ ?\{\!|\ Q\ |\!\} \Longrightarrow f \cdot s \ ?\{\!|\ \lambda r\ s.\ P\ r\ s \wedge Q\ r\ s\ |\!\}$
⟨*proof*⟩

lemma *refines-iff'*:
$refines\ f\ g\ s\ t\ R \longleftrightarrow (\forall P.\ g \cdot t \ \{\!|\ \lambda r\ s.\ \forall p\ t.\ R\ (p,\ t)\ (r,\ s) \longrightarrow P\ p\ t\ |\!\} \longrightarrow f \cdot s \ \{\!|\ P\ |\!\})$
⟨*proof*⟩

lemma *refines-weaken*:
$refines\ f\ g\ s\ t\ R \Longrightarrow (\bigwedge r\ s\ q\ t.\ R\ (r,\ s)\ (q,\ t) \Longrightarrow Q\ (r,\ s)\ (q,\ t)) \Longrightarrow refines\ f\ g\ s\ t\ Q$
⟨*proof*⟩

lemma *refines-mono*:
$(\bigwedge r\ s\ q\ t.\ R\ (r,\ s)\ (q,\ t) \Longrightarrow Q\ (r,\ s)\ (q,\ t)) \Longrightarrow refines\ f\ g\ s\ t\ R \Longrightarrow refines\ f\ g\ s\ t\ Q$
⟨*proof*⟩

lemma *refines-refl*: $(\bigwedge r\ t.\ R\ (r,\ t)\ (r,\ t)) \Longrightarrow refines\ f\ f\ s\ s\ R$
⟨*proof*⟩

lemma *refines-trans*:
$refines\ f\ g\ s\ t\ R \Longrightarrow refines\ g\ h\ t\ u\ Q \Longrightarrow$
$(\bigwedge r\ s\ p\ t\ q\ u.\ R\ (r,\ s)\ (p,\ t) \Longrightarrow Q\ (p,\ t)\ (q,\ u) \Longrightarrow S\ (r,\ s)\ (q,\ u)) \Longrightarrow$
$refines\ f\ h\ s\ u\ S$
⟨*proof*⟩

lemma *refines-trans'*: $refines\ f\ g\ s\ t1\ R \Longrightarrow refines\ g\ h\ t1\ t2\ Q \Longrightarrow refines\ f\ h\ s\ t2\ (R\ OO\ Q)$
⟨*proof*⟩

lemma *refines-strengthen*:
$refines\ f\ g\ s\ t\ R \Longrightarrow f \cdot s \ ?\{\!|\ F\ |\!\} \Longrightarrow g \cdot t \ ?\{\!|\ G\ |\!\} \Longrightarrow$
$(\bigwedge x\ s\ y\ t.\ R\ (x,\ s)\ (y,\ t) \Longrightarrow F\ x\ s \Longrightarrow G\ y\ t \Longrightarrow Q\ (x,\ s)\ (y,\ t)) \Longrightarrow$
$refines\ f\ g\ s\ t\ Q$
⟨*proof*⟩

lemma *refines-strengthen1*:
$refines\ f\ g\ s\ t\ R \Longrightarrow f \cdot s \ ?\{\!|\ F\ |\!\} \Longrightarrow$
$(\bigwedge x\ s\ y\ t.\ R\ (x,\ s)\ (y,\ t) \Longrightarrow F\ x\ s \Longrightarrow Q\ (x,\ s)\ (y,\ t)) \Longrightarrow$
$refines\ f\ g\ s\ t\ Q$
⟨*proof*⟩

lemma *refines-strengthen2*:
refines f g s t R $\Longrightarrow$ *g* $\cdot$ *t ?*$\{\!|$ *G* $|\!\}$ $\Longrightarrow$
($\bigwedge$*x s y t*. *R* (*x*, *s*) (*y*, *t*) $\Longrightarrow$ *G y t* $\Longrightarrow$ *Q* (*x*, *s*) (*y*, *t*)) $\Longrightarrow$
refines f g s t Q
⟨*proof*⟩

lemma *refines-cong-cases*:
assumes $\bigwedge$*e s e' s'*. *e* $\neq$ *default* $\Longrightarrow$ *e'* $\neq$ *default* $\Longrightarrow$
R (*Exception e*, *s*) (*Exception e'*, *s'*) $\longleftrightarrow$ *Q* (*Exception e*, *s*) (*Exception e'*, *s'*)
assumes $\bigwedge$*e s x' s'*. *e* $\neq$ *default* $\Longrightarrow$
R (*Exception e*, *s*) (*Result x'*, *s'*) $\longleftrightarrow$ *Q* (*Exception e*, *s*) (*Result x'*, *s'*)
assumes $\bigwedge$*x s e' s'*. *e'* $\neq$ *default* $\Longrightarrow$
R (*Result x*, *s*) (*Exception e'*, *s'*) $\longleftrightarrow$ *Q* (*Result x*, *s*) (*Exception e'*, *s'*)
assumes $\bigwedge$*x s x' s'*. *R* (*Result x*, *s*) (*Result x'*, *s'*) $\longleftrightarrow$ *Q* (*Result x*, *s*) (*Result x'*, *s'*)
shows *refines f f' s s' R* $\longleftrightarrow$ *refines f f' s s' Q*
⟨*proof*⟩

lemma *refines-parallel*:
always-progress g $\Longrightarrow$ *f* $\cdot$ *s* $\{\!|$ *P* $|\!\}$ $\Longrightarrow$ *g* $\cdot$ *t* $\{\!|$ *Q* $|\!\}$ $\Longrightarrow$
refines f g s t (λ(*r*, *s*) (*q*, *t*). *P r s* $\wedge$ *Q q t*)
⟨*proof*⟩

lemma *runs-to-partial-weaken*[*runs-to-vcg-weaken*]:
f $\cdot$ *s ?*$\{\!|Q|\!\}$ $\Longrightarrow$ ($\bigwedge$*r t*. *Q r t* $\Longrightarrow$ *Q' r t*) $\Longrightarrow$ *f* $\cdot$ *s ?*$\{\!|Q'|\!\}$
⟨*proof*⟩

lemma *runs-to-partial-Res-True*[*simp*]: *p* $\cdot$ *s ?*$\{\!|$ λ*Res a s*. *True* $|\!\}$
⟨*proof*⟩

lemma *runs-to-weaken*[*runs-to-vcg-weaken*]: *f* $\cdot$ *s* $\{\!|Q|\!\}$ $\Longrightarrow$ ($\bigwedge$*r t*. *Q r t* $\Longrightarrow$ *Q' r t*) $\Longrightarrow$ *f* $\cdot$ *s* $\{\!|Q'|\!\}$
⟨*proof*⟩

lemma *runs-to-partial-imp-runs-to-partial*[*mono*]:
($\bigwedge$*a s*. *P a s* $\longrightarrow$ *Q a s*) $\Longrightarrow$ *f* $\cdot$ *s ?*$\{\!|$ *P* $|\!\}$ $\longrightarrow$ *f* $\cdot$ *s ?*$\{\!|$ *Q* $|\!\}$
⟨*proof*⟩

lemma *runs-to-imp-runs-to*[*mono*]:
($\bigwedge$*a s*. *P a s* $\longrightarrow$ *Q a s*) $\Longrightarrow$ *f* $\cdot$ *s* $\{\!|$ *P* $|\!\}$ $\longrightarrow$ *f* $\cdot$ *s* $\{\!|$ *Q* $|\!\}$
⟨*proof*⟩

lemma *refines-imp-refines*[*mono*]:
($\bigwedge$*a s*. *P a s* $\longrightarrow$ *Q a s*) $\Longrightarrow$ *refines f g s t P* $\longrightarrow$ *refines f g s t Q*
⟨*proof*⟩

lemma *runs-to-strengthen*:
f $\cdot$ *s* $\{\!|$ *P* $|\!\}$ $\Longrightarrow$ *f* $\cdot$ *s ?*$\{\!|$ *Q* $|\!\}$ $\Longrightarrow$ ($\bigwedge$*a s*. *P a s* $\Longrightarrow$ *Q a s* $\Longrightarrow$ *R a s*) $\Longrightarrow$ *f* $\cdot$ *s* $\{\!|$ *R* $|\!\}$

⟨*proof*⟩

lemma *runs-to-partial-weaken-rel-spec*:
$g \cdot t$ *?*⦃ $\lambda r\ t'.\ \forall p\ s'.\ R\ (p,\ s')\ (r,\ t') \longrightarrow P\ p\ s'$ ⦄ $\Longrightarrow$ *rel-spec* $f\ g\ s\ t\ R \Longrightarrow f \cdot s$ *?*⦃ P ⦄
⟨*proof*⟩

lemma *runs-to-cong-cases*:
assumes $\bigwedge e\ s.\ e \neq default \Longrightarrow P\ (Exception\ e)\ s \longleftrightarrow Q\ (Exception\ e)\ s$
assumes $\bigwedge x\ s.\ P\ (Result\ x)\ s \longleftrightarrow Q\ (Result\ x)\ s$
shows $f \cdot s$ ⦃ P ⦄ $\longleftrightarrow f \cdot s$ ⦃ Q ⦄
⟨*proof*⟩

lemma *le-spec-monad-le-refines-iff*: $f \leq g \longleftrightarrow (\forall s.\ refines\ f\ g\ s\ s\ (=))$
⟨*proof*⟩

lemma *spec-monad-le-iff*: $f \leq g \longleftrightarrow (\forall P\ (s::'a).\ g \cdot s$ ⦃ P ⦄ $\longrightarrow f \cdot s$ ⦃ P ⦄)
⟨*proof*⟩

lemma *spec-monad-eq-iff*: $f = g \longleftrightarrow (\forall P\ s.\ f \cdot s$ ⦃ P ⦄ $\longleftrightarrow g \cdot s$ ⦃ P ⦄)
⟨*proof*⟩

lemma *spec-monad-eqI*: $(\bigwedge P\ s.\ f \cdot s$ ⦃ P ⦄ $\longleftrightarrow g \cdot s$ ⦃ P ⦄) $\Longrightarrow f = g$
⟨*proof*⟩

lemma *runs-to-Sup-iff*[*runs-to-iff*]: $(Sup\ X) \cdot s$ ⦃ P ⦄ $\longleftrightarrow (\forall x{\in}X.\ x \cdot s$ ⦃ P ⦄)
⟨*proof*⟩

lemma *runs-to-lfp*:
assumes f: *mono f*
assumes *x-s*: $P\ x\ s$
assumes $*$: $\bigwedge p.\ (\bigwedge x\ s.\ P\ x\ s \Longrightarrow p\ x \cdot s$ ⦃ R ⦄) $\Longrightarrow (\bigwedge x\ s.\ P\ x\ s \Longrightarrow f\ p\ x \cdot s$ ⦃ R ⦄)
shows *lfp f x* $\cdot\ s$ ⦃ R ⦄
⟨*proof*⟩

lemma *runs-to-partial-alt*:
$f \cdot s$ *?*⦃ Q ⦄ $\longleftrightarrow$ *run* $f\ s = \top \vee f \cdot s$ ⦃ Q ⦄
⟨*proof*⟩

lemma *runs-to-of-runs-to-partial-runs-to'*:
$f \cdot s$ ⦃ P ⦄ $\Longrightarrow f \cdot s$ *?*⦃ Q ⦄ $\Longrightarrow f \cdot s$ ⦃ Q ⦄
⟨*proof*⟩

lemma *runs-to-partial-of-runs-to*:
$f \cdot s$ ⦃ Q ⦄ $\Longrightarrow f \cdot s$ *?*⦃ Q ⦄
⟨*proof*⟩

lemma *runs-to-partial-tivial*[*simp*]: $f \cdot s$ *?*⦃ λ- -. *True* ⦄

⟨*proof*⟩

lemma *le-spec-monad-le-run-iff*: $f \leq g \longleftrightarrow (\forall s.\ run\ f\ s \leq run\ g\ s)$
⟨*proof*⟩

lemma *refines-le-run-trans*: *refines f g s s1 R* $\Longrightarrow$ *run g s1* $\leq$ *run h s2* $\Longrightarrow$ *refines f h s s2 R*
⟨*proof*⟩

lemma *le-run-refines-trans*: *run f s* $\leq$ *run g s1* $\Longrightarrow$ *refines g h s1 s2 R* $\Longrightarrow$ *refines f h s s2 R*
⟨*proof*⟩

lemma *refines-le-trans*: *refines f g s t R* $\Longrightarrow$ $g \leq h$ $\Longrightarrow$ *refines f h s t R*
⟨*proof*⟩

lemma *le-refines-trans*: $f \leq g$ $\Longrightarrow$ *refines g h s t R* $\Longrightarrow$ *refines f h s t R*
⟨*proof*⟩

lemma *le-run-refines-iff*: *run f s* $\leq$ *run g t* $\longleftrightarrow$ *refines f g s t* (=)
⟨*proof*⟩

lemma *refines-top*[*simp*]: *refines f* $\top$ *s t R*
⟨*proof*⟩

lemma *refines-Sup1*: *refines* (*Sup F*) *g s s′ R* $\longleftrightarrow$ $(\forall f{\in}F.$ *refines f g s s′ R*$)$
⟨*proof*⟩

lemma *monotone-le-iff-refines*:
monotone R ($\leq$) *F* $\longleftrightarrow$ $(\forall x\ y.\ R\ x\ y \longrightarrow (\forall s.$ *refines* (*F x*) (*F y*) *s s* (=)$))$
⟨*proof*⟩

lemma *refines-iff-runs-to*:
refines f g s t R $\longleftrightarrow$ $(\forall P.\ g \cdot t \mathbin{\{\!|} \lambda r\ t'.\ \forall p\ s'.\ R\ (p,\ s')\ (r,\ t') \longrightarrow P\ p\ s' \mathbin{|\!\}} \longrightarrow f \cdot s \mathbin{\{\!|} P \mathbin{|\!\}})$
⟨*proof*⟩

lemma *runs-to-refines-weaken′*:
refines f g s t R $\Longrightarrow$ $g \cdot t \mathbin{\{\!|} \lambda r\ t'.\ \forall p\ s'.\ R\ (p,\ s')\ (r,\ t') \longrightarrow P\ p\ s' \mathbin{|\!\}} \Longrightarrow f \cdot s \mathbin{\{\!|} P \mathbin{|\!\}}$
⟨*proof*⟩

lemma *runs-to-refines-weaken*: *refines f f′ s t* (=) $\Longrightarrow$ $f' \cdot t \mathbin{\{\!|} P \mathbin{|\!\}} \Longrightarrow f \cdot s \mathbin{\{\!|} P \mathbin{|\!\}}$
⟨*proof*⟩

lemma *refinesD-runs-to*:
assumes *f-g*: *refines f g s t R* **and** *g*: $g \cdot t \mathbin{\{\!|} P \mathbin{|\!\}}$
shows $f \cdot s \mathbin{\{\!|} \lambda r\ t.\ \exists r'\ t'.\ R\ (r,\ t)\ (r',\ t') \wedge P\ r'\ t' \mathbin{|\!\}}$

⟨*proof*⟩

lemma *rel-spec-iff-refines*:
rel-spec f g s t R ⟷ *refines f g s t R* ∧ *refines g f t s* (R^{-1-1})
⟨*proof*⟩

lemma *rel-spec-symm*: *rel-spec f g s t R* ⟷ *rel-spec g f t s* R^{-1-1}
⟨*proof*⟩

lemma *rel-specD-refines1*: *rel-spec f g s t R* ⟹ *refines f g s t R*
⟨*proof*⟩

lemma *rel-spec-mono*: $P \leq Q$ ⟹ *rel-spec f g s t P* ⟹ *rel-spec f g s t Q*
⟨*proof*⟩

lemma *rel-spec-eq*: *rel-spec f g s t* (=) ⟷ *run f s* = *run g t*
⟨*proof*⟩

lemma *rel-spec-eq-conv*: (∀ *s*. *rel-spec f g s s* (=)) ⟷ *f* = *g*
⟨*proof*⟩

lemma *rel-spec-eqD*: (⋀*s*. *rel-spec f g s s* (=)) ⟹ *f* = *g*
⟨*proof*⟩

lemma *rel-spec-refl'*: *f* ⋅ *s* ?{⦃ λ*r s*. *R* (*r*, *s*) (*r*, *s*)⦄} ⟹ *rel-spec f f s s R*
⟨*proof*⟩

lemma *rel-spec-refl*: (⋀*r s*. *R* (*r*, *s*) (*r*, *s*)) ⟹ *rel-spec f f s s R*
⟨*proof*⟩

lemma *refines-same-runs-to-partialI*:
f ⋅ *s* ?{⦃λ*r s'*. *R* (*r*, *s'*) (*r*, *s'*)⦄} ⟹ *refines f f s s R*
⟨*proof*⟩

lemma *refines-same-runs-toI*:
f ⋅ *s* {⦃λ*r s'*. *R* (*r*, *s'*) (*r*, *s'*)⦄} ⟹ *refines f f s s R*
⟨*proof*⟩

lemma *always-progress-case-prod*[*always-progress-intros*]:
always-progress (*f* (*fst p*) (*snd p*)) ⟹ *always-progress* (*case-prod f p*)
⟨*proof*⟩

lemma *refines-runs-to-partial-fuse*:
refines f f' s s' Q ⟹ *f* ⋅ *s* ?{⦃*P*⦄} ⟹
refines f f' s s' (λ(*r*,*t*) (*r'*,*t'*). *Q* (*r*,*t*) (*r'*,*t'*) ∧ *P r t*)
⟨*proof*⟩

lemma *runs-to-refines*:
f' ⋅ *s'* {⦃ *P* ⦄} ⟹ *refines f f' s s' Q* ⟹ (⋀*x s y t*. *P x s* ⟹ *Q* (*y*, *t*) (*x*, *s*) ⟹

$R\ y\ t) \Longrightarrow$
$f \cdot s \{\!|\ R\ |\!\}$
⟨*proof*⟩

lemma *runs-to-partial-subst-Res*: $f \cdot s\ ?\{\!|\lambda r.\ P\ (\textit{the-Result}\ r)|\!\} \longleftrightarrow f \cdot s\ ?\{\!|\lambda \textit{Res}\ r.\ P\ r|\!\}$
⟨*proof*⟩

lemma *runs-to-subst-Res*: $f \cdot s\ \{\!|\lambda r.\ P\ (\textit{the-Result}\ r)|\!\} \longleftrightarrow f \cdot s\ \{\!|\lambda \textit{Res}\ r.\ P\ r|\!\}$
⟨*proof*⟩

lemma *runs-to-Res*[*simp*]: $f \cdot s\ \{\!|\lambda r\ t.\ \forall v.\ r = \textit{Result}\ v \longrightarrow P\ v\ t|\!\} \longleftrightarrow f \cdot s\ \{\!|\lambda \textit{Res}\ v\ t.\ P\ v\ t|\!\}$
⟨*proof*⟩

lemma *runs-to-partial-Res*[*simp*]:
$f \cdot s\ ?\{\!|\lambda r\ t.\ \forall v.\ r = \textit{Result}\ v \longrightarrow P\ v\ t|\!\} \longleftrightarrow f \cdot s\ ?\{\!|\lambda \textit{Res}\ v\ t.\ P\ v\ t|\!\}$
⟨*proof*⟩

lemma *rel-spec-runs-to*:
assumes *f*: $f \cdot s \{\!|\ P\ |\!\}$ *always-progress f*
and *g*: $g \cdot t \{\!|\ Q\ |\!\}$ *always-progress g*
and *P*: $(\bigwedge r\ s'\ p\ t'.\ P\ r\ s' \Longrightarrow Q\ p\ t' \Longrightarrow R\ (r,\ s')\ (p,\ t'))$
shows *rel-spec f g s t R*
⟨*proof*⟩

lemma *runs-to-res-independent-res*: $f \cdot s\ \{\!|\lambda\text{-}.\ P|\!\} \longleftrightarrow f \cdot s\ \{\!|\lambda \textit{Res}\ \text{-}.\ P|\!\}$
⟨*proof*⟩

lemma *lift-state-spec-monad-eq*[*simp*]: *lift-state* $(=)\ p = p$
⟨*proof*⟩

lemma *rel-post-state-Sup*:
rel-set $(\lambda x\ y.$ *rel-post-state* $Q\ (f\ x)\ (g\ y))\ X\ Y \Longrightarrow$ *rel-post-state* $Q\ (\bigsqcup x{\in}X.\ f\ x)$ $(\bigsqcup x{\in}Y.\ g\ x)$
⟨*proof*⟩

lemma *rel-set-Result-image-iff*:
rel-set (*rel-prod* $(\lambda \textit{Res}\ v1\ \textit{Res}\ v2.\ (P\ v1\ v2))\ R$)
$((\lambda x.\ \textit{case}\ x\ \textit{of}\ (v,\ s) \Rightarrow (\textit{Result}\ v,\ s))\ `\ \textit{Vals1})$
$((\lambda x.\ \textit{case}\ x\ \textit{of}\ (v,\ s) \Rightarrow (\textit{Result}\ v,\ s))\ `\ \textit{Vals2})$
$\longleftrightarrow$
rel-set (*rel-prod P R*) *Vals1 Vals2*
⟨*proof*⟩

lemma *rel-post-state-converse-iff*:
rel-post-state $R\ X\ Y \longleftrightarrow$ *rel-post-state* $R^{-1-1}\ Y\ X$
⟨*proof*⟩

lemma *runs-to-le-post-state-iff*: *runs-to f s Q* $\longleftrightarrow$ *run f s* $\leq$ *Success* {(*r*, *s*). *Q r s*}
⟨*proof*⟩

lemma *runs-to-partial-runs-to-iff*: *runs-to-partial f s Q* $\longleftrightarrow$ (*run f s* = *Failure* $\vee$ *runs-to f s Q*)
⟨*proof*⟩

lemma *run-runs-to-extensionality*:
run f s = *run g s* $\longleftrightarrow$ ($\forall P$. *f* · *s* ⦃*P*⦄ $\longleftrightarrow$ *g* · *s* ⦃*P*⦄)
⟨*proof*⟩

lemma *le-spec-monad-runI*: ($\bigwedge s$. *run f s* $\leq$ *run g s*) $\Longrightarrow$ *f* $\leq$ *g*
⟨*proof*⟩

16.6.1 *rel-spec-monad*

lemma *rel-spec-monad-iff-rel-spec*:
rel-spec-monad R Q f g $\longleftrightarrow$ ($\forall$ *s t*. *R s t* $\longrightarrow$ *rel-spec f g s t* (*rel-prod Q R*))
⟨*proof*⟩

lemma *rel-spec-monadI*:
($\bigwedge$*s t*. *R s t* $\Longrightarrow$ *rel-spec f g s t* (*rel-prod Q R*)) $\Longrightarrow$ *rel-spec-monad R Q f g*
⟨*proof*⟩

lemma *rel-spec-monadD*:
rel-spec-monad R Q f g $\Longrightarrow$ *R s t* $\Longrightarrow$ *rel-spec f g s t* (*rel-prod Q R*)
⟨*proof*⟩

lemma *rel-spec-monad-eq-conv*: *rel-spec-monad* (=) (=) = (=)
⟨*proof*⟩

lemma *rel-spec-monad-converse-iff*:
rel-spec-monad R Q f g $\longleftrightarrow$ *rel-spec-monad* R^{-1-1} Q^{-1-1} *g f*
⟨*proof*⟩

lemma *rel-spec-monad-iff-refines*:
rel-spec-monad S R f g $\longleftrightarrow$
($\forall$ *s t*. *S s t* $\longrightarrow$ (*refines f g s t* (*rel-prod R S*) $\wedge$ *refines g f t s* (*rel-prod* R^{-1-1} S^{-1-1})))
⟨*proof*⟩

lemma *rel-spec-monad-rel-exception-or-resultI*:
rel-spec-monad R (*rel-exception-or-result* (=) (=)) *f g* $\Longrightarrow$ *rel-spec-monad R* (=) *f g*
⟨*proof*⟩

lemma *runs-to-partial-runs-to-fuse*:
assumes *part*: *f* · *s* ?⦃*Q*⦄

assumes *tot*: $f \cdot s \ \{\!|P|\!\}$
shows $f \cdot s \ \{\!|\lambda r\ t.\ Q\ r\ t \wedge P\ r\ t|\!\}$
⟨*proof*⟩

16.7 VCG basic setup

lemma *runs-to-cong-pred-only*:
$P = Q \Longrightarrow (p \cdot s \ \{\!|\ P\ |\!\}) \longleftrightarrow (p \cdot s \ \{\!|\ Q\ |\!\})$
⟨*proof*⟩

lemma *runs-to-cong-state-only*[*runs-to-vcg-cong-state-only*]:
$s = t \Longrightarrow (p \cdot s \ \{\!|\ Q\ |\!\}) \longleftrightarrow (p \cdot t \ \{\!|\ Q\ |\!\})$
⟨*proof*⟩

lemma *runs-to-partial-cong-state-only*[*runs-to-vcg-cong-state-only*]:
$s = t \Longrightarrow (p \cdot s \ ?\{\!|\ Q\ |\!\}) \longleftrightarrow (p \cdot t \ ?\{\!|\ Q\ |\!\})$
⟨*proof*⟩

lemma *runs-to-cong-program-only*[*runs-to-vcg-cong-program-only*]:
$p = q \Longrightarrow (p \cdot s \ \{\!|\ Q\ |\!\}) \longleftrightarrow (q \cdot s \ \{\!|\ Q\ |\!\})$
⟨*proof*⟩

lemma *runs-to-partial-cong-program-only*[*runs-to-vcg-cong-program-only*]:
$p = q \Longrightarrow (p \cdot s \ ?\{\!|\ Q\ |\!\}) \longleftrightarrow (q \cdot s \ ?\{\!|\ Q\ |\!\})$
⟨*proof*⟩

lemma *runs-to-case-conv*[*simp*]:
$((case\ (a,\ b)\ of\ (x,\ y) \Rightarrow f\ x\ y) \cdot s \ \{\!|\ Q\ |\!\}) \longleftrightarrow ((f\ a\ b) \cdot s \ \{\!|\ Q\ |\!\})$
⟨*proof*⟩

lemma *runs-to-partial-case-conv*[*simp*]:
$((case\ (a,\ b)\ of\ (x,\ y) \Rightarrow f\ x\ y) \cdot s \ ?\{\!|\ Q\ |\!\}) \longleftrightarrow ((f\ a\ b) \cdot s \ ?\{\!|\ Q\ |\!\})$
⟨*proof*⟩

lemma *always-progress-prod-case*[*always-progress-intros*]:
$always\text{-}progress\ (f\ (fst\ p)\ (snd\ p)) \Longrightarrow always\text{-}progress\ (case\ p\ of\ (x,\ y) \Rightarrow f\ x\ y)$
⟨*proof*⟩

lemma *runs-to-conj*:
$(f \cdot s \ \{\!|\ \lambda r\ s.\ P\ r\ s \wedge Q\ r\ s|\!\}) \longleftrightarrow (f \cdot s \ \{\!|\ \lambda r\ s.\ P\ r\ s\ |\!\}) \wedge (f \cdot s \ \{\!|\ \lambda r\ s.\ Q\ r\ s|\!\})$
⟨*proof*⟩

lemma *runs-to-all*:
$(f \cdot s \ \{\!|\ \lambda r\ s.\ \forall x.\ P\ x\ r\ s\ |\!\}) \longleftrightarrow (\forall x.\ f \cdot s \ \{\!|\ \lambda r\ s.\ P\ x\ r\ s\ |\!\})$
⟨*proof*⟩

lemma *runs-to-imp-const*:
$(f \cdot s \ \{\!|\ \lambda r\ s.\ P\ r\ s \longrightarrow Q\ |\!\}) \longleftrightarrow (Q \wedge (f \cdot s \ \{\!|\lambda\text{-}\ \text{-}.\ True\ |\!\})) \vee (\neg\ Q \wedge (f \cdot s \ \{\!|\ \lambda r\ s.\ \neg\ P\ r\ s|\!\}))$

⟨*proof*⟩

16.7.1 *res-monad* and *exn-monad* Types

type-synonym *'a val* = (*unit*, *'a*) *exception-or-result*
type-synonym (*'e*, *'a*) *xval* = (*'e option*, *'a*) *exception-or-result*
type-synonym (*'a*, *'s*) *res-monad* = (*unit*, *'a* , *'s*) *spec-monad*
type-synonym (*'e*, *'a*, *'s*) *exn-monad* = (*'e option*, *'a*, *'s*) *spec-monad*

definition *Exn* :: *'e* ⇒ (*'e*, *'a*) *xval* **where**
Exn e = *Exception* (*Some e*)

definition
case-xval :: (*'e* ⇒ *'a*) ⇒ (*'v* ⇒ *'a*) ⇒ (*'e*, *'v*) *xval* ⇒ *'a* **where**
case-xval f g x =
(*case x of*
Exception v ⇒ (*case v of Some e* ⇒ *f e* | *None* ⇒ *undefined*)
| *Result v* ⇒ *g v*)

declare [[*case-translation case-xval Exn Result*]]

inductive *rel-xval*:: (*'e* ⇒ *'f* ⇒ *bool*) ⇒ (*'a* ⇒ *'b* ⇒ *bool*) ⇒
(*'e*, *'a*) *xval* ⇒ (*'f*, *'b*) *xval* ⇒ *bool*
where
Exn: *E e f* ⟹ *rel-xval E R* (*Exn e*) (*Exn f*) |
Result: *R a b* ⟹ *rel-xval E R* (*Result a*) (*Result b*)

definition *map-xval* :: (*'e* ⇒ *'f*) ⇒ (*'a* ⇒ *'b*) ⇒ (*'e*, *'a*) *xval* ⇒ (*'f*, *'b*) *xval* **where**
map-xval f g x ≡ *case x of Exn e* ⇒ *Exn* (*f e*) | *Result v* ⇒ *Result* (*g v*)

lemma *rel-xval-eq*: *rel-xval* (=) (=) = (=)
⟨*proof*⟩

lemma *rel-xval-rel-exception-or-result-conv*:
rel-xval E R = *rel-exception-or-result* (*rel-map the OO E OO rel-map Some*) *R*
⟨*proof*⟩

lemmas *rel-xval-Exn* = *rel-xval.intros*(*1*)
lemmas *rel-xval-Result* = *rel-xval.intros*(*2*)

lemma *case-xval-simps*[*simp*]:
case-xval f g (*Exn v*) = *f v*
case-xval f g (*Result e*) = *g e*
⟨*proof*⟩

lemma *case-exception-or-result-Exn*[*simp*]:
case-exception-or-result f g (*Exn x*) = *f* (*Some x*)
⟨*proof*⟩

lemma *xval-split*:
P (*case-xval f g r*) ⟷
(∀ *e*. *r* = *Exn e* ⟶ *P* (*f e*)) ∧
(∀ *v*. *r* = *Result v* ⟶ *P* (*g v*))
⟨*proof*⟩

lemma *xval-split-asm*:
P (*case-xval f g r*) ⟷
¬ ((∃ *e*. *r* = *Exn e* ∧ ¬ *P* (*f e*)) ∨
(∃ *v*. *r* = *Result v* ∧ ¬ *P* (*g v*)))
⟨*proof*⟩

lemmas *xval-splits* = *xval-split xval-split-asm*

lemma *Exn-eq-Exn*[*simp*]: *Exn x* = *Exn y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *Exn-neq-Result*[*simp*]: *Exn x* = *Result e* ⟷ *False*
⟨*proof*⟩

lemma *Result-neq-Exn*[*simp*]: *Result e* = *Exn x* ⟷ *False*
⟨*proof*⟩

lemma *Exn-eq-Exception*[*simp*]:
Exn x = *Exception a* ⟷ *a* = *Some x*
Exception a = *Exn x* ⟷ *a* = *Some x*
⟨*proof*⟩

lemma *map-exception-or-result-Exn*[*simp*]:
(⋀*x*. *f* (*Some x*) ≠ *None*) ⟹ *map-exception-or-result f g* (*Exn x*) = *Exn* (*the* (*f* (*Some x*)))
⟨*proof*⟩

lemma *case-xval-Exception-Some-simp*[*simp*]: (*case Exception* (*Some y*) *of Exn e* ⇒ *f e* | *Result v* ⇒ *g v*) = *f y*
⟨*proof*⟩

lemma *rel-xval-simps*[*simp*]:
rel-xval E R (*Exn e*) (*Exn f*) = *E e f*
rel-xval E R (*Result v*) (*Result w*) = *R v w*
rel-xval E R (*Exn e*) (*Result w*) = *False*
rel-xval E R (*Result v*) (*Exn f*) = *False*
⟨*proof*⟩

lemma *map-xval-simps*[*simp*]:
map-xval f g (*Exn e*) = *Exn* (*f e*)
map-xval f g (*Result v*) = *Result* (*g v*)
⟨*proof*⟩

lemma *map-xval-Exn*: *map-xval f g x = Exn y* $\longleftrightarrow$ ($\exists$ *e. x = Exn e* $\wedge$ *y = f e*)
⟨*proof*⟩

lemma *map-xval-Result*: *map-xval f g x = Result y* $\longleftrightarrow$ ($\exists$ *v. x = Result v* $\wedge$ *y = g v*)
⟨*proof*⟩

lemma *Result-unit-eq*: (*x*:: *unit val*) = *Result* ()
⟨*proof*⟩

⟨*ML*⟩

lemma *ex-val-Result1*:
$\exists$ *v1*. (*x*::*'v1 val*) = *Result v1*
⟨*proof*⟩

lemma *ex-val-Result2*:
$\exists$ *v1 v2*. (*x*::(*'v1* $*$ *'v2*) *val*) = *Result* (*v1*, *v2*)
⟨*proof*⟩

lemma *ex-val-Result3*:
$\exists$ *v1 v2 v3*. (*x*::(*'v1* $*$ *'v2* $*$ *'v3*) *val*) = *Result* (*v1*, *v2*, *v3*)
⟨*proof*⟩

lemma *ex-val-Result4*:
$\exists$ *v1 v2 v3 v4*. (*x*::(*'v1* $*$ *'v2* $*$ *'v3* $*$ *'v4*) *val*) = *Result* (*v1*, *v2*, *v3*, *v4*)
⟨*proof*⟩

lemma *ex-val-Result5*:
$\exists$ *v1 v2 v3 v4 v5*. (*x*::(*'v1* $*$ *'v2* $*$ *'v3* $*$ *'v4* $*$ *'v5*) *val*) = *Result* (*v1*, *v2*, *v3*, *v4*, *v5*)
⟨*proof*⟩

lemma *ex-val-Result6*:
$\exists$ *v1 v2 v3 v4 v5 v6*. (*x*::(*'v1* $*$ *'v2* $*$ *'v3* $*$ *'v4* $*$ *'v5* $*$ *'v6*) *val*) = *Result* (*v1*, *v2*, *v3*, *v4*, *v5*, *v6*)
⟨*proof*⟩

lemma *ex-val-Result7*:
$\exists$ *v1 v2 v3 v4 v5 v6 v7*. (*x*::(*'v1* $*$ *'v2* $*$ *'v3* $*$ *'v4* $*$ *'v5* $*$ *'v6* $*$ *'v7*) *val*) = *Result* (*v1*, *v2*, *v3*, *v4*, *v5*, *v6*, *v7*)
⟨*proof*⟩

lemma *ex-val-Result8*:
$\exists$ *v1 v2 v3 v4 v5 v6 v7 v8*. (*x*::(*'v1* $*$ *'v2* $*$ *'v3* $*$ *'v4* $*$ *'v5* $*$ *'v6* $*$ *'v7* $*$ *'v8*) *val*) = *Result* (*v1*, *v2*, *v3*, *v4*, *v5*, *v6*, *v7*, *v8*)
⟨*proof*⟩

lemma *ex-val-Result9*:

$\exists\, v1\ v2\ v3\ v4\ v5\ v6\ v7\ v8\ v9.\ (x::('v1 * 'v2 * 'v3 * 'v4 * 'v5 * 'v6 * 'v7 * 'v8 * 'v9)\ val) = Result\ (v1,\ v2,\ v3,\ v4,\ v5,\ v6,\ v7,\ v8,\ v9)$
⟨*proof*⟩

lemma *ex-val-Result10*:
$\exists\, v1\ v2\ v3\ v4\ v5\ v6\ v7\ v8\ v9\ v10.\ (x::('v1 * 'v2 * 'v3 * 'v4 * 'v5 * 'v6 * 'v7 * 'v8 * 'v9 * 'v10)\ val) = Result\ (v1,\ v2,\ v3,\ v4,\ v5,\ v6,\ v7,\ v8,\ v9,\ v10)$
⟨*proof*⟩

lemma *ex-val-Result11*:
$\exists\, v1\ v2\ v3\ v4\ v5\ v6\ v7\ v8\ v9\ v10\ v11.\ (x::('v1 * 'v2 * 'v3 * 'v4 * 'v5 * 'v6 * 'v7 * 'v8 * 'v9 * 'v10 * 'v11)\ val) = Result\ (v1,\ v2,\ v3,\ v4,\ v5,\ v6,\ v7,\ v8,\ v9,\ v10,\ v11)$
⟨*proof*⟩

lemmas *ex-val-Result*[*simp*] =
ex-val-Result1
ex-val-Result2
ex-val-Result3
ex-val-Result4
ex-val-Result5
ex-val-Result6
ex-val-Result7
ex-val-Result8
ex-val-Result9
ex-val-Result10
ex-val-Result11

lemma *all-val-imp-iff*: $(\forall v.\ (r::'a\ val) = Result\ v \longrightarrow P) \longleftrightarrow P$
⟨*proof*⟩

definition *map-exn* :: $('e \Rightarrow 'f) \Rightarrow ('e,\ 'a)\ xval \Rightarrow ('f,\ 'a)\ xval$ **where**
map-exn f x =
(*case x of*
Exn e ⇒ *Exn* (*f e*)
| *Result v* ⇒ *Result v*)

lemma *map-exn-simps*[*simp*]:
map-exn f (*Exn e*) = *Exn* (*f e*)
map-exn f (*Result v*) = *Result v*
map-exn f x = *Result v* ⟷ *x* = *Result v*
⟨*proof*⟩

lemma *map-exn-id*[*simp*]: *map-exn* (λ*x*. *x*) = (λ*x*. *x*)
⟨*proof*⟩

definition *unnest-exn* :: $('e + 'a,\ 'a)\ xval \Rightarrow ('e,\ 'a)\ xval$ **where**

unnest-exn x =
(case x of
Exn e ⇒ (case e of Inl l ⇒ Exn l | Inr r ⇒ Result r)
| Result v ⇒ Result v)

lemma *unnest-exn-simps*[*simp*]:
unnest-exn (Exn (Inl l)) = Exn l
unnest-exn (Exn (Inr r)) = Result r
unnest-exn (Result v) = Result v
⟨*proof*⟩

lemma *unnest-exn-eq-simps*[*simp*]:
unnest-exn (Result r) = Result r′ ⟷ r = r′
unnest-exn (Result e) ≠ Exn e
unnest-exn (Exn e) = Result r ⟷ e = Inr r
unnest-exn (Exn e) = Exn e′ ⟷ e = Inl e′
⟨*proof*⟩

definition *the-Exn* :: (′*e*, ′*a*) *xval* ⇒ ′*e* **where**
the-Exn x ≡ (*case x of Exn e ⇒ e | Result - ⇒ undefined*)

abbreviation *the-Res* :: ′*a val* ⇒ ′*a* **where** *the-Res* ≡ *the-Result*

lemma *is-Exception-val*[*simp*]: *is-Exception* (*x*::′*a val*) = *False*
⟨*proof*⟩

lemma *is-Exception-Exn*[*simp*]: *is-Exception* (*Exn x*)
⟨*proof*⟩

lemma *is-Result-val*[*simp*]: *is-Result* (*v*::′*a val*)
⟨*proof*⟩

lemma *the-Exception-Exn*[*simp*]: *the-Exception* (*Exn e*) = *Some e*
⟨*proof*⟩

lemma *the-Exn-Exn*[*simp*]:
the-Exn (Exn e) = e
the-Exn (Exception (Some e)) = e
⟨*proof*⟩

lemma *the-Result-simp*[*simp*]:
the-Result (Result v) = v
⟨*proof*⟩

lemma *Result-the-Result-val*[*simp*]: *Result* (*the-Result* (*x*::′*a val*)) = *x*
⟨*proof*⟩

lemma *rel-exception-or-result-val-apply*:
fixes *x*::′*a val*

fixes y ::$'b$ *val*
shows *rel-exception-or-result E R x y* $\longleftrightarrow$ *rel-exception-or-result E' R x y*
⟨*proof*⟩

lemma *rel-exception-or-result-val*:
shows ((*rel-exception-or-result E R*)::$'a$ *val* $\Rightarrow$ $'b$ *val* $\Rightarrow$ *bool*) = *rel-exception-or-result E' R*
⟨*proof*⟩

lemma *rel-exception-or-result-Res-val*:
(λ*Res x Res y*. *R x y*) = *rel-exception-or-result* (λ- -. *False*) *R*
⟨*proof*⟩

definition *unite*:: ($'a$, $'a$) *xval* $\Rightarrow$ $'a$ *val* **where**
unite x = (*case x of Exn v* $\Rightarrow$ *Result v* | *Result v* $\Rightarrow$ *Result v*)

lemma *unite-simps*[*simp*]:
unite (*Exn v*) = *Result v*
unite (*Result v*) = *Result v*
⟨*proof*⟩

lemma *val-exhaust*: (⋀*v*. (*x*::$'a$ *val*) = *Result v* $\Longrightarrow$ *P*) $\Longrightarrow$ *P*
⟨*proof*⟩

lemma *val-iff*: *P* (*x*::$'a$ *val*) $\longleftrightarrow$ (∃ *v*. *x* = *Result v* ∧ *P* (*Result v*))
⟨*proof*⟩

lemma *val-nchotomy*: ∀ (*x*::$'a$ *val*). ∃ *v*. *x* = *Result v*
⟨*proof*⟩

lemma (⋀*x*::$'a$ *val*. *PROP P x*) $\Longrightarrow$ *PROP P* (*Result* (*v*::$'a$))
⟨*proof*⟩

lemma *val-Result-the-Result-conv*: (*x*::$'a$ *val*) = *Result* (*the-Result x*)
⟨*proof*⟩

lemma *split-val-all*[*simp*]: (⋀*x*::$'a$ *val*. *PROP P x*) ≡ (⋀*v*::$'a$. *PROP P* (*Result v*))
⟨*proof*⟩

lemma *split-val-Ex*[*simp*]: (∃ (*x*::$'a$ *val*). *P x*) $\longleftrightarrow$ (∃ (*v*::$'a$). *P* (*Result v*))
⟨*proof*⟩

lemma *split-val-All*[*simp*]: (∀ (*x*::$'a$ *val*). *P x*) $\longleftrightarrow$ (∀ (*v*::$'a$). *P* (*Result v*))
⟨*proof*⟩

definition *to-xval* :: ($'e$ + $'a$) $\Rightarrow$ ($'e$, $'a$) *xval* **where**
to-xval x = (*case x of Inl e* $\Rightarrow$ *Exn e* | *Inr v* $\Rightarrow$ *Result v*)

lemma *to-xval-simps*[*simp*]:
to-xval (*Inl e*) = *Exn e*
to-xval (*Inr v*) = *Result v*
⟨*proof*⟩

lemma *to-xval-Result-iff*[*simp*]: *to-xval x* = *Result v* ⟷ *x* = *Inr v*
⟨*proof*⟩

lemma *to-xval-Exn-iff*[*simp*]:
to-xval x = *Exn v* ⟷
(*x* = *Inl v*)
⟨*proof*⟩

lemma *to-xval-Exception-iff*[*simp*]:
to-xval x = *Exception v* ⟷
((*v* = *None* ∧ *x* = *Inr undefined*) ∨ (∃ *e*. *v* = *Some e* ∧ *x* = *Inl e*))
⟨*proof*⟩

definition *from-xval* :: ('*e*, '*a*) *xval* ⇒ ('*e* + '*a*) **where**
from-xval x = (*case x of Exn e* ⇒ *Inl e* | *Result v* ⇒ *Inr v*)

lemma *from-xval-simps*[*simp*]:
from-xval (*Exn e*) = *Inl e*
from-xval (*Result v*) = *Inr v*
⟨*proof*⟩

lemma *from-xval-Inr-iff*[*simp*]: *from-xval x* = *Inr v* ⟷ *x* = *Result v*
⟨*proof*⟩

lemma *from-xval-Inl-iff*[*simp*]: *from-xval x* = *Inl e* ⟷ *x* = *Exn e*
⟨*proof*⟩

lemma *to-xval-from-xval*[*simp*]: *to-xval* (*from-xval x*) = *x*
⟨*proof*⟩

lemma *from-xval-to-xval*[*simp*]: *from-xval* (*to-xval x*) = *x*
⟨*proof*⟩

lemma *rel-map-to-xval-Exn-iff*[*simp*]: *rel-map to-xval y* (*Exn e*) ⟷ *y* = *Inl e*
⟨*proof*⟩

lemma *rel-map-to-xval-Inr-iff*[*simp*]: *rel-map to-xval* (*Inr r*) *x* ⟷ *x* = *Result r*
⟨*proof*⟩

lemma *rel-map-to-xval-Inl-iff*[*simp*]: *rel-map to-xval* (*Inl l*) *x* ⟷ *x* = *Exn l*
⟨*proof*⟩

lemma *rel-map-to-xval-Result-iff*[*simp*]: *rel-map to-xval y* (*Result r*) ⟷ *y* = *Inr*

r
⟨*proof*⟩

lemma *rel-map-from-xval-Exn-iff*[*simp*]: *rel-map from-xval* (*Exn l*) *x* ⟷ *x* = *Inl l*
⟨*proof*⟩

lemma *rel-map-from-xval-Inl-iff*[*simp*]: *rel-map from-xval y* (*Inl e*) ⟷ *y* = *Exn e*
⟨*proof*⟩

lemma *rel-map-from-xval-Result-iff*[*simp*]: *rel-map from-xval* (*Result r*) *x* ⟷ *x* = *Inr r*
⟨*proof*⟩

lemma *rel-map-from-xval-Inr-iff*[*simp*]: *rel-map from-xval y* (*Inr r*) ⟷ *y* = *Result r*
⟨*proof*⟩

16.8 *res-monad* and *exn-monad* functions

abbreviation *throw e* ≡ *yield* (*Exn e*)

definition *try* :: ('*e* + '*a*, '*a*, '*s*) *exn-monad* ⇒ ('*e*, '*a*, '*s*) *exn-monad* **where**
try = *map-value unnest-exn*

definition *finally* :: ('*a*, '*a*, '*s*) *exn-monad* ⇒ ('*a*, '*s*) *res-monad* **where**
finally = *map-value unite*

definition
catch :: ('*e*, '*a*, '*s*) *exn-monad* ⇒
('*e* ⇒ ('*f*::*default*, '*a*, '*s*) *spec-monad*) ⇒
('*f*::*default*, '*a*, '*s*) *spec-monad* (**infix** ‹<*catch*>› *10*)
where
f <*catch*> *handler* ≡ *bind-handle f return* (*handler o the*)

abbreviation *bind-finally* ::
('*e*, '*a*, '*s*) *exn-monad* ⇒ (('*e*, '*a*) *xval* ⇒ ('*b*, '*s*) *res-monad*) ⇒ ('*b*, '*s*) *res-monad*
where
bind-finally ≡ *bind-exception-or-result*

definition *ignoreE* ::('*e*, '*a*, '*s*) *exn-monad* ⇒ ('*a*, '*s*) *res-monad* **where**
ignoreE f = *catch f* (λ-. *bot*)

definition *liftE* :: ('*a*, '*s*) *res-monad* ⇒ ('*e*, '*a*, '*s*) *exn-monad* **where**
liftE = *map-value* (*map-exception-or-result* (λ*x*. *undefined*) *id*)

definition *check*:: '*e* ⇒ ('*s* ⇒ '*a* ⇒ *bool*) ⇒ ('*e*, '*a*, '*s*) *exn-monad* **where**
check e p =

condition ($\lambda s.\ \exists x.\ p\ s\ x$)
(*do* { $s \leftarrow$ *get-state*; *select* $\{x.\ p\ s\ x\}$ })
(*throw* e)

abbreviation *check'* :: $'e \Rightarrow ('s \Rightarrow bool) \Rightarrow ('e, unit, 's)$ *exn-monad* **where**
check' $e\ p \equiv$ *check* e (λs -. $p\ s$)

16.9 Monad operations

16.9.1 $\top$

declare *top-spec-monad.rep-eq*[*run-spec-monad*, *simp*]

lemma *always-progress-top*[*always-progress-intros*]: *always-progress* $\top$
⟨*proof*⟩

lemma *runs-to-top*[*simp*]: $\top \cdot s \{\!|\ Q\ |\!\} \longleftrightarrow$ *False*
⟨*proof*⟩

lemma *runs-to-partial-top*[*simp*]: $\top \cdot s$ *?*$\{\!|\ Q\ |\!\} \longleftrightarrow$ *True*
⟨*proof*⟩

16.9.2 $\bot$

declare *bot-spec-monad.rep-eq*[*run-spec-monad*, *simp*]

lemma *always-progress-bot-iff*[*iff*]: *always-progress* $\bot \longleftrightarrow$ *False*
⟨*proof*⟩

lemma *runs-to-bot*[*simp*]: $\bot \cdot s \{\!|\ Q\ |\!\} \longleftrightarrow$ *True*
⟨*proof*⟩

lemma *runs-to-partial-bot*[*simp*]: $\bot \cdot s$ *?*$\{\!|\ Q\ |\!\} \longleftrightarrow$ *True*
⟨*proof*⟩

16.9.3 *fail*

lemma *always-progress-fail*[*always-progress-intros*]: *always-progress fail*
⟨*proof*⟩

lemma *run-fail*[*run-spec-monad*, *simp*]: *run fail* $s = \top$
⟨*proof*⟩

lemma *runs-to-fail*[*simp*]: *fail* $\cdot s \{\!|\ R\ |\!\} \longleftrightarrow$ *False*
⟨*proof*⟩

lemma *runs-to-partial-fail*[*simp*]: *fail* $\cdot s$ *?*$\{\!|\ R\ |\!\} \longleftrightarrow$ *True*
⟨*proof*⟩

lemma *refines-fail*[*simp*]: *refines f fail s t R*
⟨*proof*⟩

lemma *rel-spec-fail*[*simp*]: *rel-spec fail fail s t R*
⟨*proof*⟩

16.9.4 *yield*

lemma *run-yield*[*run-spec-monad*, *simp*]: *run* (*yield r*) *s* = *pure-post-state* (*r*, *s*)
⟨*proof*⟩

lemma *always-progress-yield*[*always-progress-intros*]: *always-progress* (*yield r*)
⟨*proof*⟩

lemma *yield-inj*[*simp*]: *yield x* = *yield y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *runs-to-yield-iff*[*simp*]: ((*yield r*) · *s* ⦃ *Q* ⦄) ⟷ *Q r s*
⟨*proof*⟩

lemma *runs-to-yield*[*runs-to-vcg*]: *Q r s* ⟹ *yield r* · *s* ⦃ *Q* ⦄
⟨*proof*⟩

lemma *runs-to-partial-yield-iff*[*simp*]: ((*yield r*) · *s* ?⦃ *Q* ⦄) ⟷ *Q r s*
⟨*proof*⟩

lemma *runs-to-partial-yield*[*runs-to-vcg*]: *Q r s* ⟹ *yield r* · *s* ?⦃ *Q* ⦄
⟨*proof*⟩

lemma *refines-yield-iff*[*simp*]: *refines* (*yield r*) (*yield r′*) *s s′ R* ⟷ *R* (*r*, *s*) (*r′*, *s′*)
⟨*proof*⟩

lemma *refines-yield*: *R* (*a*, *s*) (*b*, *t*) ⟹ *refines* (*yield a*) (*yield b*) *s t R*
⟨*proof*⟩

lemma *refines-yield-right-iff*:
refines f (*yield e*) *s t R* ⟷ (*f* · *s* ⦃ λ*r s′*. *R* (*r*, *s′*) (*e*, *t*) ⦄)
⟨*proof*⟩

lemma *rel-spec-yield*: *R* (*a*, *s*) (*b*, *t*) ⟹ *rel-spec* (*yield a*) (*yield b*) *s t R*
⟨*proof*⟩

lemma *rel-spec-yield-iff*[*simp*]: *rel-spec* (*yield x*) (*yield y*) *s t Q* ⟷ *Q* (*x*, *s*) (*y*, *t*)
⟨*proof*⟩

lemma *rel-spec-monad-yield*: *Q x y* ⟹ *rel-spec-monad R Q* (*yield x*) (*yield y*)
⟨*proof*⟩

16.9.5 *throw-exception-or-result*

lemma *throw-exception-or-result-bind*[*simp*]:
$e \neq default \Longrightarrow$ *throw-exception-or-result* $e >>= f =$ *throw-exception-or-result* e
⟨*proof*⟩

16.9.6 *throw*

lemma *throw-bind*[*simp*]: *throw* $e >>= f =$ *throw* e
⟨*proof*⟩

16.9.7 *get-state*

lemma *always-progress-get-state*[*always-progress-intros*]: *always-progress get-state*
⟨*proof*⟩

lemma *run-get-state*[*run-spec-monad*, *simp*]: *run get-state* $s =$ *pure-post-state* (*Result* s, s)
⟨*proof*⟩

lemma *runs-to-get-state*[*runs-to-vcg*]: *get-state* $\cdot$ s $\{\!|\lambda r\ t.\ r = Result\ s \wedge t = s|\!\}$
⟨*proof*⟩

lemma *runs-to-get-state-iff*[*runs-to-iff*]: *get-state* $\cdot$ s $\{\!|Q|\!\}$ $\longleftrightarrow$ (Q (*Result* s) s)
⟨*proof*⟩

lemma *runs-to-partial-get-state*[*runs-to-vcg*]: *get-state* $\cdot$ s $?\{\!|\lambda r\ t.\ r = Result\ s \wedge t = s|\!\}$
⟨*proof*⟩

lemma *refines-get-state*: R (*Result* s, s) (*Result* t, t) $\Longrightarrow$ *refines get-state get-state* s t R
⟨*proof*⟩

lemma *rel-spec-get-state*: R (*Result* s, s) (*Result* t, t) $\Longrightarrow$ *rel-spec get-state get-state* s t R
⟨*proof*⟩

16.9.8 *set-state*

lemma *always-progress-set-state*[*always-progress-intros*]: *always-progress* (*set-state* t)
⟨*proof*⟩

lemma *set-state-inj*[*simp*]: *set-state* $x =$ *set-state* $y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *run-set-state*[*run-spec-monad*, *simp*]: *run* (*set-state* t) $s =$ *pure-post-state* (*Result* (), t)
⟨*proof*⟩

lemma *runs-to-set-state*[*runs-to-vcg*]: *set-state t · s* {⦇λ*r t*′. *t*′ = *t*⦈}
⟨*proof*⟩

lemma *runs-to-set-state-iff*[*runs-to-iff*]: *set-state t · s* {⦇*Q*⦈} ⟷ *Q* (*Result* ()) *t*
⟨*proof*⟩

lemma *runs-to-partial-set-state*[*runs-to-vcg*]: *set-state t · s ?*{⦇λ*r t*′. *t*′ = *t*⦈}
⟨*proof*⟩

lemma *refines-set-state*:
R (*Result* (), *s*′) (*Result* (), *t*′) ⟹ *refines* (*set-state s*′) (*set-state t*′) *s t R*
⟨*proof*⟩

lemma *rel-spec-set-state*:
R (*Result* (), *s*′) (*Result* (), *t*′) ⟹ *rel-spec* (*set-state s*′) (*set-state t*′) *s t R*
⟨*proof*⟩

16.9.9 *select*

lemma *always-progress-select*[*always-progress-intros*]: *S* ≠ {} ⟹ *always-progress* (*select S*)
⟨*proof*⟩

lemma *runs-to-select*[*runs-to-vcg*]: (⋀*x*. *x* ∈ *S* ⟹ *Q* (*Result x*) *s*) ⟹ *select S* · *s* {⦇ *Q* ⦈}
⟨*proof*⟩

lemma *runs-to-select-iff*[*runs-to-iff*]: *select S* · *s* {⦇*Q*⦈} ⟷ (∀ *x*∈*S*. *Q* (*Result x*) *s*)
⟨*proof*⟩

lemma *runs-to-partial-select*[*runs-to-vcg*]: (⋀*x*. *x* ∈ *S* ⟹ *Q* (*Result x*) *s*) ⟹ *select S* · *s ?*{⦇ *Q* ⦈}
⟨*proof*⟩

lemma *run-select*[*run-spec-monad*, *simp*]: *run* (*select S*) *s* = *Success* ((λ*v*. (*Result v*, *s*)) ‘ *S*)
⟨*proof*⟩

lemma *refines-select*:
(⋀*x*. *x* ∈ *P* ⟹ ∃ *xa*∈*Q*. *R* (*Result x*, *s*) (*Result xa*, *t*)) ⟹ *refines* (*select P*) (*select Q*) *s t R*
⟨*proof*⟩

lemma *rel-spec-select*:
rel-set (λ*a b*. *R* (*Result a*, *s*) (*Result b*, *t*)) *P Q* ⟹ *rel-spec* (*select P*) (*select Q*) *s t R*
⟨*proof*⟩

16.9.10 *unknown*

lemma *runs-to-unknown*[*runs-to-vcg*]: $(\bigwedge x.\ Q\ (Result\ x)\ s) \Longrightarrow unknown \cdot s \{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *runs-to-unknown-iff*[*runs-to-iff*]: $unknown \cdot s \{\!|\ Q\ |\!\} \longleftrightarrow (\forall x.\ Q\ (Result\ x)\ s)$
⟨*proof*⟩

lemma *runs-to-partial-unknown*[*runs-to-vcg*]: $(\bigwedge x.\ Q\ (Result\ x)\ s) \Longrightarrow unknown \cdot s\ ?\{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *run-unknown*[*run-spec-monad*, *simp*]: $run\ unknown\ s = Success\ ((\lambda v.\ (Result\ v,\ s))\ `\ UNIV)$
⟨*proof*⟩

lemma *always-progress-unknown*[*always-progress-intros*]: *always-progress unknown*
⟨*proof*⟩

16.9.11 *lift-state*

lemma *run-lift-state*[*run-spec-monad*]:
$run\ (lift\text{-}state\ R\ f)\ s = lift\text{-}post\text{-}state\ (rel\text{-}prod\ (=)\ R)\ (SUP\ t{\in}\{t.\ R\ s\ t\}.\ run\ f\ t)$
⟨*proof*⟩

lemma *runs-to-lift-state-iff*[*runs-to-iff*]:
$(lift\text{-}state\ R\ f) \cdot s \{\!|\ Q\ |\!\} \longleftrightarrow (\forall s'.\ R\ s\ s' \longrightarrow f \cdot s' \{\!|\lambda r\ t'.\ \forall t.\ R\ t\ t' \longrightarrow Q\ r\ t|\!\})$
⟨*proof*⟩

lemma *runs-to-lift-state*[*runs-to-vcg*]:
$(\bigwedge s'.\ R\ s\ s' \Longrightarrow f \cdot s' \{\!|\lambda r\ t'.\ \forall t.\ R\ t\ t' \longrightarrow Q\ r\ t|\!\}) \Longrightarrow lift\text{-}state\ R\ f \cdot s \{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *runs-to-partial-lift-state*[*runs-to-vcg*]:
$(\bigwedge s'.\ R\ s\ s' \Longrightarrow f \cdot s'\ ?\{\!|\lambda r\ t'.\ \ \forall t.\ R\ t\ t' \longrightarrow Q\ r\ t|\!\}) \Longrightarrow lift\text{-}state\ R\ f \cdot s\ ?\{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *mono-lift-state*: $f \le f' \Longrightarrow lift\text{-}state\ R\ f \le lift\text{-}state\ R\ f'$
⟨*proof*⟩

16.9.12 **const***exec-concrete*

lemma *run-exec-concrete*[*run-spec-monad*]:
$run\ (exec\text{-}concrete\ st\ f)\ s = map\text{-}post\text{-}state\ (apsnd\ st)\ (\bigsqcup\ (run\ f\ `\ st\ -`\ \{s\}))$
⟨*proof*⟩

lemma *runs-to-exec-concrete-iff*[*runs-to-iff*]:
$exec\text{-}concrete\ st\ f \cdot s \{\!|\ Q\ |\!\} \longleftrightarrow (\forall t.\ s = st\ t \longrightarrow f \cdot t \{\!|\lambda r\ t.\ \ Q\ r\ (st\ t)|\!\})$
⟨*proof*⟩

lemma *runs-to-exec-concrete*[*runs-to-vcg*]:
$(\bigwedge t.\ s = st\ t \Longrightarrow f \cdot t \{\!|\lambda r\ t.\ \ Q\ r\ (st\ t)|\!\}) \Longrightarrow exec\text{-}concrete\ st\ f \cdot s \{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *runs-to-partial-exec-concrete*[*runs-to-vcg*]:
$(\bigwedge t.\ s = st\ t \Longrightarrow f \cdot t\ ?\{\!|\lambda r\ t.\ \ Q\ r\ (st\ t)|\!\}) \Longrightarrow exec\text{-}concrete\ st\ f \cdot s\ ?\{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *mono-exec-concrete*: $f \leq f' \Longrightarrow exec\text{-}concrete\ st\ f \leq exec\text{-}concrete\ st\ f'$
⟨*proof*⟩

lemma *monotone-exec-concrete-le*[*partial-function-mono*]:
$monotone\ Q\ (\leq)\ f \Longrightarrow monotone\ Q\ (\leq)\ (\lambda f'.\ exec\text{-}concrete\ st\ (f\ f'))$
⟨*proof*⟩

lemma *monotone-exec-concrete-ge*[*partial-function-mono*]:
$monotone\ Q\ (\geq)\ f \Longrightarrow monotone\ Q\ (\geq)\ (\lambda f'.\ exec\text{-}concrete\ st\ (f\ f'))$
⟨*proof*⟩

16.9.13 const *exec-abstract*

lemma *run-exec-abstract*[*run-spec-monad*]:
$run\ (exec\text{-}abstract\ st\ f)\ s = vmap\text{-}post\text{-}state\ (apsnd\ st)\ (run\ f\ (st\ s))$
⟨*proof*⟩

lemma *runs-to-exec-abstract-iff*[*runs-to-iff*]:
$exec\text{-}abstract\ st\ f \cdot s \{\!|\ Q\ |\!\} \longleftrightarrow f \cdot (st\ s) \{\!|\lambda r\ t.\ \forall s'.\ t = st\ s' \longrightarrow Q\ r\ s'|\!\}$
⟨*proof*⟩

lemma *runs-to-exec-abstract*[*runs-to-vcg*]:
$f \cdot (st\ s) \{\!|\lambda r\ t.\ \forall s'.\ t = st\ s' \longrightarrow Q\ r\ s'|\!\} \Longrightarrow exec\text{-}abstract\ st\ f \cdot s \{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *runs-to-partial-exec-abstract*[*runs-to-vcg*]:
$f \cdot (st\ s)\ ?\{\!|\lambda r\ t.\ \forall s'.\ t = st\ s' \longrightarrow Q\ r\ s'|\!\} \Longrightarrow exec\text{-}abstract\ st\ f \cdot s\ ?\{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *mono-exec-abstract*: $f \leq f' \Longrightarrow exec\text{-}abstract\ st\ f \leq exec\text{-}abstract\ st\ f'$
⟨*proof*⟩

lemma *monotone-exec-abstract-le*[*partial-function-mono*]:
$monotone\ Q\ (\leq)\ f \Longrightarrow monotone\ Q\ (\leq)\ (\lambda f'.\ exec\text{-}abstract\ st\ (f\ f'))$
⟨*proof*⟩

lemma *monotone-exec-abstract-ge*[*partial-function-mono*]:

monotone Q (≥) f ⟹ monotone Q (≥) (λf′. exec-abstract st (f f′))
⟨*proof*⟩

16.9.14 *bind-exception-or-result*

lemma *runs-to-bind-exception-or-result-iff*[*runs-to-iff*]:
bind-exception-or-result f g · s ⦃ Q ⦄ ⟷ f · s ⦃ λr t. g r · t ⦃ Q ⦄⦄
⟨*proof*⟩

lemma *runs-to-bind-exception-or-result*[*runs-to-vcg*]:
f · s ⦃ λr t. g r · t ⦃ Q ⦄⦄ ⟹ bind-exception-or-result f g · s ⦃ Q ⦄
⟨*proof*⟩

lemma *runs-to-partial-bind-exception-or-result*[*runs-to-vcg*]:
f · s ?⦃ λr t. g r · t ?⦃ Q ⦄⦄ ⟹ bind-exception-or-result f g · s ?⦃ Q ⦄
⟨*proof*⟩

lemma *refines-bind-exception-or-result*:
refines f f′ s s′ (λ(r, t) (r′, t′). refines (g r) (g′ r′) t t′ R) ⟹
refines (bind-exception-or-result f g) (bind-exception-or-result f′ g′) s s′ R
⟨*proof*⟩

lemma *refines-bind-exception-or-result′*:
assumes *f*: *refines f f′ s s′ Q*
assumes *g*: *⋀r t r′ t′. Q (r, t) (r′, t′) ⟹ refines (g r) (g′ r′) t t′ R*
shows *refines (bind-exception-or-result f g) (bind-exception-or-result f′ g′) s s′ R*
⟨*proof*⟩

lemma *mono-bind-exception-or-result*:
f ≤ f′ ⟹ g ≤ g′ ⟹ bind-exception-or-result f g ≤ bind-exception-or-result f′ g′
⟨*proof*⟩

lemma *monotone-bind-exception-or-result-le*[*partial-function-mono*]:
monotone R (≤) (λf′. f f′) ⟹ (⋀v. monotone R (≤) (λf′. g f′ v)) ⟹
monotone R (≤) (λf′. bind-exception-or-result (f f′) (g f′))
⟨*proof*⟩

lemma *monotone-bind-exception-or-result-ge*[*partial-function-mono*]:
monotone R (≥) (λf′. f f′) ⟹ (⋀v. monotone R (≥) (λf′. g f′ v)) ⟹
monotone R (≥) (λf′. bind-exception-or-result (f f′) (g f′))
⟨*proof*⟩

lemma *run-bind-exception-or-result-cong*:
assumes *∗*: *run f s = run f′ s*
assumes *∗∗*: *f · s ?⦃ λx s′. run (g x) s′ = run (g′ x) s′ ⦄*
shows *run (bind-exception-or-result f g) s = run (bind-exception-or-result f′ g′) s*
⟨*proof*⟩

16.9.15 *bind-handle*

lemma *bind-handle-eq*:
bind-handle f g h =
bind-exception-or-result f (λ*r*. *case r of Exception e* ⇒ *h e* | *Result v* ⇒ *g v*)
⟨*proof*⟩

lemma *runs-to-bind-handle-iff*[*runs-to-iff*]:
bind-handle f g h · *s* ⦃ *Q* ⦄ ⟷ *f* · *s* ⦃ λ*r t*.
(∀ *v*. *r* = *Result v* ⟶ *g v* · *t* ⦃ *Q* ⦄) ∧
(∀ *e*. *r* = *Exception e* ⟶ *e* ≠ *default* ⟶ *h e* · *t* ⦃ *Q* ⦄)⦄
⟨*proof*⟩

lemma *runs-to-bind-handle*[*runs-to-vcg*]:
f · *s* ⦃ λ*r t*. (∀ *v*. *r* = *Result v* ⟶ *g v* · *t* ⦃ *Q* ⦄) ∧
(∀ *e*. *r* = *Exception e* ⟶ *e* ≠ *default* ⟶ *h e* · *t* ⦃ *Q* ⦄)⦄ ⟹
bind-handle f g h · *s* ⦃ *Q* ⦄
⟨*proof*⟩

lemma *runs-to-bind-handle-exeption-monad*[*runs-to-vcg*]:
fixes *f* :: ('*e*, '*a*, '*s*) *exn-monad*
assumes *f*:
f · *s* ⦃ λ*r t*. (∀ *v*. *r* = *Result v* ⟶ (*g v* · *t* ⦃ *Q* ⦄)) ∧
(∀ *e*. *r* = *Exception* (*Some e*) ⟶ (*h* (*Some e*) · *t* ⦃ *Q* ⦄))⦄
shows *bind-handle f g h* · *s* ⦃ *Q* ⦄
⟨*proof*⟩

lemma *runs-to-bind-handle-res-monad*[*runs-to-vcg*]:
fixes *f* :: ('*a*, '*s*) *res-monad*
assumes *f*: *f* · *s* ⦃ λ*r t*. (∀ *v*. *r* = *Result v* ⟶ (*g v* · *t* ⦃ *Q* ⦄))⦄
shows *bind-handle f g h* · *s* ⦃ *Q* ⦄
⟨*proof*⟩

lemma *runs-to-partial-bind-handle*[*runs-to-vcg*]:
f · *s* ?⦃ λ*r t*. (∀ *v*. *r* = *Result v* ⟶ *g v* · *t* ?⦃ *Q* ⦄) ∧
(∀ *e*. *r* = *Exception e* ⟶ *e* ≠ *default* ⟶ *h e* · *t* ?⦃ *Q* ⦄)⦄ ⟹
bind-handle f g h · *s* ?⦃ *Q* ⦄
⟨*proof*⟩

lemma *runs-to-partial-bind-handle-exeption-monad*[*runs-to-vcg*]:
fixes *f* :: ('*e*, '*a*, '*s*) *exn-monad*
assumes *f*: *f* · *s* ?⦃ λ*r t*. (∀ *v*. *r* = *Result v* ⟶ (*g v* · *t* ?⦃ *Q* ⦄)) ∧
(∀ *e*. *r* = *Exception* (*Some e*) ⟶ (*h* (*Some e*) · *t* ?⦃ *Q* ⦄))⦄
shows *bind-handle f g h* · *s* ?⦃ *Q* ⦄
⟨*proof*⟩

lemma *runs-to-partial-bind-handle-res-monad*[*runs-to-vcg*]:
fixes *f* :: ('*a*, '*s*) *res-monad*
assumes *f*: *f* · *s* ?⦃ λ*r t*. (∀ *v*. *r* = *Result v* ⟶ (*g v* · *t* ?⦃ *Q* ⦄))⦄
shows *bind-handle f g h* · *s* ?⦃ *Q* ⦄

⟨*proof*⟩

lemma *mono-bind-handle*:
$f \leq f' \Longrightarrow g \leq g' \Longrightarrow h \leq h' \Longrightarrow$ *bind-handle* $f\ g\ h \leq$ *bind-handle* $f'\ g'\ h'$
⟨*proof*⟩

lemma *monotone-bind-handle-le*[*partial-function-mono*]:
monotone $R\ (\leq)\ (\lambda f'.\ f\ f') \Longrightarrow (\bigwedge v.$ *monotone* $R\ (\leq)\ (\lambda f'.\ g\ f'\ v)) \Longrightarrow$
$(\bigwedge e.$ *monotone* $R\ (\leq)\ (\lambda f'.\ h\ f'\ e)) \Longrightarrow$
monotone $R\ (\leq)\ (\lambda f'.$ *bind-handle* $(f\ f')\ (g\ f')\ (h\ f'))$
⟨*proof*⟩

lemma *monotone-bind-handle-ge*[*partial-function-mono*]:
monotone $R\ (\geq)\ (\lambda f'.\ f\ f') \Longrightarrow (\bigwedge v.$ *monotone* $R\ (\geq)\ (\lambda f'.\ g\ f'\ v)) \Longrightarrow$
$(\bigwedge e.$ *monotone* $R\ (\geq)\ (\lambda f'.\ h\ f'\ e)) \Longrightarrow$
monotone $R\ (\geq)\ (\lambda f'.$ *bind-handle* $(f\ f')\ (g\ f')\ (h\ f'))$
⟨*proof*⟩

lemma *run-bind-handle*[*run-spec-monad*]:
run (*bind-handle* $f\ g\ h)\ s =$ *bind-post-state* (*run* $f\ s$)
$(\lambda(r,\ t).$ *case* r *of Exception* $e \Rightarrow$ *run* $(h\ e)\ t \mid$ *Result* $v \Rightarrow$ *run* $(g\ v)\ t)$
⟨*proof*⟩

lemma *always-progress-bind-handle*[*always-progress-intros*]:
always-progress $f \Longrightarrow (\bigwedge v.$ *always-progress* $(g\ v)) \Longrightarrow (\bigwedge e.$ *always-progress* $(h\ e))$
$\Longrightarrow$ *always-progress* (*bind-handle* $f\ g\ h$)
⟨*proof*⟩

lemma *refines-bind-handle'*:
refines $f\ f'\ s\ s'\ (\lambda(r,\ t)\ (r',\ t').$
$(\forall\, e\ e'.\ e \neq$ *default* $\longrightarrow\ e' \neq$ *default* $\longrightarrow\ r =$ *Exception* $e \longrightarrow\ r' =$ *Exception* $e' \longrightarrow$
refines $(h\ e)\ (h'\ e')\ t\ t'\ R) \wedge$
$(\forall\, e\ x'.\ e \neq$ *default* $\longrightarrow\ r =$ *Exception* $e \longrightarrow\ r' =$ *Result* $x' \longrightarrow$
refines $(h\ e)\ (g'\ x')\ t\ t'\ R) \wedge$
$(\forall\, x\ e'.\ e' \neq$ *default* $\longrightarrow\ r =$ *Result* $x \longrightarrow\ r' =$ *Exception* $e' \longrightarrow$
refines $(g\ x)\ (h'\ e')\ t\ t'\ R) \wedge$
$(\forall\, x\ x'.\ r =$ *Result* $x \longrightarrow\ r' =$ *Result* $x' \longrightarrow$
refines $(g\ x)\ (g'\ x')\ t\ t'\ R)) \Longrightarrow$
refines (*bind-handle* $f\ g\ h$) (*bind-handle* $f'\ g'\ h'$) $s\ s'\ R$
⟨*proof*⟩

lemma *refines-bind-handle-bind-handle*:
assumes *f*: *refines* $f\ f'\ s\ s'\ Q$
assumes *ll*: $\bigwedge e\ e'\ t\ t'.\ Q$ (*Exception* e, t) (*Exception* e', $t') \Longrightarrow$
$e \neq$ *default* $\Longrightarrow\ e' \neq$ *default* $\Longrightarrow$
refines $(h\ e)\ (h'\ e')\ t\ t'\ R$
assumes *lr*: $\bigwedge e\ v'\ t\ t'.\ Q$ (*Exception* e, t) (*Result* v', $t') \Longrightarrow\ e \neq$ *default* $\Longrightarrow$
refines $(h\ e)\ (g'\ v')\ t\ t'\ R$

assumes *rl*: $\bigwedge v\ e'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Exception\ e',\ t') \Longrightarrow e' \neq default \Longrightarrow$
refines $(g\ v)\ (h'\ e')\ t\ t'\ R$
assumes *rr*: $\bigwedge v\ v'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Result\ v',\ t') \Longrightarrow$
refines $(g\ v)\ (g'\ v')\ t\ t'\ R$
shows *refines* $(bind\text{-}handle\ f\ g\ h)\ (bind\text{-}handle\ f'\ g'\ h')\ s\ s'\ R$
⟨*proof*⟩

lemma *refines-bind-handle-bind-handle-exn*:
assumes *f*: *refines* $f\ f'\ s\ s'\ Q$
assumes *ll*: $\bigwedge e\ e'\ t\ t'.\ Q\ (Exn\ e,\ t)\ (Exn\ e',\ t') \Longrightarrow$
refines $(h\ (Some\ e))\ (h'\ (Some\ e'))\ t\ t'\ R$
assumes *lr*: $\bigwedge e\ v'\ t\ t'.\ Q\ (Exn\ e,\ t)\ (Result\ v',\ t') \Longrightarrow$
refines $(h\ (Some\ e))\ (g'\ v')\ t\ t'\ R$
assumes *rl*: $\bigwedge v\ e'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Exn\ e',\ t') \Longrightarrow$
refines $(g\ v)\ (h'\ (Some\ e'))\ t\ t'\ R$
assumes *rr*: $\bigwedge v\ v'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Result\ v',\ t') \Longrightarrow$
refines $(g\ v)\ (g'\ v')\ t\ t'\ R$
shows *refines* $(bind\text{-}handle\ f\ g\ h)\ (bind\text{-}handle\ f'\ g'\ h')\ s\ s'\ R$
⟨*proof*⟩

lemma *bind-handle-return-spec-monad*[*simp*]: $bind\text{-}handle\ (return\ v)\ g\ h = g\ v$
⟨*proof*⟩

lemma *bind-handle-throw-spec-monad*[*simp*]:
$v \neq default \Longrightarrow bind\text{-}handle\ (throw\text{-}exception\text{-}or\text{-}result\ v)\ g\ h = h\ v$
⟨*proof*⟩

lemma *bind-handle-bind-handle-spec-monad*:
$bind\text{-}handle\ (bind\text{-}handle\ f\ g1\ h1)\ g2\ h2 =$
$bind\text{-}handle\ f$
$(\lambda v.\ bind\text{-}handle\ (g1\ v)\ g2\ h2)$
$(\lambda e.\ bind\text{-}handle\ (h1\ e)\ g2\ h2)$
⟨*proof*⟩

lemma *mono-bind-handle-spec-monad*:
$mono\ f \Longrightarrow (\bigwedge v.\ mono\ (\lambda x.\ g\ x\ v)) \Longrightarrow (\bigwedge e.\ mono\ (\lambda x.\ h\ x\ e)) \Longrightarrow$
$mono\ (\lambda x.\ bind\text{-}handle\ (f\ x)\ (g\ x)\ (h\ x))$
⟨*proof*⟩

lemma *rel-spec-bind-handle*:
$rel\text{-}spec\ f\ f'\ s\ s'\ (\lambda(r,\ t)\ (r',\ t').$
$(\forall e\ e'.\ e \neq default \longrightarrow e' \neq default \longrightarrow r = Exception\ e \longrightarrow r' = Exception\ e' \longrightarrow$
$rel\text{-}spec\ (h\ e)\ (h'\ e')\ t\ t'\ R) \wedge$
$(\forall e\ x'.\ e \neq default \longrightarrow r = Exception\ e \longrightarrow r' = Result\ x' \longrightarrow$
$rel\text{-}spec\ (h\ e)\ (g'\ x')\ t\ t'\ R) \wedge$
$(\forall x\ e'.\ e' \neq default \longrightarrow r = Result\ x \longrightarrow r' = Exception\ e' \longrightarrow$
$rel\text{-}spec\ (g\ x)\ (h'\ e')\ t\ t'\ R) \wedge$
$(\forall x\ x'.\ r = Result\ x \longrightarrow r' = Result\ x' \longrightarrow$

rel-spec (*g x*) (*g′ x′*) *t t′ R*)) ⟹
rel-spec (*bind-handle f g h*) (*bind-handle f′ g′ h′*) *s s′ R*
⟨*proof*⟩

lemma *bind-finally-bind-handle-conv*: *bind-finally f g* = *bind-handle f* (λ*v*. *g* (*Result v*)) (λ*e*. *g* (*Exception e*))
⟨*proof*⟩

16.9.16 (⪢=)

lemma *run-bind*[*run-spec-monad*]: *run* (*bind f g*) *s* =
bind-post-state (*run f s*) (λ(*r*, *t*). *case r of*
Exception e ⇒ *pure-post-state* (*Exception e*, *t*)
| *Result v* ⇒ *run* (*g v*) *t*)
⟨*proof*⟩

lemma *run-bind-res*: *run* (*f* >>= *g*) *s* = *bind-post-state* (*run f s*) (λ(*Res r*, *s*). *run* (*g r*) *s*)
⟨*proof*⟩

lemma *run-bind-eq-top-iff*:
run (*bind f g*) *s* = ⊤ ⟷ ¬ (*f* ⋅ *s* ⦃ λ*x s*. ∀ *a*. *x* = *Result a* ⟶ *run* (*g a*) *s* ≠ ⊤ ⦄)
⟨*proof*⟩

lemma *always-progress-bind*[*always-progress-intros*]:
always-progress f ⟹ (⋀*v*. *always-progress* (*g v*)) ⟹ *always-progress* (*bind f g*)
⟨*proof*⟩

lemma *run-bind-cong*:
assumes ∗: *run f s* = *run f′ s*
assumes ∗∗: *f* ⋅ *s* ?⦃ λ*x s′*. ∀ *v*. *x* = (*Result v*) ⟶ *run* (*g v*) *s′* = *run* (*g′ v*) *s′* ⦄
shows *run* (*bind f g*) *s* = *run* (*bind f′ g′*) *s*
⟨*proof*⟩

lemma *runs-to-bind-iff*[*runs-to-iff*]:
bind f g ⋅ *s* ⦃ *Q* ⦄ ⟷ *f* ⋅ *s* ⦃ λ*r t*.
(∀ *v*. *r* = *Result v* ⟶ *g v* ⋅ *t* ⦃ *Q* ⦄) ∧
(∀ *e*. *r* = *Exception e* ⟶ *e* ≠ *default* ⟶ *Q* (*Exception e*) *t*)⦄
⟨*proof*⟩

lemma *runs-to-bind*[*runs-to-vcg*]:
f ⋅ *s* ⦃ λ*r t*. (∀ *v*. *r* = *Result v* ⟶ *g v* ⋅ *t* ⦃ *Q* ⦄) ∧
(∀ *e*. *r* = *Exception e* ⟶ *e* ≠ *default* ⟶ *Q* (*Exception e*) *t*)⦄ ⟹
bind f g ⋅ *s* ⦃ *Q* ⦄
⟨*proof*⟩

lemma *runs-to-bind-exception*[*runs-to-vcg*]:
fixes *f* :: ('*e*, '*a*, '*s*) *exn-monad*

assumes [*runs-to-vcg*]: $f \cdot s \{\!|\ \lambda r\ t.\ (\forall v.\ r = Result\ v \longrightarrow g\ v \cdot t \{\!|\ Q\ |\!\}) \wedge$
$(\forall e.\ r = Exn\ e \longrightarrow Q\ (Exn\ e)\ t)|\!\}$
shows *bind f g* $\cdot s \{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *runs-to-bind-res*[*runs-to-vcg*]:
fixes $f :: ('a, 's)$ *res-monad*
assumes [*runs-to-vcg*]: $f \cdot s \{\!|\ \lambda Res\ v\ t.\ g\ v \cdot t \{\!|\ Q\ |\!\}|\!\}$
shows *bind f g* $\cdot s \{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *runs-to-partial-bind*[*runs-to-vcg*]:
$f \cdot s\ ?\{\!|\ \lambda r\ t.\ (\forall v.\ r = Result\ v \longrightarrow g\ v \cdot t\ ?\{\!|\ Q\ |\!\}) \wedge$
$(\forall e.\ r = Exception\ e \longrightarrow e \neq default \longrightarrow Q\ (Exception\ e)\ t)|\!\} \Longrightarrow$
bind f g $\cdot s\ ?\{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *runs-to-partial-bind-exeption-monad*[*runs-to-vcg*]:
fixes $f :: ('e, 'a, 's)$ *exn-monad*
assumes [*runs-to-vcg*]: $f \cdot s\ ?\{\!|\ \lambda r\ t.\ (\forall v.\ r = Result\ v \longrightarrow g\ v \cdot t\ ?\{\!|\ Q\ |\!\}) \wedge$
$(\forall e.\ r = Exn\ e \longrightarrow Q\ (Exn\ e)\ t)|\!\}$
shows *bind f g* $\cdot s\ ?\{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *runs-to-partial-bind-res-monad*[*runs-to-vcg*]:
fixes $f :: ('a, 's)$ *res-monad*
assumes [*runs-to-vcg*]: $f \cdot s\ ?\{\!|\ \lambda Res\ v\ t.\ g\ v \cdot t\ ?\{\!|\ Q\ |\!\}|\!\}$
shows *bind f g* $\cdot s\ ?\{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *bind-return*[*simp*]: $(bind\ m\ return) = m$
⟨*proof*⟩

lemma *bind-skip*[*simp*]: $bind\ m\ (\lambda x.\ skip) = m$
⟨*proof*⟩

lemma *return-bind*[*simp*]: $bind\ (return\ x)\ f = f\ x$
⟨*proof*⟩

lemma *bind-assoc*: $bind\ (bind\ f\ g)\ h = bind\ f\ (\lambda x.\ bind\ (g\ x)\ h)$
⟨*proof*⟩

lemma *mono-bind*: $f \leq f' \Longrightarrow g \leq g' \Longrightarrow bind\ f\ g \leq bind\ f'\ g'$
⟨*proof*⟩

lemma *mono-bind-spec-monad*:
$mono\ f \Longrightarrow (\bigwedge v.\ mono\ (\lambda x.\ g\ x\ v)) \Longrightarrow mono\ (\lambda x.\ bind\ (f\ x)\ (g\ x))$
⟨*proof*⟩

lemma *monotone-bind-le*[*partial-function-mono*]:
monotone R (≤) (λ*f'*. *f f'*) ⟹ (⋀*v*. *monotone R* (≤) (λ*f'*. *g f' v*))
⟹ *monotone R* (≤) (λ*f'*. *bind* (*f f'*) (*g f'*))
⟨*proof*⟩

lemma *monotone-bind-ge*[*partial-function-mono*]:
monotone R (≥) (λ*f'*. *f f'*) ⟹ (⋀*v*. *monotone R* (≥) (λ*f'*. *g f' v*))
⟹ *monotone R* (≥) (λ*f'*. *bind* (*f f'*) (*g f'*))
⟨*proof*⟩

lemma *refines-bind'*:
assumes *f*: *refines f f' s s'* (λ(*x*, *t*) (*x'*, *t'*).
(∀ *e*. *e* ≠ *default* ⟶ *x* = *Exception e* ⟶
((∀ *e'*. *e'* ≠ *default* ⟶ *x'* = *Exception e'* ⟶ *R* (*Exception e*, *t*) (*Exception e'*, *t'*)) ∧
(∀ *v'*. *x'* = *Result v'* ⟶ *refines* (*throw-exception-or-result e*) (*g' v'*) *t t' R*)))
∧
(∀ *v*. *x* = *Result v* ⟶
((∀ *e'*. *e'* ≠ *default* ⟶ *x'* = *Exception e'* ⟶
refines (*g v*) (*throw-exception-or-result e'*) *t t' R*) ∧
(∀ *v'*. *x'* = *Result v'* ⟶ *refines* (*g v*) (*g' v'*) *t t' R*))))
shows *refines* (*bind f g*) (*bind f' g'*) *s s' R*
⟨*proof*⟩

lemma *refines-bind*:
assumes *f*: *refines f f' s s' Q*
assumes *ll*: ⋀*e e' t t'*.
Q (*Exception e*, *t*) (*Exception e'*, *t'*) ⟹ *e* ≠ *default* ⟹ *e'* ≠ *default* ⟹
R (*Exception e*, *t*) (*Exception e'*, *t'*)
assumes *lr*: ⋀*e v' t t'*. *Q* (*Exception e*, *t*) (*Result v'*, *t'*) ⟹ *e* ≠ *default* ⟹
refines (*yield* (*Exception e*)) (*g' v'*) *t t' R*
assumes *rl*: ⋀*v e' t t'*. *Q* (*Result v*, *t*) (*Exception e'*, *t'*) ⟹ *e'* ≠ *default* ⟹
refines (*g v*) (*yield* (*Exception e'*)) *t t' R*
assumes *rr*: ⋀*v v' t t'*. *Q* (*Result v*, *t*) (*Result v'*, *t'*) ⟹
refines (*g v*) (*g' v'*) *t t' R*
shows *refines* (*bind f g*) (*bind f' g'*) *s s' R*
⟨*proof*⟩

lemma *refines-bind-bind-exn*:
assumes *f*: *refines f f' s s' Q*
assumes *ll*: ⋀*e e' t t'*. *Q* (*Exn e*, *t*) (*Exn e'*, *t'*) ⟹ *R* (*Exn e*, *t*) (*Exn e'*, *t'*)
assumes *lr*: ⋀*e v' t t'*. *Q* (*Exn e*, *t*) (*Result v'*, *t'*) ⟹ *refines* (*throw e*) (*g' v'*) *t t' R*
assumes *rl*: ⋀*v e' t t'*. *Q* (*Result v*, *t*) (*Exn e'*, *t'*) ⟹ *refines* (*g v*) (*throw e'*) *t t' R*
assumes *rr*: ⋀*v v' t t'*. *Q* (*Result v*, *t*) (*Result v'*, *t'*) ⟹ *refines* (*g v*) (*g' v'*) *t t' R*
shows *refines* (*f* ⤜ *g*) (*f'* ⤜ *g'*) *s s' R*
⟨*proof*⟩

lemma *refines-bind-res*:
assumes *f*: *refines f f' s s'* (λ(*Res r*, *t*) (*Res r'*, *t'*). *refines* (*g r*) (*g' r'*) *t t' R*)
shows *refines* ((*bind f g*)::('*a*,'*s*) *res-monad*) ((*bind f' g'*)::('*b*, '*t*) *res-monad*) *s s' R*
⟨*proof*⟩

lemma *refines-bind-res'*:
assumes *f*: *refines f f' s s' Q*
assumes *g*: ⋀*r t r' t'*. *Q* (*Result r*, *t*) (*Result r'*, *t'*) ⟹ *refines* (*g r*) (*g' r'*) *t t' R*
shows *refines* ((*bind f g*)::('*a*,'*s*) *res-monad*) ((*bind f' g'*)::('*b*, '*t*) *res-monad*) *s s' R*
⟨*proof*⟩

lemma *refines-bind-res-bind-exn*:
assumes *f*: *refines f f' s s' Q*
assumes *rl*: ⋀*v e t t'*. *Q* (*Result v*, *t*) (*Exn e*, *t'*) ⟹
refines (*g v*) (*throw e*) *t t' R*
assumes *rr*: ⋀*v v' t t'*. *Q* (*Result v*, *t*) (*Result v'*, *t'*) ⟹
refines (*g v*) (*g' v'*) *t t' R*
shows *refines* (*bind f g* :: (-, -) *res-monad*) (*bind f' g'*) *s s' R*
⟨*proof*⟩

lemma *refines-bind-bind-exn-wp*:
assumes *f*: *refines f f' s s'* (λ(*r*, *t*) (*r'*,*t'*).
(*case r of*
Exn e ⇒ (*case r' of Exn e'* ⇒ *R* (*Exn e*, *t*) (*Exn e'*, *t'*) | *Result v'* ⇒ *refines* (*throw e*) (*g' v'*) *t t' R*)
| *Result v* ⇒ (*case r' of Exn e'* ⇒ *refines* (*g v*) (*throw e'*) *t t' R* | *Result v'* ⇒ *refines* (*g v*) (*g' v'*) *t t' R*)))
shows *refines* (*f* ⤜ *g*) (*f'* ⤜ *g'*) *s s' R*
⟨*proof*⟩

lemma *rel-spec-bind-res*:
rel-spec f f' s s' (λ(*Res r*, *t*) (*Res r'*, *t'*). *rel-spec* (*g r*) (*g' r'*) *t t' R*) ⟹
rel-spec ((*bind f g*)::('*a*,'*s*) *res-monad*) ((*bind f' g'*)::('*b*, '*t*) *res-monad*) *s s' R*
⟨*proof*⟩

lemma *rel-spec-bind-res'*:
assumes *f*: *rel-spec f f' s s' Q*
assumes *g*: ⋀*r t r' t'*. *Q* (*Result r*, *t*) (*Result r'*, *t'*) ⟹ *rel-spec* (*g r*) (*g' r'*) *t t' R*
shows *rel-spec* ((*bind f g*)::('*a*,'*s*) *res-monad*) ((*bind f' g'*)::('*b*, '*t*) *res-monad*) *s s' R*
⟨*proof*⟩

lemma *rel-spec-bind-bind*:
assumes *f*: *rel-spec f f' s s' Q*

assumes *ll*: $\bigwedge e\ e'\ t\ t'.$
$Q\ (Exception\ e,\ t)\ (Exception\ e',\ t') \Longrightarrow e \neq default \Longrightarrow e' \neq default \Longrightarrow$
$R\ (Exception\ e,\ t)\ (Exception\ e',\ t')$
assumes *lr*: $\bigwedge e\ v'\ t\ t'.\ Q\ (Exception\ e,\ t)\ (Result\ v',\ t') \Longrightarrow e \neq default \Longrightarrow$
$rel\text{-}spec\ (yield\ (Exception\ e))\ (g'\ v')\ t\ t'\ R$
assumes *rl*: $\bigwedge v\ e'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Exception\ e',\ t') \Longrightarrow e' \neq default \Longrightarrow$
$rel\text{-}spec\ (g\ v)\ (yield\ (Exception\ e'))\ t\ t'\ R$
assumes *rr*: $\bigwedge v\ v'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Result\ v',\ t') \Longrightarrow$
$rel\text{-}spec\ (g\ v)\ (g'\ v')\ t\ t'\ R$
shows $rel\text{-}spec\ (bind\ f\ g)\ (bind\ f'\ g')\ s\ s'\ R$
⟨*proof*⟩

lemma *rel-spec-bind-exn*:
assumes *f*: $rel\text{-}spec\ f\ f'\ s\ s'\ Q$
assumes *ll*: $\bigwedge e\ e'\ t\ t'.\ Q\ (Exn\ e,\ t)\ (Exn\ e',\ t') \Longrightarrow R\ (Exn\ e,\ t)\ (Exn\ e',\ t')$
assumes *lr*: $\bigwedge e\ v'\ t\ t'.\ Q\ (Exn\ e,\ t)\ (Result\ v',\ t') \Longrightarrow rel\text{-}spec\ (throw\ e)\ (g'\ v')\ t\ t'\ R$
assumes *rl*: $\bigwedge v\ e'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Exn\ e',\ t') \Longrightarrow rel\text{-}spec\ (g\ v)\ (throw\ e')\ t\ t'\ R$
assumes *rr*: $\bigwedge v\ v'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Result\ v',\ t') \Longrightarrow rel\text{-}spec\ (g\ v)\ (g'\ v')\ t\ t'\ R$
shows $rel\text{-}spec\ (bind\ f\ g)\ (bind\ f'\ g')\ s\ s'\ R$
⟨*proof*⟩

lemma *refines-bind-right*:
assumes *f*: $refines\ f\ f'\ s\ s'\ Q$
assumes *ll*: $\bigwedge e\ e'\ t\ t'.$
$Q\ (Exception\ e,\ t)\ (Exception\ e',\ t') \Longrightarrow e \neq default \Longrightarrow e' \neq default \Longrightarrow$
$R\ (Exception\ e,\ t)\ (Exception\ e',\ t')$
assumes *lr*: $\bigwedge e\ v'\ t\ t'.\ Q\ (Exception\ e,\ t)\ (Result\ v',\ t') \Longrightarrow e \neq default \Longrightarrow$
$refines\ (yield\ (Exception\ e))\ (g'\ v')\ t\ t'\ R$
assumes *rl*: $\bigwedge v\ e'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Exception\ e',\ t') \Longrightarrow e' \neq default \Longrightarrow$
$R\ (Result\ v,\ t)\ (Exception\ e',\ t')$
assumes *rr*: $\bigwedge v\ v'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Result\ v',\ t') \Longrightarrow$
$refines\ (return\ v)\ (g'\ v')\ t\ t'\ R$
shows $refines\ f\ (bind\ f'\ g')\ s\ s'\ R$
⟨*proof*⟩

lemma *refines-bind-left*:
assumes *f*: $refines\ f\ f'\ s\ s'\ Q$
assumes *ll*: $\bigwedge e\ e'\ t\ t'.$
$Q\ (Exception\ e,\ t)\ (Exception\ e',\ t') \Longrightarrow e \neq default \Longrightarrow e' \neq default \Longrightarrow$
$R\ (Exception\ e,\ t)\ (Exception\ e',\ t')$
assumes *lr*: $\bigwedge e\ v'\ t\ t'.\ Q\ (Exception\ e,\ t)\ (Result\ v',\ t') \Longrightarrow e \neq default \Longrightarrow$
$R\ (Exception\ e,\ t)\ (Result\ v',\ t')$
assumes *rl*: $\bigwedge v\ e'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Exception\ e',\ t') \Longrightarrow e' \neq default \Longrightarrow$
$refines\ (g\ v)\ (yield\ (Exception\ e'))\ t\ t'\ R$
assumes *rr*: $\bigwedge v\ v'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Result\ v',\ t') \Longrightarrow$
$refines\ (g\ v)\ (return\ v')\ t\ t'\ R$

shows *refines* (*bind f g*) *f′ s s′ R*
⟨*proof*⟩

lemma *rel-spec-bind-right*:
assumes *f*: *rel-spec f f′ s s′ Q*
assumes *ll*: ⋀*e e′ t t′*.
Q (*Exception e*, *t*) (*Exception e′*, *t′*) ⟹ *e* ≠ *default* ⟹ *e′* ≠ *default* ⟹
R (*Exception e*, *t*) (*Exception e′*, *t′*)
assumes *lr*: ⋀*e v′ t t′*. *Q* (*Exception e*, *t*) (*Result v′*, *t′*) ⟹ *e* ≠ *default* ⟹
rel-spec (*yield* (*Exception e*)) (*g′ v′*) *t t′ R*
assumes *rl*: ⋀*v e′ t t′*. *Q* (*Result v*, *t*) (*Exception e′*, *t′*) ⟹ *e′* ≠ *default* ⟹
R (*Result v*, *t*) (*Exception e′*, *t′*)
assumes *rr*: ⋀*v v′ t t′*. *Q* (*Result v*, *t*) (*Result v′*, *t′*) ⟹
rel-spec (*return v*) (*g′ v′*) *t t′ R*
shows *rel-spec f* (*bind f′ g′*) *s s′ R*
⟨*proof*⟩

lemma *refines-bind-handle-left′*:
f · *s* ⦃ λ*v s′*. (∀ *r*. *v* = *Result r* ⟶ *refines* (*g r*) *k s′ t R*) ∧
(∀ *e*. *e* ≠ *default* ⟶ *v* = *Exception e* ⟶ *refines* (*h e*) *k s′ t R*) ⦄ ⟹
refines (*bind-handle f g h*) *k s t R*
⟨*proof*⟩

lemma *refines-bind-left-res*:
f · *s* ⦃ λ*Res r s′*. *refines* (*g r*) *h s′ t R* ⦄ ⟹ *refines* (*f* >>= *g*) *h s t R*
⟨*proof*⟩

lemma *refines-bind-left-exn*:
f · *s* ⦃ λ*r s′*. (∀ *a*. *r* = *Result a* ⟶ *refines* (*g a*) *h s′ t R*) ∧
(∀ *e*. *r* = *Exn e* ⟶ *refines* (*throw e*) *h s′ t R*) ⦄ ⟹
refines (*f* >>= *g*) *h s t R*
⟨*proof*⟩

lemma *runs-to-partial-bind1*:
(⋀*r s*. (*g r*) · *s* ?⦃*P*⦄) ⟹ ((*f* >>= *g*)::(′*a*, ′*s*) *res-monad*) · *s* ?⦃*P*⦄
⟨*proof*⟩

lemma *unknown-bind-const*[*simp*]: *unknown* >>= (λ*x*. *f*) = *f*
⟨*proof*⟩

lemma *bind-cong-left*:
fixes *f*::(′*e*::*default*, ′*a*, ′*s*) *spec-monad*
shows (⋀*r*. *g r* = *g′ r*) ⟹ (*f* >>= *g*) = (*f* >>= *g′*)
⟨*proof*⟩

lemma *bind-cong-right*:
fixes *f*::(′*e*::*default*, ′*a*, ′*s*) *spec-monad*
shows *f* = *f′* ⟹ (*f* >>= *g*) = (*f′* >>= *g*)
⟨*proof*⟩

lemma *rel-spec-bind-res''*:
fixes *f*::('*a*, '*s*) *res-monad*
shows *f* · *s* ?⦃ λ*Res r t*. *rel-spec* (*g r*) (*g' r*) *t t R*⦄ ⟹ *rel-spec* (*f* >>= *g*) (*f* >>= *g'*) *s s R*
⟨*proof*⟩

lemma *rel-spec-monad-bind-rel-exception-or-result*:
assumes *mn*: *rel-spec-monad R* (*rel-exception-or-result E P*) *m n*
and *fg*: *rel-fun P* (*rel-spec-monad R* (*rel-exception-or-result E Q*)) *f g*
shows *rel-spec-monad R* (*rel-exception-or-result E Q*) (*m* >>= *f*) (*n* >>= *g*)
⟨*proof*⟩

lemma *rel-spec-monad-bind-rel-exception-or-result'*:
(⋀*x*. *rel-spec-monad* (=) (*rel-exception-or-result* (=) *Q*) (*f x*) (*g x*)) ⟹
rel-spec-monad (=) (*rel-exception-or-result* (=) *Q*) (*m* >>= *f*) (*m* >>= *g*)
⟨*proof*⟩

lemma *rel-spec-monad-bind*:
rel-spec-monad R (λ*Res v1 Res v2*. *P v1 v2*) *m n* ⟹ *rel-fun P* (*rel-spec-monad R Q*) *f g* ⟹
rel-spec-monad R Q (*m* >>= *f*) (*n* >>= *g*)
⟨*proof*⟩

lemma *rel-spec-monad-bind-left*:
assumes *mn*: *rel-spec-monad R* (λ*Res v1 Res v2*. *P v1 v2*) *m n*
and *fg*: *rel-fun P* (*rel-spec-monad R Q*) *f return*
shows *rel-spec-monad R Q* (*m* >>= *f*) *n*
⟨*proof*⟩

lemma *rel-spec-monad-bind'*:
fixes *m*:: ('*a*, '*s*) *res-monad*
shows (⋀*x*. *rel-spec-monad* (=) *Q* (*f x*) (*g x*)) ⟹ *rel-spec-monad* (=) *Q* (*m* >>= *f*) (*m* >>= *g*)
⟨*proof*⟩

lemma *run-bind-cong-simple*:
run f s = *run f' s* ⟹ (⋀*v s*. *run* (*g v*) *s* = *run* (*g' v*) *s*) ⟹
run (*bind f g*) *s* = *run* (*bind f' g'*) *s*
⟨*proof*⟩

lemma *run-bind-cong-simple-same-head*: (⋀*v s*. *run* (*g v*) *s* = *run* (*g' v*) *s*) ⟹
run (*bind f g*) *s* = *run* (*bind f g'*) *s*
⟨*proof*⟩

lemma *refines-bind-same*:
refines (*f* >>= *g*) (*f* >>= *g'*) *s s R* **if** *f* · *s* ⦃λ*Res y t*. *refines* (*g y*) (*g' y*) *t t R*⦄
⟨*proof*⟩

lemma *refines-bind-handle-right-runs-to-partialI*:
$g \cdot t$ *?⦃* $\lambda r\ t'.\ (\forall e.\ e \neq default \longrightarrow r = Exception\ e \longrightarrow refines\ f\ (k\ e)\ s\ t'\ R) \wedge$
$(\forall a.\ r = Result\ a \longrightarrow refines\ f\ (h\ a)\ s\ t'\ R)$ *⦄* $\Longrightarrow$ *always-progress* $g \Longrightarrow$
refines f (*bind-handle* $g\ h\ k$) $s\ t\ R$
⟨*proof*⟩

lemma *refines-bind-right-runs-to-partialI*:
$g \cdot t$ *?⦃* $\lambda r\ t'.\ (\forall e.\ e \neq default \longrightarrow r = Exception\ e \longrightarrow$
refines f (*throw-exception-or-result* e) $s\ t'\ R) \wedge$
$(\forall a.\ r = Result\ a \longrightarrow refines\ f\ (h\ a)\ s\ t'\ R)$ *⦄* $\Longrightarrow$ *always-progress* $g \Longrightarrow$
refines f (*bind* $g\ h$) $s\ t\ R$
⟨*proof*⟩

lemma *refines-bind-right-runs-toI*:
$g \cdot t$ *⦃* $\lambda Res\ r\ t'.$ *refines* $f\ (h\ r)\ s\ t'\ R$ *⦄* $\Longrightarrow$ *always-progress* $g \Longrightarrow$
refines f (g >>= h) $s\ t\ R$
⟨*proof*⟩

lemma *refines-bind-left-refine*:
refines (f >>= g) $h\ s\ t\ R$
if *refines* $f\ f'\ s\ s$ (=) *refines* (f' >>= g) $h\ s\ t\ R$
⟨*proof*⟩

lemma *refines-bind-right-single*:
assumes x: *pure-post-state* (*Result* x, u) $\leq$ *run* $g\ t$ **and** h: *refines* $f\ (h\ x)\ s\ u\ R$
shows *refines* f (g >>= h) $s\ t\ R$
⟨*proof*⟩

lemma *return-let-bind*: (*return* (*let* $v = f'$ *in* ($g'\ v$))) = *do* {v <− *return* f'; *return* ($g'\ v$)}
⟨*proof*⟩

lemma *rel-spec-monad-rel-xval-bind*:
assumes *f-f'*: *rel-spec-monad* S (*rel-xval* $L\ P$) $f\ f'$
assumes *Res-Res*: $\bigwedge v\ v'.\ P\ v\ v' \Longrightarrow$ *rel-spec-monad* S (*rel-xval* $L\ R$) ($g\ v$) ($g'\ v'$)
shows *rel-spec-monad* S (*rel-xval* $L\ R$) (f >>= g) (f' >>= g')
⟨*proof*⟩

lemma *rel-spec-monad-rel-xval-same-bind*:
assumes *f-f'*: *rel-spec-monad* S (*rel-xval* $L\ R$) $f\ f'$
assumes *Res-Res*: $\bigwedge v\ v'.\ R\ v\ v' \Longrightarrow$ *rel-spec-monad* S (*rel-xval* $L\ R$) ($g\ v$) ($g'\ v'$)
shows *rel-spec-monad* S (*rel-xval* $L\ R$) (f >>= g) (f' >>= g')
⟨*proof*⟩

lemma *rel-spec-monad-rel-xval-result-eq-bind*:
assumes *f-f'*: *rel-spec-monad* S (*rel-xval* L (=)) $f\ f'$
assumes *Res-Res*: $\bigwedge v.$ *rel-spec-monad* S (*rel-xval* L (=)) ($g\ v$) ($g'\ v$)

shows *rel-spec-monad S* (*rel-xval L* (=)) (*f* ⤜ *g*) (*f′* ⤜ *g′*)
⟨*proof*⟩

lemma *rel-spec-monad-fail*: *rel-spec-monad Q R fail fail*
⟨*proof*⟩

lemma *bind-fail*[*simp*]: *fail* ⤜ *X* = *fail*
⟨*proof*⟩

16.9.17 *assert*

lemma *assert-simps*[*simp*]:
assert True = *return* ()
assert False = *top*
⟨*proof*⟩

lemma *always-progress-assert*[*always-progress-intros*]: *always-progress* (*assert P*)
⟨*proof*⟩

lemma *run-assert*[*run-spec-monad*]:
run (*assert P*) *s* = (*if P then pure-post-state* (*Result* (), *s*) *else* ⊤)
⟨*proof*⟩

lemma *runs-to-assert-iff*[*simp*]: *assert P* · *s* ⦃ *Q* ⦄ ⟷ *P* ∧ *Q* (*Result* ()) *s*
⟨*proof*⟩

lemma *runs-to-assert*[*runs-to-vcg*]: *P* ⟹ *Q* (*Result* ()) *s* ⟹ *assert P* · *s* ⦃ *Q* ⦄
⟨*proof*⟩

lemma *runs-to-partial-assert-iff*[*simp*]: *assert P* · *s* ?⦃ *Q* ⦄ ⟷ (*P* ⟶ *Q* (*Result* ()) *s*)
⟨*proof*⟩

lemma *refines-top-iff*[*simp*]: *refines* ⊤ *g s t R* ⟷ *run g t* = ⊤
⟨*proof*⟩

lemma *refines-assert*:
refines (*assert P*) (*assert Q*) *s t R* ⟷ (*Q* ⟶ *P* ∧ *R* (*Result* (), *s*) (*Result* (), *t*))
⟨*proof*⟩

16.9.18 *assume*

lemma *assume-simps*[*simp*]:
assume True = *return* ()
assume False = *bot*
⟨*proof*⟩

lemma *always-progress-assume*[*always-progress-intros*]: *P* ⟹ *always-progress* (*assume P*)

⟨*proof*⟩

lemma *run-assume*[*run-spec-monad*]:
run (*assume P*) *s* = (*if P then pure-post-state* (*Result* (), *s*) *else* ⊥)
⟨*proof*⟩

lemma *run-assume-simps*[*run-spec-monad*, *simp*]:
P ⟹ *run* (*assume P*) *s* = *pure-post-state* (*Result* (), *s*)
¬ *P* ⟹ *run* (*assume P*) *s* = ⊥
⟨*proof*⟩

lemma *runs-to-assume-iff*[*simp*]: *assume P* · *s* ⦃ *Q* ⦄ ⟷ (*P* ⟶ *Q* (*Result* ()) *s*)
⟨*proof*⟩

lemma *runs-to-partial-assume-iff*[*simp*]: *assume P* · *s* ?⦃ *Q* ⦄ ⟷ (*P* ⟶ *Q* (*Result* ()) *s*)
⟨*proof*⟩

lemma *runs-to-assume*[*runs-to-vcg*]: (*P* ⟹ *Q* (*Result* ()) *s*) ⟹ *assume P* · *s* ⦃ *Q* ⦄
⟨*proof*⟩

16.9.19 *assume-outcome*

lemma *run-assume-outcome*[*run-spec-monad*, *simp*]: *run* (*assume-outcome f*) *s* = *Success* (*f s*)
⟨*proof*⟩

lemma *always-progress-assume-outcome*[*always-progress-intros*]:
(⋀*s*. *f s* ≠ {}) ⟹ *always-progress* (*assume-outcome f*)
⟨*proof*⟩

lemma *runs-to-assume-outcome*[*runs-to-vcg*]:
(*assume-outcome f*) · *s* ⦃λ*r t*. (*r*, *t*) ∈ *f s*⦄
⟨*proof*⟩

lemma *runs-to-assume-outcome-iff*[*runs-to-iff*]:
(*assume-outcome f*) · *s* ⦃*Q*⦄ ⟷ (∀(*r*, *t*) ∈ *f s*. *Q r t*)
⟨*proof*⟩

lemma *runs-to-partial-assume-outcome*[*runs-to-vcg*]:
(*assume-outcome f*) · *s* ?⦃λ*r t*. (*r*, *t*) ∈ *f s*⦄
⟨*proof*⟩

lemma *assume-outcome-elementary*:
assume-outcome f = *do* {*s* ← *get-state*; (*r*, *t*) ← *select* (*f s*); *set-state t*; *yield r*}
⟨*proof*⟩

16.9.20 *assume-result-and-state*

lemma *run-assume-result-and-state*[*run-spec-monad*, *simp*]:
run (*assume-result-and-state f*) *s* = *Success* ((λ(*v*, *t*). (*Result v*, *t*)) ‘ *f s*)
⟨*proof*⟩

lemma *always-progress-assume-result-and-state*[*always-progress-intros*]:
(⋀*s*. *f s* ≠ {}) ⟹ *always-progress* (*assume-result-and-state f*)
⟨*proof*⟩

lemma *runs-to-assume-result-and-state*[*runs-to-vcg*]:
assume-result-and-state f · *s* ⦃λ*r t*. ∃ *v*. *r* = *Result v* ∧ (*v*, *t*) ∈ *f s*⦄
⟨*proof*⟩

lemma *runs-to-assume-result-and-state-iff*[*runs-to-iff*]:
assume-result-and-state f · *s* ⦃*Q*⦄ ⟷ (∀ (*v*, *t*) ∈ *f s*. *Q* (*Result v*) *t*)
⟨*proof*⟩

lemma *runs-to-partial-assume-result-and-state*[*runs-to-vcg*]:
assume-result-and-state f · *s* ?⦃λ*r t*. ∃ *v*. *r* = *Result v* ∧ (*v*, *t*) ∈ *f s*⦄
⟨*proof*⟩

lemma *refines-assume-result-and-state-right*:
f · *s* ⦃λ*r s*′. ∃ *r*′ *t*′. (*r*′, *t*′) ∈ *g t* ∧ *R* (*r*, *s*′) (*Result r*′, *t*′) ⦄ ⟹
refines f (*assume-result-and-state g*) *s t R*
⟨*proof*⟩

lemma *refines-assume-result-and-state*:
sim-set R (*P s*) (*Q t*) ⟹
refines (*assume-result-and-state P*) (*assume-result-and-state Q*) *s t*
(λ(*Res v*, *s*) (*Res w*, *t*). *R* (*v*, *s*) (*w*, *t*))
⟨*proof*⟩

lemma *refines-assume-result-and-state-iff*:
refines (*assume-result-and-state A*) (*assume-result-and-state B*) *s t Q* ⟷
sim-set (λ(*v*, *s*′) (*w*, *t*′). *Q* (*Result v*, *s*′) (*Result w*, *t*′)) (*A s*) (*B t*)
⟨*proof*⟩

16.9.21 *gets*

lemma *run-gets*[*run-spec-monad*, *simp*]:*run* (*gets f*) *s* = *pure-post-state* (*Result* (*f s*), *s*)
⟨*proof*⟩

lemma *always-progress-gets*[*always-progress-intros*]: *always-progress* (*gets f*)
⟨*proof*⟩

lemma *runs-to-gets*[*runs-to-vcg*]: *gets f* · *s* ⦃λ*r t*. *r* = *Result* (*f s*) ∧ *t* = *s*⦄
⟨*proof*⟩

lemma *runs-to-gets-iff*[*runs-to-iff*]: *gets f · s* {‖*Q*‖} ⟷ *Q* (*Result* (*f s*)) *s*
⟨*proof*⟩

lemma *runs-to-partial-gets*[*runs-to-vcg*]: *gets f · s ?*{‖λ*r t*. *r* = *Result* (*f s*) ∧ *t* = *s*‖}
⟨*proof*⟩

lemma *refines-gets*: *R* (*Result* (*f s*), *s*) (*Result* (*g t*), *t*) ⟹ *refines* (*gets f*) (*gets g*) *s t R*
⟨*proof*⟩

lemma *rel-spec-gets*: *R* (*Result* (*f s*), *s*) (*Result* (*g t*), *t*) ⟹ *rel-spec* (*gets f*) (*gets g*) *s t R*
⟨*proof*⟩

lemma *runs-to-always-progress-to-gets*:
(⋀*s*. *f · s* {‖λ*r t*. *t* = *s* ∧ *r* = *Result* (*v s*)‖}) ⟹ *always-progress f* ⟹ *f* = *gets v*
⟨*proof*⟩

lemma *gets-let-bind*: (*gets* (λ*s*. *let v* = *f′ s in* (*g′ v s*))) = *do* {*v* <− *gets f′*; *gets* (*g′ v*)}
⟨*proof*⟩

16.9.22 *assert-result-and-state*

lemma *run-assert-result-and-state*[*run-spec-monad*]:
run (*assert-result-and-state f*) *s* =
(*if f s* = {} *then* ⊤ *else Success* ((λ(*v*, *t*). (*Result v*, *t*)) ‘ *f s*))
⟨*proof*⟩

lemma *always-progress-assert-result-and-state*[*always-progress-intros*]:
always-progress (*assert-result-and-state f*)
⟨*proof*⟩

lemma *runs-to-assert-result-and-state-iff*[*runs-to-iff*]:
assert-result-and-state f · s {‖*Q*‖} ⟷ (*f s* ≠ {} ∧ (∀(*v*,*t*) ∈ *f s*. *Q* (*Result v*) *t*))
⟨*proof*⟩

lemma *runs-to-assert-result-and-state*[*runs-to-vcg*]:
f s ≠ {} ⟹ *assert-result-and-state f · s* {‖λ*r t*. ∃ *v*. *r* = *Result v* ∧ (*v*, *t*) ∈ *f s*‖}
⟨*proof*⟩

lemma *runs-to-partial-assert-result-and-state*[*runs-to-vcg*]:
assert-result-and-state f · s ?{‖λ*r t*. ∃ *v*. *r* = *Result v* ∧ (*v*, *t*) ∈ *f s*‖}
⟨*proof*⟩

lemma *runs-to-state-select*[*runs-to-vcg*]:
∃ *t*. (*s*, *t*) ∈ *R* ⟹ *state-select R · s* {‖ λ*r t*. *r* = *Result* () ∧ (*s*, *t*) ∈ *R* ‖}
⟨*proof*⟩

lemma *runs-to-partial-state-select*[*runs-to-vcg*]:
state-select R · s ?⦃ λr t. r = Result () ∧ (s, t) ∈ R ⦄
⟨*proof*⟩

lemma *refines-assert-result-and-state*:
assumes *sim*: (⋀*r′ s′*. (*r′*, *s′*) ∈ *f s* ⟹ (∃ *v′ t′*. (*v′*, *t′*) ∈ *g t* ∧ *R* (*Result r′*, *s′*) (*Result v′*, *t′*)))
assumes *emp*: *f s* = {} ⟹ *g t* = {}
shows *refines* (*assert-result-and-state f*) (*assert-result-and-state g*) *s t R*
⟨*proof*⟩

lemma *refines-state-select*:
assumes *sim*: (⋀*s′*. (*s*, *s′*) ∈ *f* ⟹ (∃ *t′*. (*t*, *t′*) ∈ *g* ∧ *R* (*Result* (), *s′*) (*Result* (), *t′*)))
assumes *emp*: ∄ *s′*. (*s*, *s′*) ∈ *f* ⟹ ∄ *t′*. (*t*, *t′*) ∈ *g*
shows *refines* (*state-select f*) (*state-select g*) *s t R*
⟨*proof*⟩

16.9.23 *assuming*

lemma *run-assuming*[*run-spec-monad*]:
run (*assuming g*) *s* = (*if g s then pure-post-state* (*Result* (), *s*) *else* ⊥)
⟨*proof*⟩

lemma *always-progress-assuming*[*always-progress-intros*]: *always-progress* (*assuming g*) ⟷ (∀ *s*. *g s*)
⟨*proof*⟩

lemma *runs-to-assuming*[*runs-to-vcg*]: *assuming g · s* ⦃λ*r t*. *g s* ∧ *r* = *Result* () ∧ *t* = *s*⦄
⟨*proof*⟩

lemma *runs-to-assuming-iff*[*runs-to-iff*]: *assuming g · s* ⦃*Q*⦄ ⟷ (*g s* ⟶ *Q* (*Result* ()) *s*)
⟨*proof*⟩

lemma *runs-to-partial-assuming*[*runs-to-vcg*]:
assuming g · s ?⦃λ*r t*. *g s* ∧ *r* = *Result* () ∧ *t* = *s* ∧ *g s*⦄
⟨*proof*⟩

lemma *assuming-state-assume*:
assuming P = *assume-result-and-state* (λ*s*. (*if P s then* {((), *s*)} *else* {}))
⟨*proof*⟩

lemma *assuming-True*[*simp*]: *assuming* (λ*s*. *True*) = *skip*
⟨*proof*⟩

lemma *refines-assuming*:

$(P\ s \Longrightarrow Q\ t) \Longrightarrow (P\ s \Longrightarrow Q\ t \Longrightarrow R\ (Result\ (),\ s)\ (Result\ (),\ t)) \Longrightarrow$
refines (*assuming* P) (*assuming* Q) $s\ t\ R$
⟨*proof*⟩

lemma *rel-spec-assuming*:
$(Q\ t \longleftrightarrow P\ s) \Longrightarrow (P\ s \Longrightarrow Q\ t \Longrightarrow R\ (Result\ (),\ s)\ (Result\ (),\ t)) \Longrightarrow$
rel-spec (*assuming* P) (*assuming* Q) $s\ t\ R$
⟨*proof*⟩

lemma *refines-bind-assuming-right*:
$P\ t \Longrightarrow (P\ t \Longrightarrow refines\ f\ (g\ ())\ s\ t\ R) \Longrightarrow refines\ f\ (assuming\ P \gg\!= g)\ s\ t\ R$
⟨*proof*⟩

lemma *refines-bind-assuming-left*:
$(P\ s \Longrightarrow refines\ (f\ ())\ g\ s\ t\ R) \Longrightarrow refines\ (assuming\ P >>\!= f)\ g\ s\ t\ R$
⟨*proof*⟩

16.9.24 *guard*

lemma *run-guard*[*run-spec-monad*]:
$run\ (guard\ g)\ s = (if\ g\ s\ then\ pure\text{-}post\text{-}state\ (Result\ (),\ s)\ else\ \top)$
⟨*proof*⟩

lemma *always-progress-guard*[*always-progress-intros*]: *always-progress* (*guard* g)
⟨*proof*⟩

lemma *runs-to-guard*[*runs-to-vcg*]: $g\ s \Longrightarrow guard\ g \cdot s\ \{\!|\lambda r\ t.\ r = Result\ () \wedge t = s|\!\}$
⟨*proof*⟩

lemma *runs-to-guard-iff*[*runs-to-iff*]: $guard\ g \cdot s\ \{\!|Q|\!\} \longleftrightarrow (g\ s \wedge Q\ (Result\ ())\ s)$
⟨*proof*⟩

lemma *runs-to-partial-guard*[*runs-to-vcg*]: $guard\ g \cdot s\ ?\{\!|\lambda r\ t.\ r = Result\ () \wedge t = s \wedge g\ s|\!\}$
⟨*proof*⟩

lemma *refines-guard*:
$(Q\ t \Longrightarrow P\ s) \Longrightarrow (P\ s \Longrightarrow Q\ t \Longrightarrow R\ (Result\ (),\ s)\ (Result\ (),\ t)) \Longrightarrow$
refines (*guard* P) (*guard* Q) $s\ t\ R$
⟨*proof*⟩

lemma *rel-spec-guard*:
$(Q\ t \longleftrightarrow P\ s) \Longrightarrow (P\ s \Longrightarrow Q\ t \Longrightarrow R\ (Result\ (),\ s)\ (Result\ (),\ t)) \Longrightarrow$
rel-spec (*guard* P) (*guard* Q) $s\ t\ R$
⟨*proof*⟩

lemma *refines-bind-guard-right*:
refines f (*guard* $P \gg\!= g$) $s\ t\ R$ **if** $P\ t \Longrightarrow refines\ f\ (g\ ())\ s\ t\ R$

⟨*proof*⟩

lemma *guard-False-fail*: *guard* (λ-. *False*) = *fail*
⟨*proof*⟩

lemma *rel-spec-monad-bind-guard*:
shows (⋀*x*. *rel-spec-monad* (=) *Q* (*f x*) (*g x*)) ⟹
rel-spec-monad (=) *Q* (*guard P* >>= *f*) (*guard P* >>= *g*)
⟨*proof*⟩

lemma *runs-to-guard-bind-iff*: ((*guard P* >>= *f*) · *s* ⦃ *Q* ⦄) ⟷ *P s* ∧ ((*f* ()) · *s* ⦃ *Q* ⦄)
⟨*proof*⟩

lemma *refines-bind-guard-right-iff*:
refines f (*guard P* ⤜ *g*) *s t R* ⟷ (*P t* ⟶ *refines f* (*g* ()) *s t R*)
⟨*proof*⟩

lemma *refines-bind-guard-right-end*:
assumes *f-g*: *refines f g s t R*
shows *refines f* (*do* {*res* <− *g*; *guard G*; *return res*}) *s t*
(λ(*r*, *s*) (*q*, *t*). *R* (*r*, *s*) (*q*, *t*) ∧
(*case q of Exception e* ⇒ *True* | *Result* - ⇒ *G t*))
⟨*proof*⟩

lemma *refines-bind-guard-right-end′*:
assumes *f-g*: *refines f g s t R*
shows *refines f* (*do* {*res* <− *g*; *guard* (*G res*); *return res*}) *s t*
(λ(*r*, *s*) (*q*, *t*). *R* (*r*, *s*) (*q*, *t*) ∧
(*case q of Exception e* ⇒ *True* | *Result v* ⇒ *G v t*))
⟨*proof*⟩

16.9.25 *assert-opt*

lemma *run-assert-opt*[*run-spec-monad*, *simp*]:
run (*assert-opt x*) *s* = (*case x of Some v* ⇒ *pure-post-state* (*Result v*, *s*) | *None* ⇒ ⊤)
⟨*proof*⟩

lemma *always-progress-assert-opt*[*always-progress-intros*]: *always-progress* (*assert-opt x*)
⟨*proof*⟩

lemma *runs-to-assert-opt*[*runs-to-vcg*]: (∃ *v*. *x* = *Some v* ∧ *Q* (*Result v*) *s*) ⟹ *assert-opt x* · *s* ⦃*Q*⦄
⟨*proof*⟩

lemma *runs-to-assert-opt-iff*[*runs-to-iff*]:
assert-opt x · *s* ⦃*Q*⦄ ⟷ (∃ *v*. *x* = *Some v* ∧ *Q* (*Result v*) *s*)

⟨*proof*⟩

lemma *runs-to-partial-assert-opt*[*runs-to-vcg*]:
(⋀*v*. *x* = *Some v* ⟹ *Q* (*Result v*) *s*) ⟹ *assert-opt x* · *s* ?⦃ *Q* ⦄
⟨*proof*⟩

16.9.26 *gets-the*

lemma *run-gets-the′*:
run (*gets-the f*) *s* = (*case f s of Some v* ⇒ *pure-post-state* (*Result v*, *s*) | *None* ⇒ ⊤)
⟨*proof*⟩

lemma *run-gets-the*[*run-spec-monad*, *simp*]:
run (*gets-the f*) *s* = (*case* (*f s*) *of Some v* ⇒ *pure-post-state* (*Result v*, *s*) | *None* ⇒ ⊤)
⟨*proof*⟩

lemma *always-progress-gets-the*[*always-progress-intros*]: *always-progress* (*gets-the f*)
⟨*proof*⟩

lemma *runs-to-gets-the*[*runs-to-vcg*]: (∃ *v*. *f s* = *Some v* ∧ *Q* (*Result v*) *s*) ⟹ *gets-the f* · *s* ⦃ *Q* ⦄
⟨*proof*⟩

lemma *runs-to-gets-the-iff*[*runs-to-iff*]:
gets-the f · *s* ⦃ *Q* ⦄ ⟷ (∃ *v*. *f s* = *Some v* ∧ *Q* (*Result v*) *s*)
⟨*proof*⟩

lemma *runs-to-partial-gets-the*[*runs-to-vcg*]:
(⋀*v*. *f s* = *Some v* ⟹ *Q* (*Result v*) *s*) ⟹ *gets-the f* · *s* ?⦃ *Q* ⦄
⟨*proof*⟩

16.9.27 *modify*

lemma *run-modify*[*run-spec-monad*, *simp*]: *run* (*modify f*) *s* = *pure-post-state* (*Result* (), *f s*)
⟨*proof*⟩

lemma *always-progress-modifiy*[*always-progress-intros*]: *always-progress* (*modify f*)
⟨*proof*⟩

lemma *runs-to-modify*[*runs-to-vcg*]: *modify f* · *s* ⦃ λ*r t*. *r* = *Result* () ∧ *t* = *f s* ⦄
⟨*proof*⟩

lemma *runs-to-modify-res*[*runs-to-vcg*]: ((*modify f*)::(*unit*, *'s*) *res-monad*) · *s* ⦃ λ*r t*. *t* = *f s* ⦄
⟨*proof*⟩

lemma *runs-to-modify-iff*[*runs-to-iff*]: *modify f* · *s* {‖*Q*‖} ⟷ *Q* (*Result* ()) (*f s*)
⟨*proof*⟩

lemma *runs-to-partial-modify*[*runs-to-vcg*]: *modify f* · *s* ?{‖ λ*r t*. *r* = *Result* () ∧ *t* = *f s* ‖}
⟨*proof*⟩

lemma *runs-to-partial-modify-res*[*runs-to-vcg*]:
((*modify f*)::(*unit*, '*s*) *res-monad*) · *s* ?{‖ λ*r t*. *t* = *f s* ‖}
⟨*proof*⟩

lemma *refines-modify*:
R (*Result* (), *f s*) (*Result* (), *g t*) ⟹ *refines* (*modify f*) (*modify g*) *s t R*
⟨*proof*⟩

lemma *rel-spec-modify*:
R (*Result* (), *f s*) (*Result* (), *g t*) ⟹ *rel-spec* (*modify f*) (*modify g*) *s t R*
⟨*proof*⟩

16.9.28 condition

lemma *run-condition*[*run-spec-monad*]: *run* (*condition c f g*) *s* = (*if c s then run f s else run g s*)
⟨*proof*⟩

lemma *run-condition-True*[*run-spec-monad*, *simp*]: *c s* ⟹ *run* (*condition c f g*) *s* = *run f s*
⟨*proof*⟩

lemma *run-condition-False*[*run-spec-monad*, *simp*]: ¬*c s* ⟹ *run* (*condition c f g*) *s* = *run g s*
⟨*proof*⟩

lemma *always-progress-condition*[*always-progress-intros*]:
always-progress f ⟹ *always-progress g* ⟹ *always-progress* (*condition c f g*)
⟨*proof*⟩

lemma *condition-swap*: (*condition C A B*) = (*condition* (λ*s*. ¬ *C s*) *B A*)
⟨*proof*⟩

lemma *condition-fail-rhs*: (*condition C X fail*) = (*guard C* >>= (λ-. *X*))
⟨*proof*⟩

lemma *condition-fail-lhs*: (*condition C fail X*) = (*guard* (λ*s*. ¬ *C s*) >>= (λ-. *X*))
⟨*proof*⟩

lemma *condition-bind-fail*[*simp*]:
(*condition C A B* >>= (λ-. *fail*)) = *condition C* (*A* >>= (λ-. *fail*)) (*B* >>= (λ-. *fail*))

⟨*proof*⟩

lemma *condition-True*[*simp*]: *condition* (λ-. *True*) *f g* = *f*
⟨*proof*⟩

lemma *condition-False*[*simp*]: *condition* (λ-. *False*) *f g* = *g*
⟨*proof*⟩

lemma *le-condition-runI*:
(⋀*s*. *c s* ⟹ *run h s* ≤ *run f s*) ⟹ (⋀*s*. ¬ *c s* ⟹ *run h s* ≤ *run g s*)
⟹ *h* ≤ *condition c f g*
⟨*proof*⟩

lemma *mono-condition-spec-monad*:
mono T ⟹ *mono F* ⟹ *mono* (λ*x*. *condition C* (*F x*) (*T x*))
⟨*proof*⟩

lemma *mono-condition*: *f* ≤ *f'* ⟹ *g* ≤ *g'* ⟹ *condition c f g* ≤ *condition c f' g'*
⟨*proof*⟩

lemma *monotone-condition-le*[*partial-function-mono*]:
monotone R (≤) (λ*f'*. *f f'*) ⟹ (*monotone R* (≤) (λ*f'*. *g f'*))
⟹ *monotone R* (≤) (λ*f'*. *condition c* (*f f'*) (*g f'*))
⟨*proof*⟩

lemma *monotone-condition-ge*[*partial-function-mono*]:
monotone R (≥) (λ*f'*. *f f'*) ⟹ (*monotone R* (≥) (λ*f'*. *g f'*))
⟹ *monotone R* (≥) (λ*f'*. *condition c* (*f f'*) (*g f'*))
⟨*proof*⟩

lemma *runs-to-condition*[*runs-to-vcg*]:
(*c s* ⟹ *f* · *s* {‖ *Q* ‖}) ⟹ (¬ *c s* ⟹ *g* · *s* {‖ *Q* ‖}) ⟹ *condition c f g* · *s* {‖ *Q* ‖}
⟨*proof*⟩

lemma *runs-to-condition-iff*[*runs-to-iff*]:
condition c f g · *s* {‖ *Q* ‖} ⟷ (*if c s then f* · *s* {‖ *Q* ‖} *else g* · *s* {‖ *Q* ‖})
⟨*proof*⟩

lemma *runs-to-partial-condition*[*runs-to-vcg*]:
(*c s* ⟹ *f* · *s* ?{‖ *Q* ‖}) ⟹ (¬ *c s* ⟹ *g* · *s* ?{‖ *Q* ‖}) ⟹ *condition c f g* · *s* ?{‖ *Q* ‖}
⟨*proof*⟩

lemma *refines-condition-iff*:
assumes *c' s'* ⟷ *c s*
shows *refines* (*condition c f g*) (*condition c' f' g'*) *s s' R* ⟷
(*if c' s' then refines f f' s s' R else refines g g' s s' R*)
⟨*proof*⟩

lemma *refines-condition*:
$P\ s \longleftrightarrow P'\ s' \Longrightarrow$
$(P\ s \Longrightarrow P'\ s' \Longrightarrow \mathit{refines}\ f\ f'\ s\ s'\ R) \Longrightarrow$
$(\neg\ P\ s \Longrightarrow \neg\ P'\ s' \Longrightarrow \mathit{refines}\ g\ g'\ s\ s'\ R) \Longrightarrow$
$\mathit{refines}\ (\mathit{condition}\ P\ f\ g)\ (\mathit{condition}\ P'\ f'\ g')\ s\ s'\ R$
⟨*proof*⟩

lemma *refines-condition-TrueI*:
assumes $c'\ s' = c\ s$ **and** $c'\ s'\ \mathit{refines}\ f\ f'\ s\ s'\ R$
shows $\mathit{refines}\ (\mathit{condition}\ c\ f\ g)\ (\mathit{condition}\ c'\ f'\ g')\ s\ s'\ R$
⟨*proof*⟩

lemma *refines-condition-FalseI*:
assumes $c'\ s' = c\ s$ **and** $\neg c'\ s'\ \mathit{refines}\ g\ g'\ s\ s'\ R$
shows $\mathit{refines}\ (\mathit{condition}\ c\ f\ g)\ (\mathit{condition}\ c'\ f'\ g')\ s\ s'\ R$
⟨*proof*⟩

lemma *refines-condition-bind-left*:
$\mathit{refines}\ (\mathit{condition}\ C\ T\ F \gg\!= X)\ Y\ s\ t\ R \longleftrightarrow$
$(C\ s \longrightarrow \mathit{refines}\ (T \gg\!= X)\ Y\ s\ t\ R) \wedge (\neg\ C\ s \longrightarrow \mathit{refines}\ (F \gg\!= X)\ Y\ s\ t\ R)$
⟨*proof*⟩

lemma *refines-condition-bind-right*:
$\mathit{refines}\ X\ (\mathit{condition}\ C\ T\ F \gg\!= Y)\ s\ t\ R \longleftrightarrow$
$(C\ t \longrightarrow \mathit{refines}\ X\ (T \gg\!= Y)\ s\ t\ R) \wedge (\neg\ C\ t \longrightarrow \mathit{refines}\ X\ (F \gg\!= Y)\ s\ t\ R)$
⟨*proof*⟩

lemma *rel-spec-condition-iff*:
assumes $c'\ s' \longleftrightarrow c\ s$
shows $\mathit{rel\text{-}spec}\ (\mathit{condition}\ c\ f\ g)\ (\mathit{condition}\ c'\ f'\ g')\ s\ s'\ R \longleftrightarrow$
$(\mathit{if}\ c'\ s'\ \mathit{then}\ \mathit{rel\text{-}spec}\ f\ f'\ s\ s'\ R\ \mathit{else}\ \mathit{rel\text{-}spec}\ g\ g'\ s\ s'\ R)$
⟨*proof*⟩

lemma *rel-spec-condition*:
$P\ s \longleftrightarrow P'\ s' \Longrightarrow$
$(P\ s \Longrightarrow P'\ s' \Longrightarrow \mathit{rel\text{-}spec}\ f\ f'\ s\ s'\ R) \Longrightarrow$
$(\neg\ P\ s \Longrightarrow \neg\ P'\ s' \Longrightarrow \mathit{rel\text{-}spec}\ g\ g'\ s\ s'\ R) \Longrightarrow$
$\mathit{rel\text{-}spec}\ (\mathit{condition}\ P\ f\ g)\ (\mathit{condition}\ P'\ f'\ g')\ s\ s'\ R$
⟨*proof*⟩

lemma *rel-spec-condition-TrueI*:
assumes $c'\ s' = c\ s$ **and** $c'\ s'\ \mathit{rel\text{-}spec}\ f\ f'\ s\ s'\ R$
shows $\mathit{rel\text{-}spec}\ (\mathit{condition}\ c\ f\ g)\ (\mathit{condition}\ c'\ f'\ g')\ s\ s'\ R$
⟨*proof*⟩

lemma *rel-spec-condition-FalseI*:
assumes $c'\ s' = c\ s$ **and** $\neg c'\ s'\ \mathit{rel\text{-}spec}\ g\ g'\ s\ s'\ R$
shows $\mathit{rel\text{-}spec}\ (\mathit{condition}\ c\ f\ g)\ (\mathit{condition}\ c'\ f'\ g')\ s\ s'\ R$
⟨*proof*⟩

lemma *refines-condition-left*:
(*P s* $\Longrightarrow$ *refines f h s t R*) $\Longrightarrow$ ($\neg$ *P s* $\Longrightarrow$ *refines g h s t R*) $\Longrightarrow$
refines (*condition P f g*) *h s t R*
⟨*proof*⟩

lemma *rel-spec-condition-left*:
(*P s* $\Longrightarrow$ *rel-spec f h s t R*) $\Longrightarrow$ ($\neg$ *P s* $\Longrightarrow$ *rel-spec g h s t R*) $\Longrightarrow$
rel-spec (*condition P f g*) *h s t R*
⟨*proof*⟩

lemma *refines-condition-true*:
P t $\Longrightarrow$ *refines f g s t R* $\Longrightarrow$ *refines f* (*condition P g h*) *s t R*
⟨*proof*⟩

lemma *rel-spec-condition-true*:
P t $\Longrightarrow$ *rel-spec f g s t R* $\Longrightarrow$
rel-spec f (*condition P g h*) *s t R*
⟨*proof*⟩

lemma *refines-condition-false*:
$\neg$ *P t* $\Longrightarrow$ *refines f h s t R* $\Longrightarrow$
refines f (*condition P g h*) *s t R*
⟨*proof*⟩

lemma *rel-spec-condition-false*:
$\neg$ *P t* $\Longrightarrow$ *rel-spec f h s t R* $\Longrightarrow$
rel-spec f (*condition P g h*) *s t R*
⟨*proof*⟩

lemma *condition-bind*:
(*condition P f g* >>= *h*) = *condition P* (*f* >>= *h*) (*g* >>= *h*)
⟨*proof*⟩

lemma *rel-spec-monad-condition*:
assumes *rel-fun R* (=) *P P′*
and *rel-spec-monad R Q f f′*
and *rel-spec-monad R Q g g′*
shows *rel-spec-monad R Q* (*condition P f g*) (*condition P′ f′ g′*)
⟨*proof*⟩

lemma *rel-spec-monad-condition-const*:
P $\longleftrightarrow$ *P′* $\Longrightarrow$ (*P* $\Longrightarrow$ *rel-spec-monad R Q f f′*) $\Longrightarrow$
($\neg$ *P* $\Longrightarrow$ *rel-spec-monad R Q g g′*) $\Longrightarrow$
rel-spec-monad R Q (*condition* (λ-. *P*) *f g*) (*condition* (λ-. *P′*) *f′ g′*)
⟨*proof*⟩

16.9.29 *when*

lemma *run-when*[*run-spec-monad*]:
run (*when c f*) *s* = (*if c then run f s else pure-post-state* (*Result* (), *s*))
⟨*proof*⟩

lemma *always-progress-when*[*always-progress-intros*]:
always-progress f ⟹ *always-progress* (*when c f*)
⟨*proof*⟩

lemma *runs-to-when*[*runs-to-vcg*]:
(*c* ⟹ *f* · *s* ⦃ *Q* ⦄) ⟹ (¬ *c* ⟹ *Q* (*Result* ()) *s*) ⟹ *when c f* · *s* ⦃ *Q* ⦄
⟨*proof*⟩

lemma *runs-to-when-iff*[*runs-to-iff*]:
(*when c f*) · *s* ⦃ *Q* ⦄ ⟷ (*if c then f* · *s* ⦃ *Q* ⦄ *else Q* (*Result* ()) *s*)
⟨*proof*⟩

lemma *runs-to-partial-when*[*runs-to-vcg*]:
(*c* ⟹ *f* · *s* ?⦃ *Q* ⦄) ⟹ (¬ *c* ⟹ *Q* (*Result* ()) *s*) ⟹ *when c f* · *s* ?⦃ *Q* ⦄
⟨*proof*⟩

lemma *mono-when*: *f* ≤ *f'* ⟹ *when c f* ≤ *when c f'*
⟨*proof*⟩

lemma *monotone-when-le*[*partial-function-mono*]:
monotone R (≤) (λ*f'*. *f f'*)
⟹ *monotone R* (≤) (λ*f'*. *when c* (*f f'*))
⟨*proof*⟩

lemma *monotone-when-ge*[*partial-function-mono*]:
monotone R (≥) (λ*f'*. *f f'*)
⟹ *monotone R* (≥) (λ*f'*. *when c* (*f f'*))
⟨*proof*⟩

lemma *when-True*[*simp*]: *when True f* = *f*
⟨*proof*⟩

lemma *when-False*[*simp*]: *when False f* = *return* ()
⟨*proof*⟩

16.9.30 While

context fixes *C* :: *'a* ⇒ *'s* ⇒ *bool* **and** *B* :: *'a* ⇒ (*'e*::*default*, *'a*, *'s*) *spec-monad*
begin

definition *whileLoop* :: *'a* ⇒ (*'e*, *'a*, *'s*) *spec-monad* **where**
whileLoop =
gfp (λ*W a*. *condition* (*C a*) (*bind* (*B a*) *W*) (*return a*))
— Collapses to *Failure* in case of any non terminating computation.

definition *whileLoop-finite* :: *'a* ⇒ (*'e*, *'a*, *'s*) *spec-monad* **where**
whileLoop-finite =
lfp (λ*W a*. *condition* (*C a*) (*bind* (*B a*) *W*) (*return a*))
— Does not collapse to *Failure* in presence of a non terminating computation. *Failure* can still occur when the body fails in some iteration. It captures the outcomes of all terminating and thus finite computations.

inductive *whileLoop-terminates* :: *'a* ⇒ *'s* ⇒ *bool* **where**
step: ⋀*a s*. (*C a s* ⟹ *B a* ∙ *s* ?⦃ λ*v s*. ∀ *a*. *v* = *Result a* ⟶ *whileLoop-terminates a s* ⦄) ⟹
whileLoop-terminates a s
— This is weaker than *run* (*whileLoop a*) *s* ≠ ⊤: as it uses partial correctness

lemma *mono-whileLoop-functional*:
mono (λ*W a*. *condition* (*C a*) (*bind* (*B a*) *W*) (*return a*))
⟨*proof*⟩

lemma *whileLoop-unroll*:
whileLoop a =
condition (*C a*) (*bind* (*B a*) *whileLoop*) (*return a*)
⟨*proof*⟩

lemma *whileLoop-finite-unfold*:
whileLoop-finite a =
condition (*C a*) (*bind* (*B a*) *whileLoop-finite*) (*return a*)
⟨*proof*⟩

lemma *whileLoop-ne-Failure*:
(*C a s* ⟹ *B a* ∙ *s* ⦃ λ*x s*. ∀ *a*. *x* = *Result a* ⟶ *run* (*whileLoop a*) *s* ≠ *Failure* ⦄) ⟹
run (*whileLoop a*) *s* ≠ *Failure*
⟨*proof*⟩

lemma *whileLoop-ne-top-induct*[*consumes 1*, *case-names step*]:
assumes *a-s*: *run* (*whileLoop a*) *s* ≠ ⊤
and *step*: ⋀*a s*. (*C a s* ⟹ *B a* ∙ *s* ⦃ λ*x s*. ∀ *a*. *x* = *Result a* ⟶ *P a s* ⦄) ⟹ *P a s*
shows *P a s*
⟨*proof*⟩

lemma *runs-to-whileLoop*:
assumes *R*: *wf R*
assumes ∗: *I* (*Result a*) *s*
assumes *P-Result*: ⋀*a s*. ¬ *C a s* ⟹ *I* (*Result a*) *s* ⟹ *P* (*Result a*) *s*
assumes *P-Exception*: ⋀*a s*. *a* ≠ *default* ⟹ *I* (*Exception a*) *s* ⟹ *P* (*Exception a*) *s*
assumes *B*: ⋀*a s*. *C a s* ⟹ *I* (*Result a*) *s* ⟹
B a ∙ *s* ⦃λ*r t*. *I r t* ∧ (∀ *b*. *r* = *Result b* ⟶ ((*b*, *t*), (*a*, *s*)) ∈ *R*)⦄

shows *whileLoop a · s* {|*P*|}
⟨*proof*⟩

lemma *runs-to-whileLoop-finite*:
assumes ∗: *I* (*Result a*) *s*
assumes *P-Result*: ⋀*a s*. ¬ *C a s* ⟹ *I* (*Result a*) *s* ⟹ *P* (*Result a*) *s*
assumes *P-Exception*: ⋀*a s*. *a* ≠ *default* ⟹ *I* (*Exception a*) *s* ⟹ *P* (*Exception a*) *s*
assumes *B*: ⋀*a s*. *C a s* ⟹ *I* (*Result a*) *s* ⟹ *B a · s* {| *I* |}
shows *whileLoop-finite a · s* {|*P*|}
⟨*proof*⟩

lemma *runs-to-partial-whileLoop-finite*:
assumes ∗: *I* (*Result a*) *s*
assumes *B*: ⋀*a s*. *C a s* ⟹ *I* (*Result a*) *s* ⟹ (*B a*) · *s* *?*{| *I* |}
assumes *P-Result*: ⋀*a s*. ¬ *C a s* ⟹ *I* (*Result a*) *s* ⟹ *P* (*Result a*) *s*
assumes *P-Exception*: ⋀*a s*. *a* ≠ *default* ⟹ *I* (*Exception a*) *s* ⟹ *P* (*Exception a*) *s*
shows *whileLoop-finite a · s* *?*{| *P* |}
⟨*proof*⟩

lemma *whileLoop-finite-eq-whileLoop-of-whileLoop-terminates*:
assumes *whileLoop-terminates a s*
shows *run* (*whileLoop-finite a*) *s* = *run* (*whileLoop a*) *s*
⟨*proof*⟩

lemma *whileLoop-terminates-of-succeeds*:
run (*whileLoop a*) *s* ≠ ⊤ ⟹ *whileLoop-terminates a s*
⟨*proof*⟩

lemma *whileLoop-finite-eq-whileLoop*:
run (*whileLoop a*) *s* ≠ ⊤ ⟹ *run* (*whileLoop-finite a*) *s* = *run* (*whileLoop a*) *s*
⟨*proof*⟩

lemma *runs-to-whileLoop-of-runs-to-whileLoop-finite-if-terminates*:
whileLoop-terminates i s ⟹ *whileLoop-finite i · s* {|*Q*|} ⟹ *whileLoop i · s* {|*Q*|}
⟨*proof*⟩

lemma *whileLoop-finite-le-whileLoop*: *whileLoop-finite a* ≤ *whileLoop a*
⟨*proof*⟩

lemma *runs-to-partial-whileLoop-finite-whileLoop*:
whileLoop-finite i · s *?*{|*Q*|} ⟹ *whileLoop i · s* *?*{|*Q*|}
⟨*proof*⟩

lemma *runs-to-partial-whileLoop*:
assumes *I* (*Result a*) *s*
assumes ⋀*a s*. ¬ *C a s* ⟹ *I* (*Result a*) *s* ⟹ *P* (*Result a*) *s*
assumes ⋀*a s*. *a* ≠ *default* ⟹ *I* (*Exception a*) *s* ⟹ *P* (*Exception a*) *s*

assumes $\bigwedge a\ s.\ C\ a\ s \Longrightarrow I\ (Result\ a)\ s \Longrightarrow (B\ a) \cdot s\ ?\{\!|\ I\ |\!\}$
shows $whileLoop\ a \cdot s\ ?\{\!|\ P\ |\!\}$
⟨*proof*⟩

lemma *always-progress-whileLoop*[*always-progress-intros*]:
assumes B: $(\bigwedge v.\ always\text{-}progress\ (B\ v))$
shows $always\text{-}progress\ (whileLoop\ a)$
⟨*proof*⟩

lemma *runs-to-partial-whileLoop-cond-false*:
$(whileLoop\ I) \cdot s\ ?\{\!|\ \lambda r\ t.\ \forall a.\ r = Result\ a \longrightarrow \neg\ C\ a\ t\ |\!\}$
⟨*proof*⟩

end

context
fixes R
fixes $C :: 'a \Rightarrow 's \Rightarrow bool$ **and** $B :: 'a \Rightarrow ('e::default, 'a, 's)\ spec\text{-}monad$
and $C' :: 'b \Rightarrow 't \Rightarrow bool$ **and** $B' :: 'b \Rightarrow ('f::default, 'b, 't)\ spec\text{-}monad$
assumes C: $\bigwedge x\ s\ x'\ s'.\ R\ (Result\ x, s)\ (Result\ x', s') \Longrightarrow C\ x\ s \longleftrightarrow C'\ x'\ s'$
assumes B: $\bigwedge x\ s\ x'\ s'.\ R\ (Result\ x, s)\ (Result\ x', s') \Longrightarrow C\ x\ s \Longrightarrow C'\ x'\ s' \Longrightarrow$
$refines\ (B\ x)\ (B'\ x')\ s\ s'\ R$
assumes R: $\bigwedge v\ s\ v'\ s'.\ R\ (v, s)\ (v', s') \Longrightarrow (\exists x.\ v = Result\ x) \longleftrightarrow (\exists x'.\ v' = Result\ x')$
begin

lemma *refines-whileLoop-finite-strong*:
assumes *x-x'*: $R\ (Result\ x, s)\ (Result\ x', s')$
shows $refines\ (whileLoop\text{-}finite\ C\ B\ x)\ (whileLoop\text{-}finite\ C'\ B'\ x')\ s\ s'$
$(\lambda(r, s)\ (r', s').\ (\forall v.\ r = Result\ v \longrightarrow (\exists v'.\ r' = Result\ v' \wedge \neg\ C\ v\ s \wedge \neg\ C'\ v'\ s')) \wedge$
$R\ (r, s)\ (r', s'))$
(**is** *refines - - - - ?R*)
⟨*proof*⟩

lemma *refines-whileLoop-finite*:
assumes *x-x'*: $R\ (Result\ x, s)\ (Result\ x', s')$
shows $refines\ (whileLoop\text{-}finite\ C\ B\ x)\ (whileLoop\text{-}finite\ C'\ B'\ x')\ s\ s'\ R$
⟨*proof*⟩

lemma *whileLoop-succeeds-terminates-of-refines*:
assumes $run\ (whileLoop\ C'\ B'\ x')\ s' \neq \top$
shows $R\ (Result\ x, s)\ (Result\ x', s') \Longrightarrow run\ (whileLoop\ C\ B\ x)\ s \neq Failure$
⟨*proof*⟩

lemma *refines-whileLoop-strong*:
assumes *x-x'*: $R\ (Result\ x, s)\ (Result\ x', s')$
shows $refines\ (whileLoop\ C\ B\ x)\ (whileLoop\ C'\ B'\ x')\ s\ s'$
$(\lambda(r, s)\ (r', s').\ (\forall v.\ r = Result\ v \longrightarrow (\exists v'.\ r' = Result\ v' \wedge \neg\ C\ v\ s \wedge \neg\ C'$

$v'\ s'))\ \wedge$
$R\ (r,\ s)\ (r',\ s'))$
⟨*proof*⟩

lemma *refines-whileLoop*:
assumes *x-x'*: $R\ (Result\ x,\ s)\ (Result\ x',\ s')$
shows *refines* $(whileLoop\ C\ B\ x)\ (whileLoop\ C'\ B'\ x')\ s\ s'\ R$
⟨*proof*⟩

end

lemma *runs-to-whileLoop-res'*:
assumes *R*: *wf R*
assumes ∗: $I\ a\ s$
assumes *P-Result*: $\bigwedge a\ s.\ \neg\ C\ a\ s \Longrightarrow I\ a\ s \Longrightarrow P\ (Result\ a)\ s$
assumes *B*: $\bigwedge a\ s.\ C\ a\ s \Longrightarrow I\ a\ s \Longrightarrow$
$B\ a\ \cdot\ s\ \{\!|\lambda r\ t.\ (\forall\ b.\ r = Result\ b \longrightarrow I\ b\ t \wedge ((b,\ t),\ (a,\ s)) \in R)|\!\}$
shows $(whileLoop\ C\ B\ a{::}('a,\ 's)\ res\text{-}monad) \cdot s\ \{\!|P|\!\}$
⟨*proof*⟩

lemma *runs-to-whileLoop-res*:
assumes *B*: $\bigwedge a\ s.\ C\ a\ s \Longrightarrow I\ a\ s \Longrightarrow$
$B\ a\ \cdot\ s\ \{\!|\lambda Res\ r\ t.\ I\ r\ t \wedge ((r,\ t),\ (a,\ s)) \in R|\!\}$
assumes *P-Result*: $\bigwedge a\ s.\ I\ a\ s \Longrightarrow \neg\ C\ a\ s \Longrightarrow P\ a\ s$
assumes *R*: *wf R*
assumes ∗: $I\ a\ s$
shows $(whileLoop\ C\ B\ a{::}('a,\ 's)\ res\text{-}monad) \cdot s\ \{\!|\lambda Res\ r.\ P\ r|\!\}$
⟨*proof*⟩

lemma *runs-to-whileLoop-variant-res*:
assumes *I*: $\bigwedge r\ s\ c.\ I\ r\ s\ c \Longrightarrow$
$C\ r\ s \Longrightarrow (B\ r) \cdot s\ \{\!|\ \lambda Res\ q\ t.\ \exists\ c'.\ I\ q\ t\ c' \wedge (c',\ c) \in R\ |\!\}$
assumes *Q*: $\bigwedge r\ s\ c.\ I\ r\ s\ c \Longrightarrow \neg\ C\ r\ s \Longrightarrow Q\ r\ s$
assumes *R*: *wf R*
shows $I\ r\ s\ c \Longrightarrow (whileLoop\ C\ B\ r{::}('a,\ 's)\ res\text{-}monad) \cdot s\ \{\!|\lambda Res\ r.\ Q\ r\ |\!\}$
⟨*proof*⟩

lemma *runs-to-whileLoop-inc-res*:
assumes ∗: $\bigwedge i.\ i < M \Longrightarrow (B\ (F\ i)) \cdot (S\ i)\ \{\!|\ \lambda Res\ r'\ t'.\ r' = (F\ (Suc\ i)) \wedge t' = (S\ (Suc\ i))\ |\!\}$
and [*simp*]: $\bigwedge i.\ i \leq M \Longrightarrow C\ (F\ i)\ (S\ i) \longleftrightarrow i < M$
and [*simp*]: $i = F\ 0\ si = S\ 0\ t = F\ M\ st = S\ M$
shows $(whileLoop\ C\ B\ i{::}('a,\ 's)\ res\text{-}monad) \cdot si\ \{\!|\ \lambda Res\ r'\ t'.\ r' = t \wedge t' = st\ |\!\}$
⟨*proof*⟩

lemma *runs-to-whileLoop-dec-res*:
assumes ∗: $\bigwedge i{::}nat.\ i > 0 \Longrightarrow i \leq M \Longrightarrow$
$(B\ (F\ i)) \cdot (S\ i)\ \{\!|\ \lambda Res\ r'\ t'.\ r' = (F\ (i - 1)) \wedge t' = (S\ (i - 1))\ |\!\}$

and [*simp*]: $\bigwedge i.\ C\ (F\ i)\ (S\ i) \longleftrightarrow i > 0$
and [*simp*]: $i = F\ M\ si = S\ M\ t = F\ 0\ st = S\ 0$
shows $(whileLoop\ C\ B\ i{::}('a, 's)\ res\text{-}monad) \cdot si\ \{\!|\ \lambda Res\ r'\ t'.\ r' = t \wedge t' = st\ |\!\}$
⟨*proof*⟩

lemma *runs-to-whileLoop-exn*:
assumes *R*: *wf R*
assumes ∗: $I\ (Result\ a)\ s$
assumes *P-Result*: $\bigwedge a\ s.\ \neg\ C\ a\ s \Longrightarrow I\ (Result\ a)\ s \Longrightarrow P\ (Result\ a)\ s$
assumes *P-Exn*: $\bigwedge a\ s.\ I\ (Exn\ a)\ s \Longrightarrow P\ (Exn\ a)\ s$
assumes *B*: $\bigwedge a\ s.\ C\ a\ s \Longrightarrow I\ (Result\ a)\ s \Longrightarrow$
$B\ a\ \cdot s\ \{\!|\lambda r\ t.\ I\ r\ t \wedge (\forall b.\ r = Result\ b \longrightarrow ((b,\ t),\ (a,\ s)) \in R)|\!\}$
shows $whileLoop\ C\ B\ a \cdot s\ \{\!|P|\!\}$
⟨*proof*⟩

lemma *runs-to-whileLoop-exn′*:
assumes *B*: $\bigwedge a\ s.\ I\ (Result\ a)\ s \Longrightarrow C\ a\ s \Longrightarrow$
$B\ a\ \cdot s\ \{\!|\lambda r\ t.\ I\ r\ t \wedge (\forall b.\ r = Result\ b \longrightarrow ((b,\ t),\ (a,\ s)) \in R)|\!\}$
assumes *P-Result*: $\bigwedge a\ s.\ \ I\ (Result\ a)\ s \Longrightarrow \neg\ C\ a\ s \Longrightarrow P\ (Result\ a)\ s$
assumes *P-Exn*: $\bigwedge a\ s.\ I\ (Exn\ a)\ s \Longrightarrow P\ (Exn\ a)\ s$
assumes *R*: *wf R*
assumes ∗: $I\ (Result\ a)\ s$
shows $whileLoop\ C\ B\ a \cdot s\ \{\!|P|\!\}$
⟨*proof*⟩

lemma *runs-to-partial-whileLoop-res*:
assumes ∗: $I\ a\ s$
assumes *P-Result*: $\bigwedge a\ s.\ \neg\ C\ a\ s \Longrightarrow I\ a\ s \Longrightarrow P\ (Result\ a)\ s$
assumes *B*: $\bigwedge a\ s.\ C\ a\ s \Longrightarrow I\ a\ s \Longrightarrow$
$(B\ a) \cdot s\ ?\{\!|\lambda r\ t.\ (\forall b.\ r = Result\ b \longrightarrow I\ b\ t)|\!\}$
shows $(whileLoop\ C\ B\ a{::}('a, 's)\ res\text{-}monad) \cdot s\ ?\{\!|P|\!\}$
⟨*proof*⟩

lemma *runs-to-partial-whileLoop-exn*:
assumes ∗: $I\ (Result\ a)\ s$
assumes *P-Result*: $\bigwedge a\ s.\ \neg\ C\ a\ s \Longrightarrow I\ (Result\ a)\ s \Longrightarrow P\ (Result\ a)\ s$
assumes *P-Exn*: $\bigwedge a\ s.\ I\ (Exn\ a)\ s \Longrightarrow P\ (Exn\ a)\ s$
assumes *B*: $\bigwedge a\ s.\ C\ a\ s \Longrightarrow I\ (Result\ a)\ s \Longrightarrow$
$(B\ a) \cdot s\ ?\{\!|\lambda r\ t.\ I\ r\ t|\!\}$
shows $whileLoop\ C\ B\ a \cdot s\ ?\{\!|P|\!\}$
⟨*proof*⟩

notation (**output**)
whileLoop (‹(‹*notation*=‹*prefix whileLoop*››*whileLoop* (-)// (-))› [*1000*, *1000*] *1000*)

lemma *whileLoop-mono*: $b \leq b' \Longrightarrow whileLoop\ c\ b\ i \leq whileLoop\ c\ b'\ i$

⟨*proof*⟩

lemma *whileLoop-finite-mono*: $b \leq b' \Longrightarrow$ *whileLoop-finite c b i* $\leq$ *whileLoop-finite c b' i*
⟨*proof*⟩

lemma *monotone-whileLoop-le*[*partial-function-mono*]:
$(\bigwedge x.$ *monotone R* $(\leq)$ $(\lambda f.$ *b f x*$)) \Longrightarrow$ *monotone R* $(\leq)$ $(\lambda f.$ *whileLoop c* $(b\ f)$ *i*$)$
⟨*proof*⟩

lemma *monotone-whileLoop-ge*[*partial-function-mono*]:
$(\bigwedge x.$ *monotone R* $(\geq)$ $(\lambda f.$ *b f x*$)) \Longrightarrow$ *monotone R* $(\geq)$ $(\lambda f.$ *whileLoop c* $(b\ f)$ *i*$)$
⟨*proof*⟩

lemma *monotone-whileLoop-finite-le*[*partial-function-mono*]:
$(\bigwedge x.$ *monotone R* $(\leq)$ $(\lambda f.$ *b f x*$)) \Longrightarrow$ *monotone R* $(\leq)$ $(\lambda f.$ *whileLoop-finite c* $(b\ f)$ *i*$)$
⟨*proof*⟩

lemma *monotone-whileLoop-finite-ge*[*partial-function-mono*]:
$(\bigwedge x.$ *monotone R* $(\geq)$ $(\lambda f.$ *b f x*$)) \Longrightarrow$ *monotone R* $(\geq)$ $(\lambda f.$ *whileLoop-finite c* $(b\ f)$ *i*$)$
⟨*proof*⟩

lemma *run-whileLoop-le-invariant-cong*:
assumes *I*: *I* (*Result i*) *s*
assumes *invariant*: $\bigwedge r\ s.$ *C r s* $\Longrightarrow$ *I* (*Result r*) *s* $\Longrightarrow$ *B r* $\cdot$ *s* $?\{\!|I|\!\}$
assumes *C*: $\bigwedge r\ s.$ *I* (*Result r*) *s* $\Longrightarrow$ *C r s* $=$ *C' r s*
assumes *B*: $\bigwedge r\ s.$ *C r s* $\Longrightarrow$ *I* (*Result r*) *s* $\Longrightarrow$ *run* (*B r*) *s* $=$ *run* (*B' r*) *s*
shows *run* (*whileLoop C B i*) *s* $\leq$ *run* (*whileLoop C' B' i*) *s*
⟨*proof*⟩

lemma *run-whileLoop-invariant-cong*:
assumes *I*: *I* (*Result i*) *s*
assumes *invariant*: $\bigwedge r\ s.$ *C r s* $\Longrightarrow$ *I* (*Result r*) *s* $\Longrightarrow$ *B r* $\cdot$ *s* $?\{\!|I|\!\}$
assumes *C*: $\bigwedge r\ s.$ *I* (*Result r*) *s* $\Longrightarrow$ *C r s* $=$ *C' r s*
assumes *B*: $\bigwedge r\ s.$ *C r s* $\Longrightarrow$ *I* (*Result r*) *s* $\Longrightarrow$ *run* (*B r*) *s* $=$ *run* (*B' r*) *s*
shows *run* (*whileLoop C B i*) *s* $=$ *run* (*whileLoop C' B' i*) *s*
⟨*proof*⟩

lemma *whileLoop-cong*:
assumes *C*: $\bigwedge r\ s.$ *C r s* $=$ *C' r s*
assumes *B*: $\bigwedge r\ s.$ *C r s* $\Longrightarrow$ *run* (*B r*) *s* $=$ *run* (*B' r*) *s*
shows *whileLoop C B* $=$ *whileLoop C' B'*
⟨*proof*⟩

lemma *refines-whileLoop'*:
assumes *C*: $\bigwedge a\ s\ b\ t.$ *R* (*Result a*, *s*) (*Result b*, *t*) $\Longrightarrow$ *C a s* $\longleftrightarrow$ *C' b t*
and *B*:

$\bigwedge a\ s\ b\ t.\ R\ (Result\ a,\ s)\ (Result\ b,\ t) \Longrightarrow C\ a\ s \Longrightarrow C'\ b\ t \Longrightarrow refines\ (B\ a)\ (B'\ b)\ s\ t\ R$
and I: $R\ (Result\ I,\ s)\ (Result\ I',\ s')$
and R:
$\bigwedge r\ s\ r'\ s'.\ R\ (r,\ s)\ (r',\ s') \Longrightarrow rel\text{-}exception\text{-}or\text{-}result\ (\lambda\text{- - .}\ True)\ (\lambda\text{- - .}\ True)\ r\ r'$
shows $refines\ (whileLoop\ C\ B\ I)\ (whileLoop\ C'\ B'\ I')\ s\ s'$
$(\lambda(r,\ s)\ (r',\ s').\ (\forall v.\ r = Result\ v \longrightarrow (\exists v'.\ r' = Result\ v' \wedge \neg\ C\ v\ s \wedge \neg\ C'\ v'\ s')) \wedge$
$R\ (r,\ s)\ (r',\ s'))$
⟨*proof*⟩

lemma *rel-spec-whileLoop'*:
assumes C: $\bigwedge a\ s\ b\ t.\ R\ (Result\ a,\ s)\ (Result\ b,\ t) \Longrightarrow C\ a\ s \longleftrightarrow C'\ b\ t$
and B: $\bigwedge a\ s\ b\ t.\ R\ (Result\ a,\ s)\ (Result\ b,\ t) \Longrightarrow C\ a\ s \Longrightarrow C'\ b\ t \Longrightarrow rel\text{-}spec\ (B\ a)\ (B'\ b)\ s\ t\ R$
and I: $R\ (Result\ I,\ s)\ (Result\ I',\ s')$
and R: $\bigwedge r\ s\ r'\ s'.\ R\ (r,\ s)\ (r',\ s') \Longrightarrow rel\text{-}exception\text{-}or\text{-}result\ (\lambda\text{- - .}\ True)\ (\lambda\text{- - .}\ True)\ r\ r'$
shows $rel\text{-}spec\ (whileLoop\ C\ B\ I)\ (whileLoop\ C'\ B'\ I')\ s\ s'$
$(\lambda(r,\ s)\ (r',\ s').\ (\forall v.\ r = Result\ v \longrightarrow (\exists v'.\ r' = Result\ v' \wedge \neg\ C\ v\ s \wedge \neg\ C'\ v'\ s')) \wedge$
$R\ (r,\ s)\ (r',\ s'))$
⟨*proof*⟩

lemma *rel-spec-whileLoop*:
assumes C: $\bigwedge a\ s\ b\ t.\ R\ (Result\ a,\ s)\ (Result\ b,\ t) \Longrightarrow C\ a\ s \longleftrightarrow C'\ b\ t$
and B: $\bigwedge a\ s\ b\ t.\ R\ (Result\ a,\ s)\ (Result\ b,\ t) \Longrightarrow C\ a\ s \Longrightarrow C'\ b\ t \Longrightarrow rel\text{-}spec\ (B\ a)\ (B'\ b)\ s\ t\ R$
and I: $R\ (Result\ I,\ s)\ (Result\ I',\ s')$
and R: $\bigwedge r\ s\ r'\ s'.\ R\ (r,\ s)\ (r',\ s') \Longrightarrow rel\text{-}exception\text{-}or\text{-}result\ (\lambda\text{- - .}\ True)\ (\lambda\text{- - .}\ True)\ r\ r'$
shows $rel\text{-}spec\ (whileLoop\ C\ B\ I)\ (whileLoop\ C'\ B'\ I')\ s\ s'\ R$
⟨*proof*⟩

lemma *do-whileLoop-combine*:
$do\ \{\ x1 \leftarrow body\ x0;\ whileLoop\ P\ body\ x1\ \} =$
$do\ \{$
$(b,\ y) \leftarrow whileLoop\ (\lambda(b,\ x)\ s.\ b \longrightarrow P\ x\ s)\ (\lambda(b,\ x).\ do\ \{\ y \leftarrow body\ x;\ return\ (True,\ y)\ \})$
$(False,\ x0);$
$return\ y$
$\}$
⟨*proof*⟩

lemma *rel-spec-whileLoop-res*:
assumes C: $\bigwedge a\ s\ b\ t.\ R\ (Result\ a,\ s)\ (Result\ b,\ t) \Longrightarrow C\ a\ s \longleftrightarrow C'\ b\ t$
and B: $\bigwedge a\ s\ b\ t.\ R\ (Result\ a,\ s)\ (Result\ b,\ t) \Longrightarrow C\ a\ s \Longrightarrow C'\ b\ t \Longrightarrow rel\text{-}spec\ (B\ a)\ (B'\ b)\ s\ t\ R$

and I: R (*Result* I, s) (*Result* I', s')
shows *rel-spec* ((*whileLoop* C B I)::('a, 's) *res-monad*) ((*whileLoop* C' B' I')::('b, 't) *res-monad*) s s' R
⟨*proof*⟩

lemma *rel-spec-monad-whileLoop*:
assumes *init*: R I I'
assumes *cond*: $\bigwedge x\ y.\ R\ x\ y \Longrightarrow$ (*rel-fun* S (=)) (C x) (C' y)
assumes *body*: $\bigwedge x\ y.\ R\ x\ y \Longrightarrow$ *rel-spec-monad* S (*rel-exception-or-result* E R) (B x) (B' y)
shows *rel-spec-monad* S (*rel-exception-or-result* E R) (*whileLoop* C B I) (*whileLoop* C' B' I')
⟨*proof*⟩

lemma *rel-spec-monad-whileLoop-res'*:
assumes *init*: R I I'
assumes *cond*: $\bigwedge x\ y.\ R\ x\ y \Longrightarrow$ (*rel-fun* S (=)) (C x) (C' y)
assumes *body*: $\bigwedge x\ y.\ R\ x\ y \Longrightarrow$ *rel-spec-monad* S (λ*Res* x *Res* y. R x y) (B x) (B' y)
shows *rel-spec-monad* S (λ*Res* x *Res* y. R x y)
((*whileLoop* C B I)::('a, 's) *res-monad*)
((*whileLoop* C' B' I')::('b, 't) *res-monad*)
⟨*proof*⟩

lemma *rel-spec-monad-whileLoop-res*:
assumes *init*: R I I'
assumes *cond*: *rel-fun* R (*rel-fun* S (=)) C C'
assumes *body*: *rel-fun* R (*rel-spec-monad* S (λ*Res* x *Res* y. R x y)) B B'
shows *rel-spec-monad* S (λ*Res* x *Res* y. R x y)
((*whileLoop* C B I)::('a, 's) *res-monad*)
((*whileLoop* C' B' I')::('b, 't) *res-monad*)
⟨*proof*⟩

lemma *runs-to-whileLoop-finite-exn*:
assumes B: $\bigwedge r\ s.\ I$ (*Result* r) $s \Longrightarrow C\ r\ s \Longrightarrow (B\ r) \bullet s\ \{\!|\ I\ |\!\}$
assumes Qr: $\bigwedge r\ s.\ I$ (*Result* r) $s \Longrightarrow \neg\ C\ r\ s \Longrightarrow Q$ (*Result* r) s
assumes Ql: $\bigwedge e\ s.\ I$ (*Exn* e) $s \Longrightarrow Q$ (*Exn* e) s
assumes I: I (*Result* r) s
shows (*whileLoop-finite* C B r) $\bullet$ s $\{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *runs-to-whileLoop-finite-res*:
assumes B: $\bigwedge r\ s.\ I\ r\ s \Longrightarrow C\ r\ s \Longrightarrow (B\ r) \bullet s\ \{\!|\lambda$*Res* r. $I\ r\ |\!\}$
assumes Q: $\bigwedge r\ s.\ I\ r\ s \Longrightarrow \neg\ C\ r\ s \Longrightarrow Q\ r\ s$
assumes I: $I\ r\ s$
shows ((*whileLoop-finite* C B r)::('a, 's) *res-monad*) $\bullet$ s $\{\!|\lambda$*Res* r. $Q\ r\ |\!\}$
⟨*proof*⟩

lemma *runs-to-whileLoop-eq-whileLoop-finite*:

run (*whileLoop C B r*) *s* ≠ ⊤ ⟹
(*whileLoop C B r*) · *s* {‖ *Q* ‖} ⟷ (*whileLoop-finite C B r*) · *s* {‖ *Q* ‖}
⟨*proof*⟩

lemma *whileLoop-finite-cond-fail*:
¬ *C r s* ⟹ (*run* (*whileLoop-finite C B r*) *s*) = (*run* (*return r*) *s*)
⟨*proof*⟩

lemma *runs-to-whileLoop-finite-cond-fail*:
¬ *C r s* ⟹ (*whileLoop-finite C B r*) · *s* {‖ *Q* ‖} ⟷ (*return r*) · *s* {‖ *Q* ‖}
⟨*proof*⟩

lemma *runs-to-whileLoop-cond-fail*:
¬ *C r s* ⟹ (*whileLoop C B r*) · *s* {‖ *Q* ‖} ⟷ (*return r*) · *s* {‖ *Q* ‖}
⟨*proof*⟩

lemma *whileLoop-finite-unfold′*:
(*whileLoop-finite C B r*) =
((*condition* (*C r*) (*B r*) (*return r*)) >>= (*whileLoop-finite C B*))
⟨*proof*⟩

lemma *runs-to-whileLoop-unroll*:
assumes ¬ *C r s* ⟹ *P r s*
assumes [*runs-to-vcg*]: *C r s* ⟹ (*B r*) · *s* {‖ λ*Res r t*. ((*whileLoop C B r*) · *t* {‖λ*Res r*. *P r* ‖}) ‖}
shows (*whileLoop C B r*) · *s* {‖λ*Res r*. *P r* ‖}
⟨*proof*⟩

lemma *runs-to-partial-whileLoop-unroll*:
assumes ¬ *C r s* ⟹ *P r s*
assumes *C r s* ⟹ (*B r*) · *s* ?{‖ λ*Res r t*. ((*whileLoop C B r*) · *t* ?{‖λ*Res r*. *P r* ‖}) ‖}
shows (*whileLoop C B r*) · *s* ?{‖λ*Res r*. *P r* ‖}
⟨*proof*⟩

lemma *runs-to-whileLoop-unroll-exn*:
assumes ¬ *C r s* ⟹ *P* (*Result r*) *s*
assumes [*runs-to-vcg*]: *C r s* ⟹ (*B r*) · *s* {‖ λ*r t*.
(∀ *a*. *r* = *Result a* ⟶ ((*whileLoop C B a*) · *t* {‖ *P* ‖})) ∧
(∀ *e*. *r* = *Exn e* ⟶ *P* (*Exn e*) *t*) ‖}
shows (*whileLoop C B r*) · *s* {‖λ*r s′*. *P r s′* ‖}
⟨*proof*⟩

lemma *runs-to-partial-whileLoop-unroll-exn*:
assumes ¬ *C r s* ⟹ *P* (*Result r*) *s*
assumes *C r s* ⟹ (*B r*) · *s* ?{‖ λ*r t*.
(∀ *a*. *r* = *Result a* ⟶ ((*whileLoop C B a*) · *t* ?{‖ *P* ‖})) ∧

$(\forall e.\ r = Exn\ e \longrightarrow P\ (Exn\ e)\ t)\ \}$
shows $(whileLoop\ C\ B\ r) \cdot s\ ?\{\lambda r\ s'.\ P\ r\ s'\ \}$
⟨*proof*⟩

lemma *refines-whileLoop-exn*:
assumes C: $\bigwedge a\ s\ b\ t.\ R\ (Result\ a,\ s)\ (Result\ b,\ t) \Longrightarrow C\ a\ s \longleftrightarrow C'\ b\ t$
and B: $\bigwedge a\ s\ b\ t.\ R\ (Result\ a,\ s)\ (Result\ b,\ t) \Longrightarrow C\ a\ s \Longrightarrow C'\ b\ t \Longrightarrow refines\ (B\ a)\ (B'\ b)\ s\ t\ R$
and I: $R\ (Result\ I,\ s)\ (Result\ I',\ s')$
and *R1*: $\bigwedge a\ s\ b\ t.\ R\ (Exn\ a,\ s)\ (Result\ b,\ t) \Longrightarrow False$
and *R2*: $\bigwedge a\ s\ b\ t.\ R\ (Result\ a,\ s)\ (Exn\ b,\ t) \Longrightarrow False$
shows $refines\ (whileLoop\ C\ B\ I)\ (whileLoop\ C'\ B'\ I')\ s\ s'\ R$
⟨*proof*⟩

lemma *refines-whileLoop''*:
assumes C: $\bigwedge a\ s\ b\ t.\ R\ (Result\ a,\ s)\ (Result\ b,\ t) \Longrightarrow C\ a\ s \longleftrightarrow C'\ b\ t$
and B: $\bigwedge a\ s\ b\ t.\ R\ (Result\ a,\ s)\ (Result\ b,\ t) \Longrightarrow C\ a\ s \Longrightarrow C'\ b\ t \Longrightarrow refines\ (B\ a)\ (B'\ b)\ s\ t\ R$
and I: $R\ (Result\ I,\ s)\ (Result\ I',\ s')$
and R: $\bigwedge r\ s\ r'\ s'.\ R\ (r,\ s)\ (r',\ s') \Longrightarrow rel\text{-}exception\text{-}or\text{-}result\ (\lambda\text{- - .}\ True)\ (\lambda\text{- - .}\ True)\ r\ r'$
shows $refines\ (whileLoop\ C\ B\ I)\ (whileLoop\ C'\ B'\ I')\ s\ s'\ R$
⟨*proof*⟩

lemma *rel-spec-monad-whileLoop-exn*:
assumes *init*: $R\ I\ I'$
assumes *cond*: $\bigwedge x\ y.\ R\ x\ y \Longrightarrow (rel\text{-}fun\ S\ (=))\ (C\ x)\ (C'\ y)$
assumes *body*: $\bigwedge x\ y.\ R\ x\ y \Longrightarrow rel\text{-}spec\text{-}monad\ S\ (rel\text{-}xval\ E\ R)\ (B\ x)\ (B'\ y)$
shows $rel\text{-}spec\text{-}monad\ S\ (rel\text{-}xval\ E\ R)\ (whileLoop\ C\ B\ I)\ (whileLoop\ C'\ B'\ I')$
⟨*proof*⟩

lemma *refines-whileLoop-guard-right*:
assumes $\bigwedge x\ s\ x'\ s'.\ R\ (Result\ x,\ s)\ (Result\ x',\ s') \Longrightarrow G'\ x'\ s' \Longrightarrow C\ x\ s = C'\ x'\ s'$
assumes $\bigwedge x\ s\ x'\ s'.\ R\ (Result\ x,\ s)\ (Result\ x',\ s') \Longrightarrow C\ x\ s \Longrightarrow C'\ x'\ s' \Longrightarrow G'\ x'\ s' \Longrightarrow refines\ (B\ x)\ (B'\ x')\ s\ s'\ R$
assumes $\bigwedge v\ s\ v'\ s'.\ R\ (v,\ s)\ (v',\ s') \Longrightarrow (\exists x.\ v = Result\ x) = (\exists x'.\ v' = Result\ x')$
assumes $R\ (Result\ x,\ s)\ (Result\ x',\ s')$
assumes $G\ s' = G'\ x'\ s'$
shows $refines\ (whileLoop\ C\ B\ x)\ (guard\ G \gg\!= (\lambda\text{-.}\ whileLoop\ C'\ (\lambda r.\ do\ \{res <\!- B'\ r;\ guard\ (G'\ res);\ return\ res\})\ x'))\ s\ s'\ R$
⟨*proof*⟩

16.9.31 *map-value*

lemma *map-value-lift-state*: $map\text{-}value\ f\ (lift\text{-}state\ R\ g) = lift\text{-}state\ R\ (map\text{-}value\ f\ g)$
⟨*proof*⟩

lemma *run-map-value*[*run-spec-monad*]: *run* (*map-value f g*) *s* =
map-post-state (λ(*v*, *s*). (*f v*, *s*)) (*run g s*)
⟨*proof*⟩

lemma *always-progress-map-value*[*always-progress-intros*]:
always-progress g ⟹ *always-progress* (*map-value f g*)
⟨*proof*⟩

lemma *runs-to-map-value-iff*[*runs-to-iff*]: *map-value f g* • *s* {‖ *Q* ‖} ⟷ *g* • *s* {⦃λ*r t*. *Q* (*f r*) *t*⦄}
⟨*proof*⟩

lemma *runs-to-partial-map-value-iff*[*runs-to-iff*]:
map-value f g • *s* ?{‖ *Q* ‖} ⟷ *g* • *s* ?{⦃λ*r t*. *Q* (*f r*) *t*⦄}
⟨*proof*⟩

lemma *runs-to-map-value*[*runs-to-vcg*]: *g* • *s* {⦃λ*r t*. *Q* (*f r*) *t*⦄} ⟹ *map-value f g* • *s* {‖ *Q* ‖}
⟨*proof*⟩

lemma *runs-to-partial-map-value*[*runs-to-vcg*]:
g • *s* ?{⦃λ*r t*. *Q* (*f r*) *t*⦄} ⟹ *map-value f g* • *s* ?{‖ *Q* ‖}
⟨*proof*⟩

lemma *mono-map-value*: *g* ≤ *g'* ⟹ *map-value f g* ≤ *map-value f g'*
⟨*proof*⟩

lemma *monotone-map-value-le*[*partial-function-mono*]:
monotone R (≤) (λ*f'*. *g f'*) ⟹ *monotone R* (≤) (λ*f'*. *map-value f* (*g f'*))
⟨*proof*⟩

lemma *monotone-map-value-ge*[*partial-function-mono*]:
monotone R (≥) (λ*f'*. *g f'*) ⟹ *monotone R* (≥) (λ*f'*. *map-value f* (*g f'*))
⟨*proof*⟩

lemma *map-value-fail*[*simp*]: *map-value f fail* = *fail*
⟨*proof*⟩

lemma *map-value-map-exn-gets*[*simp*]: *map-value* (*map-exn emb*) (*gets x*) = *gets x*
⟨*proof*⟩

lemma *refines-map-value-right-iff*:
refines f (*map-value m g*) *s t R* ⟷ *refines f g s t* (λ(*x*, *s*) (*y*, *t*). *R* (*x*, *s*) (*m y*, *t*))
⟨*proof*⟩

lemma *refines-map-value-left-iff*:

refines (*map-value m f*) *g s t R* ⟷ *refines f g s t* (λ(*x*, *s*) (*y*, *t*). *R* (*m x*, *s*) (*y*, *t*))
⟨*proof*⟩

lemma *rel-spec-map-value-right-iff*:
rel-spec f (*map-value m g*) *s t R* ⟷ *rel-spec f g s t* (λ(*x*, *s*) (*y*, *t*). *R* (*x*, *s*) (*m y*, *t*))
⟨*proof*⟩

lemma *rel-spec-map-value-left-iff*:
rel-spec (*map-value m f*) *g s t R* ⟷ *rel-spec f g s t* (λ(*x*, *s*) (*y*, *t*). *R* (*m x*, *s*) (*y*, *t*))
⟨*proof*⟩

lemma *refines-map-value-right-same*:
refines f (*map-value m f*) *s s* (λ(*x*, *s*) (*y*, *t*). *y* = *m x* ∧ *s* = *t*)
⟨*proof*⟩

lemma *refines-map-value*:
assumes *refines f f' s t Q*
assumes ⋀*r s' w t'*. *Q* (*r*, *s'*) (*w*, *t'*) ⟹ *R* (*g r*, *s'*) (*g' w*, *t'*)
shows *refines* (*map-value g f*) (*map-value g' f'*) *s t R*
⟨*proof*⟩

lemma *map-value-id*[*simp*]: *map-value* (λ*x*. *x*) = (λ*x*. *x*)
⟨*proof*⟩

16.9.32 *liftE*

lemma *run-liftE*[*run-spec-monad*]:
run (*liftE f*) *s* =
map-post-state (λ(*v*, *s*). (*map-exception-or-result* (λ*x*. *undefined*) *id v*, *s*)) (*run f s*)
⟨*proof*⟩

lemma *always-progress-liftE*[*always-progress-intros*]:
always-progress f ⟹ *always-progress* (*liftE f*)
⟨*proof*⟩

lemma *runs-to-liftE-iff*:
liftE f ∙ *s* ⦃ *Q* ⦄ ⟷ *f* ∙ *s* ⦃λ*r t*. *Q* (*map-exception-or-result* (λ*x*. *undefined*) *id r*) *t*⦄
⟨*proof*⟩

lemma *runs-to-liftE-iff-Res*[*runs-to-iff*]:
liftE f ∙ *s* ⦃ *Q* ⦄ ⟷ *f* ∙ *s* ⦃ λ*Res r*. *Q* (*Result r*)⦄
⟨*proof*⟩

lemma *runs-to-liftE'*:

$f \cdot s \ \{\!|\lambda r\ t.\ Q\ (\mathit{map\text{-}exception\text{-}or\text{-}result}\ (\lambda x.\ \mathit{undefined})\ \mathit{id}\ r)\ t|\!\} \Longrightarrow \mathit{liftE}\ f \cdot s \ \{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *runs-to-liftE*[*runs-to-vcg*]: $f \cdot s \ \{\!|\ \lambda \mathit{Res}\ r.\ Q\ (\mathit{Result}\ r)|\!\} \Longrightarrow \mathit{liftE}\ f \cdot s \ \{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *refines-liftE-left-iff*:
$\mathit{refines}\ (\mathit{liftE}\ f)\ g\ s\ t\ R \longleftrightarrow$
$\mathit{refines}\ f\ g\ s\ t\ (\lambda(x,\ s')\ (y,\ t').\ \forall v.\ x = \mathit{Result}\ v \longrightarrow R\ (\mathit{Result}\ v,\ s')\ (y,\ t'))$
⟨*proof*⟩

lemma *refines-liftE-right-iff*:
$\mathit{refines}\ f\ (\mathit{liftE}\ g)\ s\ t\ R \longleftrightarrow$
$\mathit{refines}\ f\ g\ s\ t\ (\lambda(x,\ s')\ (y,\ t').\ \forall v.\ y = \mathit{Result}\ v \longrightarrow R\ (x,\ s')\ (\mathit{Result}\ v,\ t'))$
⟨*proof*⟩

lemma *rel-spec-liftE*:
$\mathit{rel\text{-}spec}\ (\mathit{liftE}\ f)\ g\ s\ t\ R \longleftrightarrow$
$\mathit{rel\text{-}spec}\ f\ g\ s\ t\ (\lambda(x,\ s')\ (y,\ t').\ \forall v.\ x = \mathit{Result}\ v \longrightarrow R\ (\mathit{Result}\ v,\ s')\ (y,\ t'))$
⟨*proof*⟩

lemma *rel-spec-monad-bind-liftE*:
$\mathit{rel\text{-}spec\text{-}monad}\ R\ (\lambda \mathit{Res}\ v1\ \mathit{Res}\ v2.\ P\ v1\ v2)\ m\ n \Longrightarrow \mathit{rel\text{-}fun}\ P\ (\mathit{rel\text{-}spec\text{-}monad}\ R\ Q)\ f\ g \Longrightarrow$
$\mathit{rel\text{-}spec\text{-}monad}\ R\ Q\ ((\mathit{liftE}\ m) >>= f)\ ((\mathit{liftE}\ n) >>= g)$
⟨*proof*⟩

lemma *rel-spec-monad-bind-liftE′*:
$(\bigwedge x.\ \mathit{rel\text{-}spec\text{-}monad}\ (=)\ Q\ (f\ x)\ (g\ x)) \Longrightarrow \mathit{rel\text{-}spec\text{-}monad}\ (=)\ Q\ (\mathit{liftE}\ m >>= f)\ (\mathit{liftE}\ m >>= g)$
⟨*proof*⟩

lemma *runs-to-partial-liftE′*:
$f \cdot s\ ?\{\!|\lambda r\ t.\ Q\ (\mathit{map\text{-}exception\text{-}or\text{-}result}\ (\lambda x.\ \mathit{undefined})\ \mathit{id}\ r)\ t|\!\} \Longrightarrow \mathit{liftE}\ f \cdot s\ ?\{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *runs-to-partial-liftE*[*runs-to-vcg*]:
assumes [*runs-to-vcg*]: $f \cdot s\ ?\{\!|\ \lambda \mathit{Res}\ r.\ Q\ (\mathit{Result}\ r)|\!\}$ **shows** $\mathit{liftE}\ f \cdot s\ ?\{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *mono-lifE*: $f \leq f' \Longrightarrow \mathit{liftE}\ f\ \leq \mathit{liftE}\ f'$
⟨*proof*⟩

lemma *monotone-liftE-le*[*partial-function-mono*]:
$\mathit{monotone}\ R\ (\leq)\ (\lambda f'.\ f\ f') \Longrightarrow \mathit{monotone}\ R\ (\leq)\ (\lambda f'.\ \mathit{liftE}\ (f\ f'))$
⟨*proof*⟩

lemma *monotone-liftE-ge*[*partial-function-mono*]:
monotone R (≥) (λ*f'*. *f f'*) ⟹ *monotone R* (≥) (λ*f'*. *liftE* (*f f'*))
⟨*proof*⟩

lemma *bind-handle-liftE*: *bind-handle* (*liftE f*) *g h* = *bind f g*
⟨*proof*⟩

lemma *liftE-top*[*simp*]: *liftE* ⊤ = ⊤
⟨*proof*⟩

lemma *liftE-bot*[*simp*]: *liftE bot* = *bot*
⟨*proof*⟩

lemma *liftE-fail*[*simp*]: *liftE fail* = *fail*
⟨*proof*⟩

lemma *liftE-return*[*simp*]: *liftE* (*return x*) = *return x*
⟨*proof*⟩

lemma *liftE-yield-Exception*[*simp*]:
liftE (*yield* (*Exception x*)) = *skip*
⟨*proof*⟩

lemma *liftE-throw-exception-or-result*[*simp*]:
liftE (*throw-exception-or-result x*) = *return undefined*
⟨*proof*⟩

lemma *liftE-get-state*[*simp*]: *liftE* (*get-state*) = *get-state*
⟨*proof*⟩

lemma *liftE-set-state*[*simp*]: *liftE* (*set-state s*) = *set-state s*
⟨*proof*⟩

lemma *liftE-select*[*simp*]: *liftE* (*select S*) = *select S*
⟨*proof*⟩

lemma *liftE-unknown*[*simp*]: *liftE* (*unknown*) = *unknown*
⟨*proof*⟩

lemma *liftE-lift-state*: *liftE* (*lift-state R f*) = *lift-state R* (*liftE f*)
⟨*proof*⟩

lemma *liftE-exec-concrete*: *liftE* (*exec-concrete st f*) = *exec-concrete st* (*liftE f*)
⟨*proof*⟩

lemma *liftE-exec-abstract*: *liftE* (*exec-abstract st f*) = *exec-abstract st* (*liftE f*)
⟨*proof*⟩

lemma *liftE-assert*[*simp*]: *liftE* (*assert P*) = *assert P*
⟨*proof*⟩

lemma *liftE-assume*[*simp*]: *liftE* (*assume P*) = *assume P*
⟨*proof*⟩

lemma *liftE-gets*[*simp*]: *liftE* (*gets f*) = *gets f*
⟨*proof*⟩

lemma *liftE-guard*[*simp*]: *liftE* (*guard P*) = *guard P*
⟨*proof*⟩

lemma *liftE-assert-opt*[*simp*]: *liftE* (*assert-opt v*) = *assert-opt v*
⟨*proof*⟩

lemma *liftE-gets-the*[*simp*]: *liftE* (*gets-the f*) = *gets-the f*
⟨*proof*⟩

lemma *liftE-modify*[*simp*]: *liftE* (*modify f*) = *modify f*
⟨*proof*⟩

lemma *liftE-bind*: *liftE x* >>= (λ*a*. *liftE* (*y a*)) = *liftE* (*x* >>= *y*)
⟨*proof*⟩

lemma *bindE-liftE-skip*: *liftE* (*f* ≫= (λ*y*. *skip*)) ≫= *g* = *liftE f* ≫= (λ-. *g* ())
⟨*proof*⟩

lemma *liftE-state-select*[*simp*]: *liftE* (*state-select f*) = *state-select f*
⟨*proof*⟩

lemma *liftE-assume-result-and-state*[*simp*]:
liftE (*assume-result-and-state f*) = *assume-result-and-state f*
⟨*proof*⟩

lemma *map-value-map-exn-liftE* [*simp*]:
map-value (*map-exn emb*) (*liftE f*) = *liftE f*
⟨*proof*⟩

lemma *liftE-condition*: *liftE* (*condition c f g*) = *condition c* (*liftE f*) (*liftE g*)
⟨*proof*⟩

lemma *liftE-whileLoop*: *liftE* (*whileLoop C B I*) = *whileLoop C* (λ*r*. *liftE* (*B r*)) *I*
⟨*proof*⟩

16.9.33 *try*

lemma *run-try*[*run-spec-monad*]:
run (*try f*) *s* = *map-post-state* (λ(*v*, *s*). (*unnest-exn v*, *s*)) (*run f s*)
⟨*proof*⟩

lemma *always-progress-try*[*always-progress-intros*]: *always-progress f* $\Longrightarrow$ *always-progress* (*try f*)
⟨*proof*⟩

lemma *runs-to-try*[*runs-to-vcg*]: *f* · *s* ⦃$\lambda r\ t.\ Q$ (*unnest-exn r*) *t*⦄ $\Longrightarrow$ *try f* · *s* ⦃ *Q* ⦄
⟨*proof*⟩

lemma *runs-to-try-iff*[*runs-to-iff*]: *try f* · *s* ⦃ *Q* ⦄ $\longleftrightarrow$ *f* · *s* ⦃$\lambda r\ t.\ Q$ (*unnest-exn r*) *t*⦄
⟨*proof*⟩

lemma *runs-to-partial-try*[*runs-to-vcg*]: *f* · *s* ?⦃$\lambda r\ t.\ Q$ (*unnest-exn r*) *t*⦄ $\Longrightarrow$ *try f* · *s* ?⦃ *Q* ⦄
⟨*proof*⟩

lemma *mono-try*: *f* $\leq$ *f'* $\Longrightarrow$ *try f* $\leq$ *try f'*
⟨*proof*⟩

lemma *monotone-try-le*[*partial-function-mono*]:
monotone R ($\leq$) ($\lambda f'.\ f\ f'$) $\Longrightarrow$ *monotone R* ($\leq$) ($\lambda f'.$ *try* (*f f'*))
⟨*proof*⟩

lemma *monotone-try-ge*[*partial-function-mono*]:
monotone R ($\geq$) ($\lambda f'.\ f\ f'$) $\Longrightarrow$ *monotone R* ($\geq$) ($\lambda f'.$ *try* (*f f'*))
⟨*proof*⟩

lemma *refines-try-right-same*:
refines f (*try f*) *s s* ($\lambda(x,\ s)\ (y,\ t).\ y$ = *unnest-exn x* $\wedge$ *s* = *t*)
⟨*proof*⟩

16.9.34 *finally*

lemma *run-finally*[*run-spec-monad*]:
run (*finally f*) *s* = *map-post-state* ($\lambda(v,\ s).$ (*unite v*, *s*)) (*run f s*)
⟨*proof*⟩

lemma *always-progress-finally*[*always-progress-intros*]:
always-progress f $\Longrightarrow$ *always-progress* (*finally f*)
⟨*proof*⟩

lemma *runs-to-finally-iff*[*runs-to-iff*]:
finally f · *s* ⦃ *Q* ⦄ $\longleftrightarrow$
f · *s* ⦃$\lambda r\ t.$ ($\forall v.\ r$ = *Result v* $\longrightarrow$ *Q* (*Result v*) *t*) $\wedge$ ($\forall v.\ r$ = *Exn v* $\longrightarrow$ *Q* (*Result v*) *t*)⦄
⟨*proof*⟩

lemma *runs-to-finally'*: *f* · *s* ⦃$\lambda r\ t.\ Q$ (*unite r*) *t*⦄ $\Longrightarrow$ *finally f* · *s* ⦃ *Q* ⦄

⟨*proof*⟩

lemma *runs-to-finally*[*runs-to-vcg*]:
$f \cdot s \ \{\!|\lambda r\ t.\ (\forall v.\ r = Result\ v \longrightarrow Q\ (Result\ v)\ t) \wedge (\forall v.\ r = Exn\ v \longrightarrow Q\ (Result\ v)\ t)|\!\} \Longrightarrow$
finally $f \cdot s \ \{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *runs-to-partial-finally'*: $f \cdot s \ ?\{\!|\lambda r\ t.\ Q\ (unite\ r)\ t|\!\} \Longrightarrow$ *finally* $f \cdot s \ ?\{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *runs-to-partial-finally-iff*:
finally $f \cdot s \ ?\{\!|\ Q\ |\!\} \longleftrightarrow$
$f \cdot s \ ?\{\!|\lambda r\ t.\ (\forall v.\ r = Result\ v \longrightarrow Q\ (Result\ v)\ t) \wedge (\forall v.\ r = Exn\ v \longrightarrow Q\ (Result\ v)\ t)|\!\}$
⟨*proof*⟩

lemma *runs-to-partial-finally*[*runs-to-vcg*]:
$f \cdot s \ ?\{\!|\lambda r\ t.\ (\forall v.\ r = Result\ v \longrightarrow Q\ (Result\ v)\ t) \wedge (\forall v.\ r = Exn\ v \longrightarrow Q\ (Result\ v)\ t)|\!\} \Longrightarrow$
finally $f \cdot s \ ?\{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *mono-finally*: $f \leq f' \Longrightarrow$ *finally* $f \leq$ *finally* f'
⟨*proof*⟩

lemma *monotone-finally-le*[*partial-function-mono*]:
monotone $R\ (\leq)\ (\lambda f'.\ f\ f') \Longrightarrow$ *monotone* $R\ (\leq)\ (\lambda f'.$ *finally* $(f\ f'))$
⟨*proof*⟩

lemma *monotone-finally-ge*[*partial-function-mono*]:
monotone $R\ (\geq)\ (\lambda f'.\ f\ f') \Longrightarrow$ *monotone* $R\ (\geq)\ (\lambda f'.$ *finally* $(f\ f'))$
⟨*proof*⟩

16.9.35 (<*catch*>)

lemma *run-catch*[*run-spec-monad*]:
run $(f$ <*catch*> $h)\ s =$
bind-post-state $(run\ f\ s)$
$(\lambda(r,\ t).$ *case* r *of* $Exn\ e \Rightarrow run\ (h\ e)\ t \mid Result\ v \Rightarrow run\ (return\ v)\ t)$
⟨*proof*⟩

lemma *always-progress-catch*[*always-progress-intros*]:
always-progress $f \Longrightarrow (\bigwedge e.$ *always-progress* $(h\ e)) \Longrightarrow$ *always-progress* $(f$ <*catch*> $h)$
⟨*proof*⟩

lemma *runs-to-catch-iff*[*runs-to-iff*]:

$(f <catch> h) \cdot s \{\!|\ Q\ |\!\} \longleftrightarrow$
$f \cdot s \{\!|\ \lambda r\ t.\ (\forall v.\ r = Result\ v \longrightarrow Q\ (Result\ v)\ t) \wedge (\forall e.\ r = Exn\ e \longrightarrow h\ e \cdot t \{\!|\ Q\ |\!\})|\!\}$
⟨*proof*⟩

lemma *runs-to-catch*[*runs-to-vcg*]:
$f \cdot s \{\!|\ \lambda r\ t.\ (\forall v.\ r = Result\ v \longrightarrow Q\ (Result\ v)\ t) \wedge$
$(\forall e.\ r = Exn\ e \longrightarrow h\ e \cdot t \{\!|\ Q\ |\!\})|\!\} \Longrightarrow$
$(f <catch> h) \cdot s \{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *runs-to-partial-catch*[*runs-to-vcg*]:
$f \cdot s\ ?\{\!|\ \lambda r\ t.\ (\forall v.\ r = Result\ v \longrightarrow Q\ (Result\ v)\ t) \wedge$
$(\forall e.\ r = Exn\ e \longrightarrow h\ e \cdot t\ ?\{\!|\ Q\ |\!\})|\!\} \Longrightarrow$
$(f <catch> h) \cdot s\ ?\{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *mono-catch*: $f \leq f' \Longrightarrow h \leq h' \Longrightarrow catch\ f\ h \leq catch\ f'\ h'$
⟨*proof*⟩

lemma *monotone-catch-le*[*partial-function-mono*]:
$monotone\ R\ (\leq)\ (\lambda f'.\ f\ f') \Longrightarrow (\bigwedge e.\ monotone\ R\ (\leq)\ (\lambda f'.\ h\ f'\ e))$
$\Longrightarrow monotone\ R\ (\leq)\ (\lambda f'.\ catch\ (f\ f')\ (h\ f'))$
⟨*proof*⟩

lemma *monotone-catch-ge*[*partial-function-mono*]:
$monotone\ R\ (\geq)\ (\lambda f'.\ f\ f') \Longrightarrow (\bigwedge e.\ monotone\ R\ (\geq)\ (\lambda f'.\ h\ f'\ e))$
$\Longrightarrow monotone\ R\ (\geq)\ (\lambda f'.\ catch\ (f\ f')\ (h\ f'))$
⟨*proof*⟩

lemma *catch-liftE*: $catch\ (liftE\ g)\ h = g$
⟨*proof*⟩

lemma *refines-catch*:
assumes *f*: $refines\ f\ f'\ s\ s'\ Q$
assumes *ll*: $\bigwedge e\ e'\ t\ t'.\ Q\ (Exn\ e,\ t)\ (Exn\ e',\ t') \Longrightarrow$
$refines\ (h\ e)\ (h'\ e')\ t\ t'\ R$
assumes *lr*: $\bigwedge e\ v'\ t\ t'.\ Q\ (Exn\ e,\ t)\ (Result\ v',\ t') \Longrightarrow$
$refines\ (h\ e)\ (return\ v')\ t\ t'\ R$
assumes *rl*: $\bigwedge v\ e'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Exn\ e',\ t') \Longrightarrow$
$refines\ (return\ v)\ (h'\ e')\ t\ t'\ R$
assumes *rr*: $\bigwedge v\ v'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Result\ v',\ t') \Longrightarrow$
$R\ (Result\ v,\ t)\ (Result\ v',\ t')$
shows $refines\ (catch\ f\ h)\ (catch\ f'\ h')\ s\ s'\ R$
⟨*proof*⟩

lemma *rel-spec-catch*:
assumes *f*: $rel\text{-}spec\ f\ f'\ s\ s'\ Q$
assumes *ll*: $\bigwedge e\ e'\ t\ t'.\ Q\ (Exn\ e,\ t)\ (Exn\ e',\ t') \Longrightarrow$

rel-spec (*h e*) (*h′ e′*) *t t′ R*
assumes *lr*: $\bigwedge e\ v'\ t\ t'.\ Q\ (Exn\ e,\ t)\ (Result\ v',\ t') \Longrightarrow$
rel-spec (*h e*) (*return v′*) *t t′ R*
assumes *rl*: $\bigwedge v\ e'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Exn\ e',\ t') \Longrightarrow$
rel-spec (*return v*) (*h′ e′*) *t t′ R*
assumes *rr*: $\bigwedge v\ v'\ t\ t'.\ Q\ (Result\ v,\ t)\ (Result\ v',\ t') \Longrightarrow$
R (*Result v*, *t*) (*Result v′*, *t′*)
shows *rel-spec* (*catch f h*) (*catch f′ h′*) *s s′ R*
⟨*proof*⟩

16.9.36 *check*

lemma *run-check*[*run-spec-monad*]: *run* (*check e p*) *s* =
(*if* $(\exists x.\ p\ s\ x)$ *then Success* ((λ*x*. (*x*, *s*)) ‘ (*Result* ‘ {*x*. *p s x*})) *else*
pure-post-state (*Exn e*, *s*))
⟨*proof*⟩

lemma *always-progress-check*[*always-progress-intros*, *simp*]: *always-progress* (*check e p*)
⟨*proof*⟩

lemma *runs-to-check-iff*[*runs-to-iff*]:
(*check e p*) · *s* {‖*Q*‖} ⟷ (*if* $(\exists x.\ p\ s\ x)$ *then* $(\forall x.\ p\ s\ x \longrightarrow Q\ (Result\ x)\ s)$ *else* *Q* (*Exn e*) *s*)
⟨*proof*⟩

lemma *runs-to-check*[*runs-to-vcg*]:
$(\exists x.\ p\ s\ x \Longrightarrow (\forall x.\ p\ s\ x \longrightarrow Q\ (Result\ x)\ s)) \Longrightarrow (\forall x.\ \neg\ p\ s\ x) \Longrightarrow Q\ (Exn\ e)\ s \Longrightarrow$
check e p · *s* {‖*Q*‖}
⟨*proof*⟩

lemma *runs-to-partial-check*[*runs-to-vcg*]:
$(\exists x.\ p\ s\ x \Longrightarrow (\forall x.\ p\ s\ x \longrightarrow Q\ (Result\ x)\ s)) \Longrightarrow (\forall x.\ \neg\ p\ s\ x) \Longrightarrow Q\ (Exn\ e)\ s \Longrightarrow$
check e p · *s* ?{‖*Q*‖}
⟨*proof*⟩

lemma *refines-check-right-ok*:
Q t a ⟹ *refines f* (*g a*) *s t R* ⟹ *refines f* (*check q Q* >>= *g*) *s t R*
⟨*proof*⟩

lemma *refines-check-right-fail*:
$(\forall a.\ \neg\ Q\ t\ a) \Longrightarrow f$ · *s* {‖ λ*r s′*. *R* (*r*, *s′*) (*Exn q*, *t*) ‖} ⟹ *refines f* (*check q Q* >>= *g*) *s t R*
⟨*proof*⟩

lemma *refines-throwError-check*:
$(\forall a.\ \neg\ P\ t\ a) \Longrightarrow R\ (Exn\ r,\ s)\ (Exn\ q,\ t) \Longrightarrow$

refines (*throw r*) (*check q P* >>= *f*) *s t R*
⟨*proof*⟩

lemma *refines-condition-neg-check*:
P s ⟷ (∀ *a*. ¬ *Q t a*) ⟹
(*P s* ⟹ ∀ *a*. ¬ *Q t a* ⟹ *R* (*Result r*, *s*) (*Exn q*, *t*)) ⟹
(⋀*a*. ¬ *P s* ⟹ *Q t a* ⟹ *refines f* (*g a*) *s t R*) ⟹
refines (*condition P* (*return r*) *f*) (*check q Q* >>= *g*) *s t R*
⟨*proof*⟩

16.9.37 *ignoreE*

named-theorems *ignoreE-simps* ‹*Rewrite rules to push* ***const***‹*ignoreE*› *inside.*›

lemma *ignoreE-eq*: *ignoreE f* = *vmap-value* (*map-exception-or-result* (λ*x*. *undefined*) *id*) *f*
⟨*proof*⟩

lemma *run-ignoreE*: *run* (*ignoreE f*) *s* =
bind-post-state (*run f s*)
(λ(*r*, *t*). *case r of Exn e* ⇒ ⊥ | *Result v* ⇒ *pure-post-state* (*Result v*, *t*))
⟨*proof*⟩

lemma *runs-to-ignoreE-iff*[*runs-to-iff*]:
(*ignoreE f*) ⋅ *s* ⦃ *Q* ⦄ ⟷ *f* ⋅ *s* ⦃ λ*r t*. (∀ *v*. *r* = *Result v* ⟶ *Q* (*Result v*) *t*)⦄
⟨*proof*⟩

lemma *runs-to-ignoreE*[*runs-to-vcg*]:
f ⋅ *s* ⦃ λ*r t*. (∀ *v*. *r* = *Result v* ⟶ *Q* (*Result v*) *t*)⦄ ⟹ (*ignoreE f*) ⋅ *s* ⦃ *Q* ⦄
⟨*proof*⟩

lemma *liftE-le-iff-le-ignoreE*: *liftE f* ≤ *g* ⟷ *f* ≤ *ignoreE g*
⟨*proof*⟩

lemma *runs-to-partial-ignoreE-iff*:
ignoreE f ⋅ *s* ?⦃ *P* ⦄ ⟷ *f* ⋅ *s* ?⦃ λ*r t*. ∀ *v*. *r* = *Result v* ⟶ *P* (*Result v*) *t* ⦄
⟨*proof*⟩

lemma *runs-to-partial-ignoreE*[*runs-to-vcg*]:
f ⋅ *s* ?⦃ λ*r t*. (∀ *v*. *r* = *Result v* ⟶ *Q* (*Result v*) *t*)⦄ ⟹ (*ignoreE f*) ⋅ *s* ?⦃ *Q* ⦄
⟨*proof*⟩

lemma *mono-ignoreE*: *f* ≤ *f'* ⟹ *ignoreE f* ≤ *ignoreE f'*
⟨*proof*⟩

lemma *monotone-ignoreE-le*[*partial-function-mono*]:
monotone R (≤) (λ*f'*. *f f'*)
⟹ *monotone R* (≤) (λ*f'*. *ignoreE* (*f f'*))
⟨*proof*⟩

lemma *monotone-ignoreE-ge*[*partial-function-mono*]:
monotone R $(\geq)$ $(\lambda f'.\ f\ f')$
$\Longrightarrow$ *monotone* R $(\geq)$ $(\lambda f'.\ ignoreE\ (f\ f'))$
⟨*proof*⟩

lemma *ignoreE-liftE* [*simp*]: *ignoreE* (*liftE* f) = f
⟨*proof*⟩

lemma *ignoreE-top*[*simp*]: *ignoreE* $\top$ = $\top$
⟨*proof*⟩

lemma *ignoreE-bot*[*simp*]: *ignoreE bot* = *bot*
⟨*proof*⟩

lemma *ignoreE-fail*[*simp*]: *ignoreE fail* = *fail*
⟨*proof*⟩

lemma *ignoreE-return*[*simp*]: *ignoreE* (*return* x) = *return* x
⟨*proof*⟩

lemma *ignoreE-throw*[*simp*]: *ignoreE* (*throw* x) = *bot*
⟨*proof*⟩

lemma *ignoreE-yield-Exception*[*simp*]: $e \neq default \Longrightarrow$ *ignoreE* (*yield* (*Exception* e)) = *bot*
⟨*proof*⟩

lemma *ignoreE-guard*[*simp*]: *ignoreE* (*guard* P) = *guard* P
⟨*proof*⟩

lemma *ignoreE-get-state*[*simp*]: *ignoreE* (*get-state*) = *get-state*
⟨*proof*⟩

lemma *ignoreE-set-state*[*simp*]: *ignoreE* (*set-state* s) = *set-state* s
⟨*proof*⟩

lemma *ignoreE-select*[*simp*]: *ignoreE* (*select* S) = *select* S
⟨*proof*⟩

lemma *ignoreE-unknown*[*simp*]: *ignoreE* (*unknown*) = *unknown*
⟨*proof*⟩

lemma *ignoreE-exec-concrete*[*simp*]: *ignoreE* (*exec-concrete st* f) = *exec-concrete st* (*ignoreE* f)
⟨*proof*⟩

lemma *ignoreE-assert*[*simp*]: *ignoreE* (*assert* P) = *assert* P
⟨*proof*⟩

lemma *ignoreE-assume*[*simp*]: *ignoreE* (*assume P*) = *assume P*
⟨*proof*⟩

lemma *ignoreE-gets*[*simp*]: *ignoreE* (*gets f*) = *gets f*
⟨*proof*⟩

lemma *ignoreE-assert-opt*[*simp*]: *ignoreE* (*assert-opt v*) = *assert-opt v*
⟨*proof*⟩

lemma *ignoreE-gets-the*[*simp*]: *ignoreE* (*gets-the f*) = *gets-the f*
⟨*proof*⟩

lemma *ignoreE-modify*[*simp*]: *ignoreE* (*modify f*) = *modify f*
⟨*proof*⟩

lemma *ignoreE-condition*[*ignoreE-simps*]:
ignoreE (*condition c f g*) = *condition c* (*ignoreE f*) (*ignoreE g*)
⟨*proof*⟩

lemma *ignoreE-when*[*simp*]: *ignoreE* (*when P f*) = *when P* (*ignoreE f*)
⟨*proof*⟩

lemma *ignoreE-bind*[*ignoreE-simps*]:
ignoreE (*bind f g*) = *bind* (*ignoreE f*) (λ*v*. (*ignoreE* (*g v*)))
⟨*proof*⟩

lemma *ignoreE-map-exn*[*simp*]: *ignoreE* (*map-value* (*map-exn f*) *g*) = *ignoreE g*
⟨*proof*⟩

lemma *ignoreE-whileLoop*[*ignoreE-simps*]:
ignoreE (*whileLoop C B I*) = *whileLoop C* (λ*x*. *ignoreE* (*B x*)) *I*
⟨*proof*⟩

16.9.38 *on-exit′*

lemmas *bind-finally-def* = *bind-exception-or-result-def*

lemma *bind-exception-or-result-liftE-assoc*:
bind-exception-or-result (*bind* (*liftE f*) *g*) *h* = *bind f* (λ*v*. *bind-exception-or-result* (*g v*) *h*)
⟨*proof*⟩

lemma *bind-exception-or-result-bind-guard-assoc*:
bind-exception-or-result (*bind* (*guard P*) *g*) *h* =
bind (*guard P*) (λ*v*. *bind-exception-or-result* (*g v*) *h*)
⟨*proof*⟩

lemma *on-exit-bind-exception-or-result-conv*:

on-exit f cleanup = *bind-exception-or-result f* (λ*x*. *do* {*state-select cleanup*; *yield x*})
⟨*proof*⟩

lemma *guard-on-exit-bind-exception-or-result-conv*:
guard-on-exit f P cleanup =
bind-exception-or-result f (λ*x*. *do* {*guard P*; *state-select cleanup*; *yield x*})
⟨*proof*⟩

lemma *assume-result-and-state-check-only-state*:
assume-result-and-state (λ*s*. {((), *s'*). *s'* = *s* ∧ *P s*}) = *assuming P*
⟨*proof*⟩

lemma *assume-on-exit-bind-exception-or-result-conv*:
assume-on-exit f P cleanup = *bind-exception-or-result f*
(λ*x*. *do* {*assume-result-and-state* (λ*s*. {((), *s'*). *s'* = *s* ∧ *P s*}); *state-select cleanup*; *yield x*})
⟨*proof*⟩

lemma *monotone-on-exit'-le*[*partial-function-mono*]:
monotone R (≤) (λ*f'*. *f f'*) ⟹ *monotone R* (≤) (λ*f'*. *g f'*) ⟹
monotone R (≤) (λ*f'*. *on-exit'* (*f f'*) (*g f'*))
⟨*proof*⟩

lemma *monotone-on-exit'-ge*[*partial-function-mono*]:
monotone R (≥) (λ*f'*. *f f'*) ⟹ *monotone R* (≥) (λ*f'*. *g f'*) ⟹
monotone R (≥) (λ*f'*. *on-exit'* (*f f'*) (*g f'*))
⟨*proof*⟩

lemma *runs-to-on-exit'-iff*[*runs-to-iff*]:
on-exit' f c · *s* ⦃ *Q* ⦄ ⟷
f · *s* ⦃ λ*r t*. *c* · *t* ⦃ λ*q t*. *Q* (*case q of Exception e* ⇒ *Exception e* | *Result* - ⇒ *r*) *t* ⦄ ⦄
⟨*proof*⟩

lemma *runs-to-on-exit'*[*runs-to-vcg*]:
f · *s* ⦃ λ*r t*. *c* · *t* ⦃ λ*q t*. *Q* (*case q of Exception e* ⇒ *Exception e* | *Result* - ⇒ *r*) *t* ⦄ ⦄ ⟹ *on-exit' f c* · *s* ⦃ *Q* ⦄
⟨*proof*⟩

lemma *runs-to-partial-on-exit'*[*runs-to-vcg*]:
f · *s* ?⦃ λ*r t*. *c* · *t* ?⦃ λ*q t*. *Q* (*case q of Exception e* ⇒ *Exception e* | *Result* - ⇒ *r*) *t* ⦄ ⦄ ⟹ *on-exit' f c* · *s* ?⦃ *Q* ⦄
⟨*proof*⟩

lemma *refines-on-exit'*:
assumes *f*: *refines f f' s s'*
(λ(*r*, *t*) (*r'*, *t'*). *refines c c' t t'* (λ(*q*, *t*) (*q'*, *t'*). *R*
(*case q of Exception e* ⇒ *Exception e* | *Result* - ⇒ *r*, *t*)

(*case* q' *of Exception* $e' \Rightarrow$ *Exception* $e' \mid$ *Result* - $\Rightarrow r', t'$)))
shows *refines* (*on-exit'* $f\ c$) (*on-exit'* $f'\ c'$) $s\ s'\ R$
⟨*proof*⟩

lemma *on-exit'-skip*: *on-exit'* f *skip* $= f$
⟨*proof*⟩

16.9.39 *run-bind*

lemma *run-run-bind*: *run* (*run-bind* $f\ t\ g$) $s =$ *bind-post-state* (*run* $f\ t$) ($\lambda(r,\ t)$. *run* ($g\ r\ t$) s)
⟨*proof*⟩

lemma *runs-to-run-bind-iff*[*runs-to-iff*]:
(*run-bind* $f\ t\ g$) $\cdot\ s\ \{\!|Q|\!\} \longleftrightarrow f \cdot t\ \{\!|\lambda r\ t.\ (g\ r\ t) \cdot s\ \{\!|Q|\!\}\ |\!\}$
⟨*proof*⟩

lemma *runs-to-run-bind*[*runs-to-vcg*]:
$f \cdot t\ \{\!|\lambda r\ t.\ (g\ r\ t) \cdot s\ \{\!|Q|\!\}\ |\!\} \Longrightarrow$ (*run-bind* $f\ t\ g$) $\cdot\ s\ \{\!|Q|\!\}$
⟨*proof*⟩

lemma *runs-to-partial-run-bind*[*runs-to-vcg*]:
$f \cdot t\ ?\{\!|\lambda r\ t.\ (g\ r\ t) \cdot s\ ?\{\!|Q|\!\}\ |\!\} \Longrightarrow$ (*run-bind* $f\ t\ g$) $\cdot\ s\ ?\{\!|Q|\!\}$
⟨*proof*⟩

lemma *mono-run-bind*: $f \leq f' \Longrightarrow g \leq g' \Longrightarrow$ *run-bind* $f\ t\ g \leq$ *run-bind* $f'\ t\ g'$
⟨*proof*⟩

lemma *monotone-run-bind-le*[*partial-function-mono*]:
monotone $R\ (\leq)\ (\lambda f'.\ f\ f') \Longrightarrow (\bigwedge r\ t.$ *monotone* $R\ (\leq)\ (\lambda f'.\ g\ f'\ r\ t))$
$\Longrightarrow$ *monotone* $R\ (\leq)\ (\lambda f'.$ *run-bind* $(f\ f')\ t\ (g\ f'))$
⟨*proof*⟩

lemma *monotone-run-bind-ge*[*partial-function-mono*]:
monotone $R\ (\geq)\ (\lambda f'.\ f\ f') \Longrightarrow (\bigwedge r\ t.$ *monotone* $R\ (\geq)\ (\lambda f'.\ g\ f'\ r\ t))$
$\Longrightarrow$ *monotone* $R\ (\geq)\ (\lambda f'.$ *run-bind* $(f\ f')\ t\ (g\ f'))$
⟨*proof*⟩

lemma *liftE-run-bind*: *liftE* (*run-bind* $f\ t\ g$) $=$ *run-bind* $f\ t\ (\lambda r\ t.$ *liftE* $(g\ r\ t))$
⟨*proof*⟩

lemma *exec-concrete-run-bind*: *exec-concrete* $st\ f =$
do {
$s \leftarrow$ *get-state*;
$t \leftarrow$ *select* $\{t.\ st\ t = s\}$;
run-bind $f\ t\ (\lambda r'\ t'.$ *do* {*set-state* ($st\ t'$); *yield* r' })
}
⟨*proof*⟩

lemma *exec-abstract-run-bind*: *exec-abstract st f* =
do {
s ← *get-state*;
run-bind f (*st s*) (λ*r′ t′*. *do* {*s′* ← *select* {*s′*. *t′* = *st s′*}; *set-state s′*; *yield r′* })
}
⟨*proof*⟩

lemma *refines-run-bind*:
refines f f′ x y (λ(*r*,*x′*) (*r′*, *y′*). *refines* (*g r x′*) (*g′ r′ y′*) *s t R*) ⟹
refines (*run-bind f x g*) (*run-bind f′ y g′*) *s t R*
⟨*proof*⟩

16.9.40 Iteration of monadic actions

fun *iter-spec-monad* **where**
iter-spec-monad f 0 = *skip* |
iter-spec-monad f (*Suc n*) = *do* {*iter-spec-monad f n*; *f n* }

lemma *iter-spec-monad-unfold*:
0 < *n* ⟹ *iter-spec-monad f n* = *do* {*iter-spec-monad f* (*n*−*1*); *f* (*n*−*1*) }
⟨*proof*⟩

lemma *iter-spec-monad-cong*:
(⋀*j*. *j* < *i* ⟹ *f j* = *g j*) ⟹ *iter-spec-monad f i* = *iter-spec-monad g i*
⟨*proof*⟩

16.9.41 *forLoop*

definition *forLoop*:: *int* ⇒ *int* ⇒ (*int* ⇒ ('*a*, '*s*) *res-monad*) ⇒ (*int*, '*s*) *res-monad*
where
forLoop z (*m*::*int*) *B* = *whileLoop* (λ*i s*. *i* < *m*) (λ*i*. *do* {*B i*; *return* (*i* + *1*) }) *z*

lemma *runs-to-forLoop*:
assumes *z* ≤ *m*
assumes *I*: ⋀*r s*. *I r s* ⟹ *z* ≤ *r* ⟹ *r* < *m* ⟹ (*B r*) · *s* {| λ- *t*. *I* (*r* + *1*) *t* |}
assumes *Q*: ⋀*s*. *I m s* ⟹ *Q m s*
shows *I z s* ⟹ (*forLoop z m B*) · *s* {|λ*Res r*. *Q r*|}
⟨*proof*⟩

lemma *whileLoop-cong-inv*:
run ((*whileLoop C f z*)::('*a*, '*s*) *res-monad*) *s* = *run* (*whileLoop D g z*) *s*
if *I*: ⋀*i s*. *I i s* ⟹ *C i s* ⟹ *f i* · *s* ?{| λ*v′ s′*. ∀ *i′*. *v′* = *Result i′* ⟶ *I i′ s′*|}
and *I-eq*: ⋀*i s*. *I i s* ⟹ *C i s* ⟹ *run* (*f i*) *s* = *run* (*g i*) *s*
and *C-eq*: ⋀*i s*. *I i s* ⟹ *C i s* ⟷ *D i s*
and *I z s*
for *s*::'*s* **and** *z*::'*a* **and** *R*::('*a* × '*s*) *rel*
⟨*proof*⟩

lemma *forLoop-skip*: *forLoop z m f* = *return z* **if** *z* ≥ *m*
⟨*proof*⟩

lemma *forLoop-eq-whileLoop*: *forLoop z m f = whileLoop C g z*
  if $\bigwedge i.\ z \leq i \Longrightarrow i < m \Longrightarrow g\ i = do\ \{\ y \leftarrow f\ i;\ return\ (i + 1)\ \}$
    $\bigwedge i\ s.\ z \leq i \Longrightarrow i \leq m \Longrightarrow C\ i\ s \longleftrightarrow i < m$
    $\bigwedge s.\ z > m \Longrightarrow \neg\ C\ z\ s$
⟨*proof*⟩

lemma *runs-to-partial-forLoopE*: *forLoop z m f · s ?⦃ λx s′. ∀ v. x = Result v ⟶ v = m ⦄*
  if $z \leq m$
⟨*proof*⟩

16.9.42 *whileLoop-unroll-reachable*

context fixes *C* :: *'a ⇒ 's ⇒ bool* **and** *B* :: *'a ⇒* (*'e*::*default*, *'a*, *'s*) *spec-monad*
begin

inductive *whileLoop-unroll-reachable* :: *'a ⇒ 's ⇒ 'a ⇒ 's ⇒ bool* **for** *a s* **where**
  initial[*intro*, *simp*]: *whileLoop-unroll-reachable a s a s*
| *step*: $\bigwedge b\ t\ X\ c\ u.$
    whileLoop-unroll-reachable a s b t ⟹ C b t ⟹
    run (*B b*) *t* = *Success X ⟹* (*Result c*, *u*) ∈ *X ⟹*
    whileLoop-unroll-reachable a s c u

lemma *whileLoop-unroll-reachable-trans*:
  assumes *a-b*: *whileLoop-unroll-reachable a s b t* **and** *b-c*: *whileLoop-unroll-reachable b t c u*
  shows *whileLoop-unroll-reachable a s c u*
⟨*proof*⟩

end

lemma *run-whileLoop-unroll-reachable-cong*:
  assumes *eq*:
    $\bigwedge b\ t.$ *whileLoop-unroll-reachable C B a s b t ⟹ C b t ⟷ C′ b t*
    $\bigwedge b\ t.$ *whileLoop-unroll-reachable C B a s b t ⟹ C b t ⟹ run* (*B b*) *t* = *run* (*B′ b*) *t*
  shows *run* (*whileLoop C B a*) *s* = *run* (*whileLoop C′ B′ a*) *s*
⟨*proof*⟩

16.9.43 *on-exit*

lemma *refines-rel-prod-on-exit*:
  assumes *f*: *refines* $f_c\ f_a\ s_c\ s_a$ (*rel-prod R S′*)
  assumes *cleanup*: $\bigwedge s_c\ s_a\ t_c.\ S'\ s_c\ s_a \Longrightarrow (s_c,\ t_c) \in cleanup_c \Longrightarrow \exists\, t_a.\ (s_a,\ t_a) \in cleanup_a \wedge S\ t_c\ t_a$
  assumes *emp*: $\bigwedge s_c\ s_a.\ S'\ s_c\ s_a \Longrightarrow \nexists\, t_c.\ (s_c,\ t_c) \in cleanup_c \Longrightarrow \nexists\, t_a.\ (s_a,\ t_a) \in cleanup_a$
  shows *refines* (*on-exit* f_c $cleanup_c$) (*on-exit* f_a $cleanup_a$) s_c s_a (*rel-prod R S*)
  ⟨*proof*⟩

lemma *refines-runs-to-partial-rel-prod-on-exit*:
assumes *f*: *refines* f_c f_a s_c s_a (*rel-prod R S'*)
assumes *runs-to*: $f_c \cdot s_c$ $?\{\!|\lambda r\ t.\ P\ t|\!\}$
assumes *cleanup*: $\bigwedge s_c\ s_a\ t_c.\ S'\ s_c\ s_a \Longrightarrow (s_c,\ t_c) \in cleanup_c \Longrightarrow P\ s_c \Longrightarrow \exists\, t_a.\ (s_a,\ t_a) \in cleanup_a \wedge S\ t_c\ t_a$
assumes *emp*: $\bigwedge s_c\ s_a.\ S'\ s_c\ s_a \Longrightarrow P\ s_c \Longrightarrow \nexists\, t_c.\ (s_c,\ t_c) \in cleanup_c \Longrightarrow \nexists\, t_a.\ (s_a,\ t_a) \in cleanup_a$
shows *refines* (*on-exit* f_c $cleanup_c$) (*on-exit* f_a $cleanup_a$) s_c s_a (*rel-prod R S*)
⟨*proof*⟩

lemma *rel-spec-monad-mono*:
assumes *Q*: *rel-spec-monad R Q f g* **and** *QQ'*: $\bigwedge x\ y.\ Q\ x\ y \Longrightarrow Q'\ x\ y$
shows *rel-spec-monad R Q' f g*
⟨*proof*⟩

lemma *gets-return*: *gets* (λ-. *x*) = *return x*
⟨*proof*⟩

lemma *bind-handle-bind-exception-or-result-conv*:
bind-handle f g h =
bind-exception-or-result f
(λ*Exception e* ⇒ *h e* | *Result v* ⇒ *g v*)
⟨*proof*⟩

lemma *bind-handle-bind-exception-or-result-conv-exn*: *bind-handle f g* (λ*Some e* ⇒ *h e*) =
bind-exception-or-result f
(λ*Exn e* ⇒ *h e* | *Result v* ⇒ *g v*)
⟨*proof*⟩

lemma *try-nested-bind-exception-or-result-conv*:
shows *try* (*f* >>= *g*) =
(*bind-exception-or-result f*
(λ*Exn e* ⇒ (*case e of Inl l* ⇒ *throw l* | *Inr r* ⇒ *return r*)
| *Result v* ⇒ *try* (*g v*)))
⟨*proof*⟩

lemma *try-nested-bind-handle-conv*:
shows *try* (*f* >>= *g*) =
(*bind-handle f* (λ*v*. *try* (*g v*))
(λ*Some e* ⇒ (*case e of Inl l* ⇒ *throw l* | *Inr r* ⇒ *return r*)))
⟨*proof*⟩

definition *no-fail*:: ('*s* ⇒ *bool*) ⇒ ('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒ *bool* **where**
no-fail P f ≡ ∀ *s*. *P s* ⟶ *run f s* ≠ ⊤

definition *no-throw*:: ('*s* ⇒ *bool*) ⇒ ('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒ *bool* **where**

no-throw P f ≡ ∀ *s*. *P s* ⟶ *f* • *s* ?⦃ λ*r t*. ∃ *v*. *r* = *Result v*⦄

definition *no-return*:: (*'s* ⇒ *bool*) ⇒ (*'e*::*default*, *'a*, *'s*) *spec-monad* ⇒ *bool* **where**
no-return P f ≡ ∀ *s*. *P s* ⟶ *f* • *s* ?⦃ λ*r t*. ∃ *e*. *r* = *Exception e* ∧ *e* ≠ *default*⦄

lemma *no-return-exn-def*: *no-return P f* ⟷ (∀ *s*. *P s* ⟶ *f* • *s* ?⦃ λ*r t*. ∃ *e*. *r* = *Exn e*⦄)
⟨*proof*⟩

lemma *no-throw-gets*[*simp*]: *no-throw P* (*gets f*)
⟨*proof*⟩

lemma *no-throw-modify*[*simp*]: *no-throw P* (*modify f*)
⟨*proof*⟩

lemma *no-throw-select*[*simp*]: *no-throw P* (*select f*)
⟨*proof*⟩

lemma *always-progress-select-UNIV*[*simp*]: *always-progress* (*select UNIV*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-xval-catch*:
assumes *fh*: *rel-spec-monad R* (*rel-xval E Q*) *f h*
assumes *gi*: *rel-fun E* (*rel-spec-monad R* (*rel-xval E2 Q*)) *g i*
shows *rel-spec-monad R* (*rel-xval E2 Q*) (*f* <*catch*> *g*) (*h* <*catch*> *i*)
⟨*proof*⟩

16.10 Setup for Tagging

lemma *runs-to-tag*:
(¶ *tag* ⟹ *f* • *s* ⦃ *P* ⦄) ⟹ (*tag* | *f*) • *s* ⦃ *P* ⦄
⟨*proof*⟩

lemma *runs-to-tag-guard*:
fixes *g* :: *'a* ⇒ *bool*
and *s* :: *'a*
assumes *tag* | *g s*
assumes *P* (*Result* ()) *s*
shows (*tag* | *guard g*) • *s* ⦃ *P* ⦄
⟨*proof*⟩

bundle *runs-to-vcg-tagging-setup*
begin

unbundle *basic-vcg-tagging-setup*

lemmas [*runs-to-vcg*] = *runs-to-tag runs-to-tag-guard*

end

lemma *refines-set-state-right*:
refines f (set-state t' ≫= g) s t R ⟷ refines f (g ()) s t' R
⟨*proof*⟩

lemma *refines-bind-modify-right-iff*:
refines f (modify m ≫= g) s t R ⟷ refines f (g ()) s (m t) R
⟨*proof*⟩

lemma *refines-guard-right-iff*:
refines f (guard P) s t R ⟷ (P t ⟶ refines f skip s t R)
⟨*proof*⟩

lemma *refines-select-right-witness*:
x ∈ X ⟹ refines f (g x) s t Q ⟹ refines f ((select X) >>= g) s t Q
⟨*proof*⟩

lemmas *refines-select-right = refines-select-right-witness*

lemma *refines-unknown-right*:
refines f (g x) s t R ⟹ refines f (unknown >>= g) s t R
⟨*proof*⟩

lemma *refines-bind-liftE-modify-iff*:
refines f (do {
liftE g;
modify h
}) x y R =
refines f (do {
g;
modify h
}) x y (λ(x, s') (y, t'). ∀ v. y = Result v ⟶ R (x, s') (Result v, t'))
⟨*proof*⟩

lemma *refines-get-state-right-iff*:
refines f (do {x ← get-state; h x }) s t Q ⟷
refines f (h t) s t Q
⟨*proof*⟩

lemma *refines-gets-the-right-iff*:
refines f (do {x ← gets-the m; h x } :: ('a::default, 'b, 'c) spec-monad) s t Q ⟷
(∀ r. m t = Some r ⟶ refines f (h r) s t Q)
⟨*proof*⟩

lemma *refines-gets-the-right*:
(⋀r. m t = Some r ⟹ refines f (h r) s t Q) ⟹
refines f (do {x ← gets-the m; h x } :: ('a::default, 'b, 'c) spec-monad) s t Q
⟨*proof*⟩

lemma *refines-assert-right-iff*:
refines f (do {x ← assert P; h x} :: ('a::default, 'b, 'c) spec-monad) s t Q ⟷
(P ⟶ refines f (h r) s t Q)
⟨*proof*⟩

lemma *refines-modify-right*:
refines C (modify f) s t R ⟷ (C • s {| λr s'. R (r, s') (Result (), f t) |})
⟨*proof*⟩

lemma *refines-liftE-left*:
refines (liftE f) g s t R ⟷ refines f g s t (λ(Res a, s) b. R (Result a, s) b)
⟨*proof*⟩

lemma *refines-liftE*:
refines f g s t R ⟹
refines (liftE f) (liftE g) s t
(λ(x, s') (y, t'). ∃r q. x = Result r ∧ y = Result q ∧ R (Result r, s') (Result q, t'))
⟨*proof*⟩

lemma *refines-catch-right*:
assumes *refines f g s t (λ(x, s').*
λ(Result y, t') ⇒ R (x, s') (Result y, t')
|(Exn e, t') ⇒ refines (yield x) (h e) s' t' R)
shows *refines f (g <catch> h) s t R*
⟨*proof*⟩

lemma *return-catch*:
(return x <catch> H) = return x
⟨*proof*⟩

lemma *refines-gets-right*:
f • s {| λr s'. R (r, s') (Result (m t), t) |} ⟹ refines f (gets m) s t R
⟨*proof*⟩

lemma *refines-when*:
P ⟷ P' ⟹
(P ⟹ P' ⟹ refines f f' s t R) ⟹
(¬ P ⟹ ¬ P' ⟹ R (Result (), s) (Result (), t)) ⟹
refines (when P f) (when P' f') s t R
⟨*proof*⟩

lemma *refines-case-prod*:
(⋀a b c d. x = (a, b) ⟹ y = (c, d) ⟹ refines (f a b) (g c d) s t R) ⟹
refines (case x of (a, b) ⇒ f a b) (case y of (c, d) ⇒ g c d) s t R
⟨*proof*⟩

lemma *refines-unless*:
(¬ P ⟹ refines f g s t R) ⟹ (P ⟶ R (Result (), s) (Result (), t)) ⟹

refines (*unless P f*) (*unless P g*) *s t R*
⟨*proof*⟩

lemma *refines-ext-left*:
f · *s-A* {‖ *P* ‖} ⟹ *refines g f s-C s-A R* ⟹
refines g f s-C s-A (λ*a b*. *R a b* ∧ *P* (*fst b*) (*snd b*))
⟨*proof*⟩

lemma *refines-return-throw*:
refines (*return x*) (*throw e*) *s t R* ⟷ *R* (*Result x*, *s*) (*Exn e*, *t*)
⟨*proof*⟩

lemma *refines-returnOk-throw*:
refines (*return x*) (*throw e*) *s t R* ⟷ *R* (*Result x*, *s*) (*Exn e*, *t*)
⟨*proof*⟩

lemma *refines-throw-returnOk*:
refines (*throw e*) (*return x*) *s t R* ⟷ *R* (*Exn e*, *s*) (*Result x*, *t*)
⟨*proof*⟩

lemma *refines-condition-right-iff*:
refines f (*condition P g h*) *s t R* ⟷ *refines f* (*if P t then g else h*) *s t R*
⟨*proof*⟩

lemma *refines-gets-theE-right-iff*:
m t = *Some x* ⟹
refines f (*gets-the m* >>= *g*) *s t R* ⟷ *refines f* (*g x*) *s t R*
⟨*proof*⟩

lemma *refines-bind-bindE*:
assumes *f*: *refines f f′ s t Q*
assumes *g′*: ⋀*r s x t*. *Q* (*Result r*, *s*) (*Result x*, *t*) ⟹ *refines* (*g r*) (*g′ x*) *s t R*
assumes *g*: ⋀*r s e t*. *Q* (*Result r*, *s*) (*Exn e*, *t*) ⟹ (*g r*) · *s* {‖ λ*r s′*. *R* (*r*, *s′*) (*Exn e*, *t*) ‖}
shows *refines* ((*bind f g*)::(′*a*, ′*s*) *res-monad*) (*f′* >>= *g′*) *s t R*
⟨*proof*⟩

lemma *refines-whileLoop-cond-exn*:
assumes *C*: ⋀*x s x′ s′*. *R* (*Result x*, *s*) (*Result x′*, *s′*) ⟹ *C x s* = *C′ x′ s′*
assumes *B*: ⋀*x s x′ s′*. *R* (*Result x*, *s*) (*Result x′*, *s′*) ⟹ *C x s* ⟹ *C′ x′ s′* ⟹
refines (*B x*) (*B′ x′*) *s s′ R*
assumes *R-ER*: ⋀*e s x′ s′*. *e* ≠ *default* ⟹ *R* (*Exception e*, *s*) (*Result x′*, *s′*) ⟹ ¬ *C′ x′ s′*
assumes *R-RE*: ⋀*x s e′ s′*. *e′* ≠ *default* ⟹ *R* (*Result x*, *s*) (*Exception e′*, *s′*) ⟹ ¬ *C x s*
assumes *x*: *R* (*Result x*, *s*) (*Result x′*, *s′*)
shows *refines* (*whileLoop C B x*) (*whileLoop C′ B′ x′*) *s s′ R*
⟨*proof*⟩

lemma *refines-on-exit'-right*:
assumes *f*: *refines f f' s s'*
(λ(*r*, *t*) (*r'*, *t'*). *refines skip c' t t'* (λ(*q*, *t*) (*q'*, *t'*). *R*
(*case q of Exception e* ⇒ *Exception e* | *Result* - ⇒ *r*, *t*)
(*case q' of Exception e'* ⇒ *Exception e'* | *Result* - ⇒ *r'*, *t'*)))
shows *refines f* (*on-exit' f' c'*) *s s' R*
⟨*proof*⟩
lemma *runs-to-whileLoop'*:
assumes *B*[*runs-to-vcg*]: ⋀*r s x*. *I x* (*Result r*) *s* ⟹ *C r s* ⟹
(*B r*) · *s* ⦃ λ*q t*. ∃ *y*. *I y q t* ∧ (∀ *a*. *q* = *Result a* ⟶ (*y*, *x*) ∈ *R*) ⦄
assumes *Qr*: ⋀*x r s*. *I x* (*Result r*) *s* ⟹ ¬ *C r s* ⟹ *Q* (*Result r*) *s*
assumes *Ql*: ⋀*x e s*. *I x* (*Exn e*) *s* ⟹ *Q* (*Exn e*) *s*
assumes *R*: *wf R*
assumes *I*: *I x* (*Result r*) *s*
shows (*whileLoop C B r*) · *s* ⦃ *Q* ⦄
⟨*proof*⟩

lemma *runs-to-whileLoop-variant-exn*:
assumes *I*: ⋀*r s x*. *I* (*Result r*) *s x* ⟹
C r s ⟹ (*B r*) · *s* ⦃ λ*q t*. ∃ *y*. *I q t y* ∧ (∀ *a*. *q* = *Result a* ⟶ (*y*, *x*) ∈ *R*) ⦄
assumes *Q-Result*: ⋀*r s c*. *I* (*Result r*) *s c* ⟹ ¬ *C r s* ⟹ *Q* (*Result r*) *s*
assumes *Q-Exn*: ⋀*e s c*. *I* (*Exn e*) *s c* ⟹ *Q* (*Exn e*) *s*
assumes *R*: *wf R*
shows *I* (*Result r*) *s x* ⟹ (*whileLoop C B r*::('*e*, '*a*, '*s*) *exn-monad*) · *s* ⦃ *Q* ⦄
⟨*proof*⟩

lemma *when-when-combine*:
when P (*when Q f*) = *when* (*P* ∧ *Q*) *f*
⟨*proof*⟩

lemma *whileLoop-False*[*simp*]: *whileLoop* (λ- -. *False*) *B a* = *return a*
⟨*proof*⟩

lemma *refines-throw-refl*:
refines (*throw i*) (*throw i*) *s-A s-A* (λ(*r-C*, *t-C*) (*r-A*, *t-A*). *r-C* = *r-A* ∧ *t-C* = *s-A* ∧ *t-A* = *s-A*)
⟨*proof*⟩

lemma *refinesI-runs-to*:
(*f* · *s* ⦃ λ*Res r s'*. *refines* (*return r*) *g s' t R* ⦄) ⟹ *refines f g s t R*
⟨*proof*⟩

lemma *refinesI-runs-to'*:
(*f* · *s* ⦃ λ*r s'*. *refines* (*yield r*) *g s' t R* ⦄) ⟹ *refines f g s t R*
⟨*proof*⟩

lemma *refines-selectE-right*:
x ∈ *X* ⟹ *refines f* (*g x*) *s t R* ⟹ *refines f* (*select X* >>= *g*) *s t R*
⟨*proof*⟩

lemma *refines-unknownE-right*:
refines f (g x) s t R ⟹ refines f (unknown >>= g) s t R
⟨*proof*⟩

lemma *refines-liftE-set-state-right*:
refines f (g ()) s t′ R ⟹ refines f (liftE (set-state t′) >>= g) s t R
⟨*proof*⟩

lemma *refines-modifyE*:
refines (modify m) (modify n) s t (λx y. x = (Result (), m s) ∧ y = (Result (), n t))
⟨*proof*⟩

lemma *refines-modifyE-bindE*:
refines (f ()) (g ()) (m s) (n t) R ⟹ refines (modify m >>= f) (modify n >>= g) s t R
⟨*proof*⟩

lemma *refines-bindE-right*:
assumes *f-g*: *refines f g s t (λ(Res r, s′) (r′, t′).*
(∀ x. r′ = Result x ⟶ refines (return r) (h x) s′ t′ R) ∧
(∀ e. r′ = Exn e ⟶ R (Result r, s′) (Exn e, t′)))
shows *refines f (g >>= h) s t R*
⟨*proof*⟩

lemma *refines-gets-left-iff*:
refines (gets r) g s t R ⟷ refines (return (r s)) g s t R
⟨*proof*⟩

lemma *refines-right*:
assumes *f*: *refines (skip::(unit,′t) res-monad) f s t R* **and**
g: *⋀x t. R (Result (), s) (Result x, t) ⟹ refines g (g′ x) s t Q*
shows *refines (g::(′b, ′t) res-monad) ((f ≫= g′)::(′a, ′s) res-monad) s t Q*
⟨*proof*⟩

lemma *refines-skip-select*:
y ∈ Y ⟹
refines skip (select Y) s t (λx (y′, t′). x = (Result (), s) ∧ t′ = t ∧ y′ = Result y)
⟨*proof*⟩

lemma *refines-modify-left-iff*:
refines (modify m) g s t R ⟷ refines (return ()) g (m s) t R
⟨*proof*⟩

lemma *refines-modify-bind-return-bind*:
Q (m H) c ⟹ (⋀H′ b. Q H′ b ⟹ refines (g ()) (h b) H′ A R) ⟹
refines (modify m >>= g) (return c >>= h) H A R

⟨*proof*⟩

lemma *refines-return-of-runs-to*:
f · *s* {‖ *P* ‖} ⟹ *refines f* (*return x*) *s t* (λ(*a*, *b*) (*c*, *d*). *P a b* ∧ *c* = *Result x* ∧ *d* = *t*)
⟨*proof*⟩

lemma *refines-select-iff*:
refines (*select P*) (*select Q*) *s t R* ⟷ (∀ *x*∈*P*. ∃ *xa*∈*Q*. *R* (*Result x*, *s*) (*Result xa*, *t*))
⟨*proof*⟩

lemma *select-empty-bind*[*simp*]:
select {} >>= *f* = *select* {}
⟨*proof*⟩

lemma *on-exit-eq*:
on-exit f cleanup =
do {
r <− *catch f* (λ*e*.
do {
- <− *liftE* (*state-select cleanup*);
throw e
});
liftE (*state-select cleanup*);
return r
}
⟨*proof*⟩

lemma *select-spec-commute*:
assumes *X*: (*X* = {} ⟹ ∀ *s*. ∃ *t*. (*s*, *t*) ∈ *H*)
shows (*do* { *x* ← *select X*; - ← *state-select H*; *return x* }) =
(*do* { - ← *state-select H*; *select X* })
⟨*proof*⟩

lemma *refines-skip-right*:
assumes *f*: *f* · *s* {‖ λ*r s*′. *R* (*r*, *s*′) (*Result* (), *t*)‖} **shows** *refines f skip s t R*
⟨*proof*⟩

lemma *refines-select-right-iff*:
refines f (*select X*) *s t R* ⟷ (*f* · *s* {‖ λ*r s*′. ∃ *x*∈*X*. *R* (*r*, *s*′) (*Result x*, *t*)‖})
⟨*proof*⟩

lemma *refines-return-right-iff*:
refines f (*return a*) *s t R* ⟷
(*f* · *s* {‖ λ*r s*′. *R* (*r*, *s*′) (*Result a*, *t*) ‖})
⟨*proof*⟩

lemma *refines-skip-right-iff*:
refines f skip s t R $\longleftrightarrow$ *(f · s {| λr s′. R (r, s′) (Result (), t)|})*
⟨*proof*⟩

lemma *select-state-assume*:
(do {
x ← select X;
assume-result-and-state (F x)
}) =
assume-result-and-state (λs. ⋃x∈X. F x s)
⟨*proof*⟩

lemma *getsE-fold-into-modifyE*:
((do { x ← gets f; modify (g x) }) :: ('e, unit,'s) exn-monad) =
modify (λs. g (f s) s)
((do { x ← gets f; - ← modify (g x); h }) :: ('e, 'a, 's) exn-monad) =
do { modify (λs. g (f s) s); h }
⟨*proof*⟩

lemma *refines-whileLoop-right*:
assumes *I*: *R f I s t*
assumes *final*: *⋀f x s t. R f x s t ⟹ ¬ C x t ⟹ f · s {| λy s′. P (y, s′) (Result x, t) |}*
assumes *R-step*: *⋀f x s t. R f x s t ⟹ C x t ⟹*
∃(g::('a, 's) res-monad) f′. run f s = run (bind g f′) s ∧
refines g (B x) s t (λ(Res a, s) (Res b, t). R (f′ a) b s t)
shows *refines f (whileLoop C B I) s t P*
⟨*proof*⟩

lemma *gets-condition*:
gets r >>= (λx. condition (P x) f g) = condition (λs. P (r s) s) f g
⟨*proof*⟩

lemma *condition-select-empty*:
condition P (select {}) f =
do {
b ← gets P;
assume (¬ b);
f
}
⟨*proof*⟩

lemma *not-is-Result-Exn*[*simp*]: *¬ is-Result (Exn e)*
⟨*proof*⟩

lemma *map-value-map-lift-result-return*:
map-value (map-exception-or-result E (λx. x)) (return r) = return r
⟨*proof*⟩

lemma *refines-throw-throw*:
refines (*throw e*) (*throw e'*) *s t R* = *R* (*Exn e*, *s*) (*Exn e'*, *t*)
⟨*proof*⟩

lemma *refines-bind-bind-exn'*:
assumes *f*: *refines f f' s s' Q*
assumes *ll*: ⋀*e e' t t'*. *Q* (*Exception e*, *t*) (*Exn e'*, *t'*) ⟹
e ≠ *default* ⟹
R (*Exception e*, *t*) (*Exn e'*, *t'*)
assumes *lr*: ⋀*e v' t t'*. *Q* (*Exception e*, *t*) (*Result v'*, *t'*) ⟹
e ≠ *default* ⟹
refines (*yield* (*Exception e*)) (*g' v'*) *t t' R*
assumes *rl*: ⋀*v e t t'*. *Q* (*Result v*, *t*) (*Exn e*, *t'*) ⟹
refines (*g v*) (*throw e*) *t t' R*
assumes *rr*: ⋀*v v' t t'*. *Q* (*Result v*, *t*) (*Result v'*, *t'*) ⟹
refines (*g v*) (*g' v'*) *t t' R*
shows *refines* (*bind f g*) (*bind f' g'*) *s s' R*
⟨*proof*⟩
lemma *guard-bind-cong*:
g = *g'* ⟹ (⋀*s*. *g' s* ⟹ *run c s* = *run c' s*) ⟹
do {- <− *guard g*;
c
} =
do {- <− *guard g'*;
c'
}
⟨*proof*⟩

lemma *modify-guard-swap*:
(⋀*s*. *P* (*f s*) = *P s*) ⟹
do {
- <− *modify f*;
guard P
} =
do {
- <− *guard P*;
modify f
}
⟨*proof*⟩

lemma *modify-guard-bind-swap*:
(⋀*s*. *P* (*f s*) = *P s*) ⟹
do {
- <− *modify f*;
- <− *guard P*;
X
} =
do {
- <− *guard P*;

$- <- \mathit{modify}\ f;$
X
}
⟨*proof*⟩

lemma *when-throw-cong*: $P = P' \Longrightarrow (\bigwedge s.\ P' \Longrightarrow \mathit{run}\ Y\ s = \mathit{run}\ Y'\ s) \Longrightarrow$
do { $- <- \mathit{when}\ (\neg\ P)\ (\mathit{throw}\ x);\ Y$ } =
do {$- <- \mathit{when}\ (\neg\ P')\ (\mathit{throw}\ x);\ Y'$ }
⟨*proof*⟩

lemma *condition-throw*:
condition $(\lambda s.\ P)\ (\mathit{throw}\ e)\ B = \mathit{do}$ { *when* $P\ (\mathit{throw}\ e);\ B$ }
⟨*proof*⟩

lemma *map-value-map-lift-result-bind*[*simp*]:
map-value (*map-exception-or-result* (λx::*unit*. *undefined*) f) (*bind* $m\ g$) =
bind m (λx. *map-value* (*map-exception-or-result* (λx. *undefined*) f) ($g\ x$))
⟨*proof*⟩

lemma *map-value-map-result-bind*[*simp*]:
map-value (*map-exception-or-result id* f) (*bind* $m\ g$) =
bind m (λx. *map-value* (*map-exception-or-result id* f) ($g\ x$))
⟨*proof*⟩

lemma *map-value-yield*[*simp*]: *map-value* f (*yield* x) = *yield* ($f\ x$)
⟨*proof*⟩

lemma *map-result-return*[*simp*]: *map-value* (*map-exception-or-result id* f) (*return* x) = *return* ($f\ x$)
⟨*proof*⟩

lemma *map-result-throw*[*simp*]: *map-value* (*map-exception-or-result id* f) (*throw* x) = *throw* x
⟨*proof*⟩

lemma *map-value-compose*[*simp*]: *map-value* f (*map-value* $g\ m$) = (*map-value* (f ∘ g) m)
⟨*proof*⟩

lemma *map-result-compose*[*simp*]:
map-exception-or-result id f o map-exception-or-result id g = map-exception-or-result id (*f o g*)
⟨*proof*⟩

lemma *map-value-condition*:
map-value f (*condition* $c\ g\ h$) = *condition* c (*map-value* $f\ g$) (*map-value* $f\ h$)
⟨*proof*⟩

lemma *whenE-throwError-cong*:

$P = P' \Longrightarrow (\bigwedge s.\ P' \Longrightarrow run\ Y\ s = run\ Y'\ s) \Longrightarrow$
do { - <− *when* (¬ *P*) (*throw x*); *Y* } =
do { - <− *when* (¬ *P'*) (*throw x*); *Y'* }
⟨*proof*⟩

lemma *modifyE-guardE-bindE-swap*:
$(\bigwedge s.\ P\ (f\ s) = P\ s) \Longrightarrow$
do {
- <− *modify f*;
- <− *guard P*;
X
} =
do {
- <− *guard P*;
- <− *modify f*;
X
}
⟨*proof*⟩

lemma *condition-throwError*:
condition (λ*s*. *P*) (*throw e*) *B* = *do* { *when P* (*throw e*); *B* }
⟨*proof*⟩

lemma *refines-bind-condition-check*:
assumes *f*: $f \cdot s \lbrace\!\lbrace\ \lambda Res\ r\ s'.\ s' = s \wedge$
(*P r s* ⟷ (∀ *a*. ¬ *Q t a*)) ∧
(*P r s* ⟶ (∀ *a*. ¬ *Q t a*) ⟶ *R* (*Result e*, *s*) (*Exn q*, *t*)) $\rbrace\!\rbrace$
assumes *g-h*: $\bigwedge x\ y.\ \neg\ P\ x\ s \Longrightarrow Q\ t\ y \Longrightarrow refines\ (g\ x)\ (h\ y)\ s\ t\ R$
shows *refines*
(*do* { *x* ← *f*; *condition* (*P x*) (*return e*) (*g x*) })
(*do* { *x* ← *check q Q*; *h x* }) *s t R*
⟨*proof*⟩

lemma *refinesI-runs-to-vcg*:
$f \cdot s \lbrace\!\lbrace\ P\ \rbrace\!\rbrace \Longrightarrow (\bigwedge r\ t.\ P\ r\ t \Longrightarrow Q\ (r,\ t)\ (r,\ t)) \Longrightarrow refines\ f\ f\ s\ s\ Q$
⟨*proof*⟩

lemma *refines-gets-the-right-fixed*:
assumes *m*:
(*gets-the m* :: ('*a*::*default*, '*b*, '*c*) *spec-monad*) $\cdot\ t \lbrace\!\lbrace\ \lambda r\ t.\ t = t' \wedge r = Result\ r'$
$\rbrace\!\rbrace$
shows *refines f* (*h r'*) *s t' Q* ⟹
refines f (*do* {*x* ← *gets-the m*; *h x* } :: ('*a*::*default*, '*b*, '*c*) *spec-monad*) *s t Q*
⟨*proof*⟩

lemma *refines-gets-the*:
$(\bigwedge b.\ n\ t = Some\ b \Longrightarrow \exists\ a.\ m\ s = Some\ a \wedge Q\ (Result\ a,\ s)\ (Result\ b,\ t)) \Longrightarrow$
refines (*gets-the m*) (*gets-the n*) *s t Q*
⟨*proof*⟩

lemma *refines-bind-gets-the*:
(⋀*x*. *refines* (*f x*) (*g x*) *s s Q*) ⟹
refines (*do* { *x* ← *gets-the m*; *f x* }) (*do* { *x* ← *gets-the m*; *g x* }) *s s Q*
⟨*proof*⟩

lemma *refines-bind-left'*:
refines f f' s t (λ(*Res r*, *s'*) (*Res q*, *t'*). *refines* (*g r*) (*return q*) *s' t' Q*) ⟹
refines (*do* { *x* ← *f*; *g x* }) *f' s t Q*
⟨*proof*⟩

lemma *refines-check-eq*:
refines (*check i p*) (*check i p*) *s-A s-A*
(λ(*r-C*, *t-C*) (*r-A*, *t-A*). *r-C* = *r-A* ∧ *t-C* = *s-A* ∧ *t-A* = *s-A*)
⟨*proof*⟩

lemma *runs-to-check'*[*runs-to-vcg*]:
(⋀*x*. *p s x* ⟹ *Q* (*Result x*) *s*) ⟹ ((⋀*x*. ¬ *p s x*) ⟹ *Q* (*Exn e*) *s*) ⟹ (*check e p*) • *s* {‖*Q*‖}
⟨*proof*⟩

lemma *refines-when-check*:
P = (∀ *a*. ¬ *Q t a*) ⟹
(*P* ⟹ ∀ *a*. ¬ *Q t a* ⟹ *R* (*Exn r*, *s*) (*Exn q*, *t*)) ⟹
(⋀*a*. ¬ *P* ⟹ *Q t a* ⟹ *refines* (*f* ()) (*g a*) *s t R*) ⟹
refines (*bind* (*when P* (*throw r*)) *f*) (*bind* (*check q Q*) *g*) *s t R*
⟨*proof*⟩

lemma *refines-bind-gets-right*:
refines f (*bind* (*gets g*) *h*) *s t R* = *refines f* (*h* (*g t*)) *s t R*
⟨*proof*⟩

lemma *modify-id-return*:
modify id = *return* ()
⟨*proof*⟩

lemma *refines-bind-exn-bind-exn*:
assumes *f*: *refines f f' s s' Q*
assumes *ll*: ⋀*e e' t t'*. *Q* (*Exn e*, *t*) (*Exn e'*, *t'*) ⟹
R (*Exn e*, *t*) (*Exn e'*, *t'*)
assumes *lr*: ⋀*e v' t t'*. *Q* (*Exn e*, *t*) (*Result v'*, *t'*) ⟹
refines (*throw e*) (*g' v'*) *t t' R*
assumes *rl*: ⋀*v e t t'*. *Q* (*Result v*, *t*) (*Exn e*, *t'*) ⟹
refines (*g v*) (*throw e*) *t t' R*
assumes *rr*: ⋀*v v' t t'*. *Q* (*Result v*, *t*) (*Result v'*, *t'*) ⟹
refines (*g v*) (*g' v'*) *t t' R*
shows *refines* (*bind f g*) (*bind f' g'*) *s s' R*
⟨*proof*⟩

lemma *refines-finally-left*:
assumes ∗:
refines f g s t (λ(*r-C*, *t-C*) *A*. ∀ *x*. *r-C* = *Exn x* ∨ *r-C* = *Result x* ⟶ *R* (*Result x*, *t-C*) *A*)
shows *refines* (*finally f*) *g s t R*
⟨*proof*⟩

lemma *refines-map-value-left*:
refines f (*map-value m g*) *s t R* = *refines f g s t* (λ*A* (*r*, *u*). *R A* (*m r*, *u*))
⟨*proof*⟩

lemma *refines-map-value-right*:
refines (*map-value m f*) *g s t R* = *refines f g s t* (λ(*r*, *u*). *R* (*m r*, *u*))
⟨*proof*⟩

lemma *refines-when′*:
(*P* ⟹ *refines f g s t R*) ⟹ (¬*P* ⟶ *R* (*Result* (), *s*) (*Result* (), *t*)) ⟹
refines (*when P f*) (*when P g*) *s t R*
⟨*proof*⟩

lemma *top-eq-fail*: *top* = *fail*
⟨*proof*⟩

lemma *refines-bind-get-state*:
refines (*f s*) (*g t*) *s t R* ⟹ *refines* (*get-state* ⤜ *f*) (*get-state* ⤜ *g*) *s t R*
⟨*proof*⟩

lemma *refines-bind-assert*:
refines (*assert P* ⤜ *f*) (*assert Q* ⤜ *g*) *s t R*
if ∗: *Q* ⟹ *P Q* ⟹ *P* ⟹ *refines* (*f* ()) (*g* ()) *s t R*
⟨*proof*⟩

lemma *refines-bind-set-state*: *refines* (*set-state s′* ⤜ *f*) (*set-state t′* ⤜ *g*) *s t R*
if ∗: *refines* (*f* ()) (*g* ()) *s′ t′ R*
⟨*proof*⟩

lemma *refines-bind-select*: *refines* (*select P* ⤜ *f*) (*select Q* ⤜ *g*) *s t R*
if ∗: *sim-set S P Q* ⋀*v v′*. *S v v′* ⟹ *refines* (*f v*) (*g v′*) *s t R*
⟨*proof*⟩

lemma *refines-assert-result-and-state′*:
refines (*assert-result-and-state f*) (*assert-result-and-state g*) *s t*
(*rel-prod* (*rel-exception-or-result E R*) *S*)
if ∗: *S s t sim-set* (*rel-prod R S*) (*f s*) (*g t*) *f s* = {} ⟹ *g t* = {}
⟨*proof*⟩

lemma *runs-to-partial-assert-result-and-state-iff*:
assert-result-and-state f ⋆ *s* ?⦃ *P* ⦄ ⟷ (∀ (*a*, *s*)∈*f s*. *P* (*Result a*) *s*)
⟨*proof*⟩

lemma *bind-state-assume-eq*: *run* (*do* { (-::*unit*) <− *assume-result-and-state P*; *f* }) *s* = *run f s*
if ⋀*s s'*. ((), *s'*) ∈ *P s* ⟹ *s'* = *s* ((), *s*) ∈ *P s*
⟨*proof*⟩

lemma *refines-bindE-right'*:
assumes *f*: *refines f f' s s' Q*
assumes *ll*: ⋀*e e' t t'*. *Q* (*Exn e*, *t*) (*Exn e'*, *t'*) ⟹ *R* (*Exn e*, *t*) (*Exn e'*, *t'*)
assumes *lr*: ⋀*e v' t t'*. *Q* (*Exn e*, *t*) (*Result v'*, *t'*) ⟹ *refines* (*throw e*) (*g' v'*) *t t' R*
assumes *rl*: ⋀*v e' t t'*. *Q* (*Result v*, *t*) (*Exn e'*, *t'*) ⟹ *refines* ((*return v*)) (*throw e'*) *t t' R*
assumes *rr*: ⋀*v v' t t'*. *Q* (*Result v*, *t*) (*Result v'*, *t'*) ⟹ *refines* ((*return v*)) (*g' v'*) *t t' R*
shows *refines f* (*f'* >>= *g'*) *s s' R*
⟨*proof*⟩

lemma *refines-exec-modify-step-right*:
assumes *refines* (*return x*) *g s* (*upd t*) *Q*
shows *refines* (*return x*) (*do* { - <− (*modify* (*upd*)); *g* }) *s t Q*
⟨*proof*⟩

lemma *runs-to-whileLoop-variant*:
assumes *I*: ⋀*r s c*. *I r s c* ⟹
C r s ⟹ (*B r*) · *s* {‖ λ*Res q t*. ∃ *c'*. *I q t c'* ∧ (*c'*, *c*) ∈ *R* ‖}
assumes *Q*: ⋀*r s c*. *I r s c* ⟹ ¬ *C r s* ⟹ *Q r s*
assumes *R*: *wf R*
shows *I r s c* ⟹ (*whileLoop C B r*) · *s* {‖ λ*Res r s*. *Q r s* ‖}
⟨*proof*⟩

lemma *catch-throw*[*simp*]: *catch f throw* = *f*
⟨*proof*⟩

lemma *select-singleton*: *select* {*x*} = *return x*
⟨*proof*⟩

lemma *gets-gets-combine*:
(*do* { *x* ← *gets f*; *gets* (*g x*) }) = *gets* (λ*h*. *g* (*f h*) *h*)
⟨*proof*⟩

lemma *gets-gets*:
(*do* { *x* ← *gets f*; *y* ← *gets* (*g x*); *C y* }) =
(*do* { *y* ← *gets* (λ*h*. *g* (*f h*) *h*); *C y* })
⟨*proof*⟩

lemma *refines-whileLoop-exn-strong*:
assumes *C*: ⋀*v s v' s'*.
R (*v*, *s*) (*v'*, *s'*) ⟹

$(\exists\, x.\ v = \mathit{Result}\ x \wedge C\ x\ s) \longleftrightarrow (\exists\, x'.\ v' = \mathit{Result}\ x' \wedge C'\ x'\ s')$
assumes B: $\bigwedge x\ s\ x'\ s'.\ R\ (\mathit{Result}\ x,\ s)\ (\mathit{Result}\ x',\ s') \Longrightarrow C\ x\ s \Longrightarrow C'\ x'\ s' \Longrightarrow$
$\mathit{refines}\ (B\ x)\ (B'\ x')\ s\ s'\ R$
assumes Q: $\bigwedge x\ s\ x'\ s'.\ R\ (x,\ s)\ (x',\ s') \Longrightarrow$
$(\bigwedge r.\ x = \mathit{Result}\ r \Longrightarrow \neg\ C\ r\ s) \Longrightarrow (\bigwedge r'.\ x' = \mathit{Result}\ r' \Longrightarrow \neg\ C'\ r'\ s') \Longrightarrow$
$Q\ (x,\ s)\ (x',\ s')$
assumes x: $R\ (\mathit{Result}\ x,\ s)\ (\mathit{Result}\ x',\ s')$
shows $\mathit{refines}\ (\mathit{whileLoop}\ C\ B\ x)\ (\mathit{whileLoop}\ C'\ B'\ x')\ s\ s'\ Q$
⟨*proof*⟩

lemma *not-default-Some*:
$x \neq \mathit{default} \longleftrightarrow (\exists\, a.\ x = \mathit{Some}\ a)$
⟨*proof*⟩

lemma *gets-fold-into-modify*:
$\mathit{do}\ \{\ x \leftarrow \mathit{gets}\ f;\ \text{-} \leftarrow \mathit{modify}\ (g\ x);\ h\ \} = \mathit{do}\ \{\ \mathit{modify}\ (\lambda s.\ g\ (f\ s)\ s);\ h\ \}$
$\mathit{do}\ \{\ x \leftarrow \mathit{gets}\ f;\ \mathit{modify}\ (g\ x)\ \} = \mathit{modify}\ (\lambda s.\ g\ (f\ s)\ s)$
⟨*proof*⟩

lemma *assume-eq-select*: $\mathit{assume}\ p = \mathit{select}\ \{x.\ p\}$
⟨*proof*⟩

lemma *condition-bot*:
$\mathit{condition}\ P\ \bot\ f =$
$\mathit{do}\ \{$
$b \leftarrow \mathit{gets}\ P;$
$\mathit{assume}\ (\neg\ b);$
f
$\}$
⟨*proof*⟩

lemma *unless-not*: $\mathit{unless}\ (\neg\ P) = \mathit{when}\ P$
⟨*proof*⟩

lemma *select-bind-select*:
$\mathit{select}\ X >>= (\lambda x.\ \mathit{select}\ (Y\ x)) = \mathit{select}\ \{y.\ \exists\, x{\in}X.\ y{\in}Y\ x\}$
⟨*proof*⟩

lemma *case-exception-or-result-iff*:
$\mathit{case\text{-}exception\text{-}or\text{-}result}\ P\ Q\ r \longleftrightarrow$
$(\forall\, x.\ r = \mathit{Result}\ x \longrightarrow Q\ x) \wedge (\forall\, e.\ r = \mathit{Exception}\ e \longrightarrow e \neq \mathit{default} \longrightarrow P\ e)$
⟨*proof*⟩

lemma *case-exception-or-result-lam-iff*:
$\mathit{case\text{-}exception\text{-}or\text{-}result}\ P\ Q\ r =$
$(\lambda s.\ (\forall\, x.\ r = \mathit{Result}\ x \longrightarrow Q\ x\ s) \wedge (\forall\, e.\ r = \mathit{Exception}\ e \longrightarrow e \neq \mathit{default} \longrightarrow P\ e\ s))$
⟨*proof*⟩

lemma *runs-to-cong-split*:
(∀ *a s*. *P* (*Result a*) *s* ⟷ *Q* (*Result a*) *s*) ⟹
(∀ *e s*. *e* ≠ *default* ⟶ *P* (*Exception e*) *s* ⟷ *Q* (*Exception e*) *s*) ⟹
f ⋅ *s* ⦃ *P* ⦄ ⟷ *f* ⋅ *s* ⦃ *Q* ⦄
⟨*proof*⟩

lemma *left-total-rel-map*: *left-total* (*rel-map f*)
⟨*proof*⟩

lemma *rel-exception-or-result-Exception-right*:
e ≠ *default* ⟹
rel-exception-or-result E A x (*Exception e*) ⟷
(∃ *e'*. *e'* ≠ *default* ∧ *x* = *Exception e'* ∧ *E e' e*)
⟨*proof*⟩

lemma *rel-exception-or-result-Exception-left*:
e ≠ *default* ⟹
rel-exception-or-result E A (*Exception e*) *y* ⟷
(∃ *e'*. *e'* ≠ *default* ∧ *y* = *Exception e'* ∧ *E e e'*)
⟨*proof*⟩

lemma *rel-exception-or-result-Result-right*:
rel-exception-or-result E A x (*Result a*) ⟷ (∃ *a'*. *x* = *Result a'* ∧ *A a' a*)
⟨*proof*⟩

lemma *rel-exception-or-result-Result-left*:
rel-exception-or-result E A (*Result a*) *x* ⟷ (∃ *a'*. *x* = *Result a'* ∧ *A a a'*)
⟨*proof*⟩

lemma *right-unique-rel-exception-of-regular*:
right-unique E ⟹ *right-unique F* ⟹ *right-unique* (*rel-exception-or-result E F*)
⟨*proof*⟩

lemma *left-total-rel-exception-of-regular*:
(⋀*x*. *x* ≠ *default* ⟹ ∃ *y*. *y* ≠ *default* ∧ *E x y*) ⟹ *left-total F* ⟹
left-total (*rel-exception-or-result E F*)
⟨*proof*⟩

lemma *left-unique-rel-map*: *inj f* ⟹ *left-unique* (*rel-map f*)
⟨*proof*⟩

lemma *rel-exception-or-result-sym*:
rel-exception-or-result E^{-1-1} F^{-1-1} *s t* ⟹ *rel-exception-or-result E F t s*
⟨*proof*⟩

lemma *rel-post-state-inv*: *rel-post-state* R^{-1-1} *t s* ⟷ *rel-post-state R s t*
⟨*proof*⟩

lemma *rel-spec-inv*: *rel-spec B A t s* R^{-1-1} ⟷ *rel-spec A B s t R*

⟨*proof*⟩

lemma *refines-gets'-right*:
refines f (*do* { *x* ← *gets g*; *h x* }) *s t R* ⟷ *refines f* (*h* (*g t*)) *s t R*
⟨*proof*⟩

lemma *sim-post-state-Success-right'*:
sim-post-state R f (*Success s*) ⟷ *f* ≤ *Success* {*x*. ∃ *y*∈*s*. *R x y*}
⟨*proof*⟩

lemma *Sup-eq-top-post-state*[*simp*]:
Sup (*X* :: *'a post-state set*) = ⊤ ⟷ ⊤ ∈ *X*
⟨*proof*⟩

lemma *sim-post-state-Success-right*:
sim-post-state R x (*Success t*) ⟷ (∃ *s*. *x* = *Success s* ∧ (∀ *a*∈*s*. ∃ *b*∈*t*. *R a b*))
⟨*proof*⟩

lemma *sim-post-state-Success-left*:
sim-post-state R (*Success s*) *y* ⟷ (∀ *t*. *y* = *Success t* ⟶ (∀ *a*∈*s*. ∃ *b*∈*t*. *R a b*))
⟨*proof*⟩

lemma *sim-post-state-bind-Success-right*:
sim-post-state R f (*bind-post-state g* (λ*x*. *Success* (*p x*))) ⟷
sim-post-state (λ*a b*. ∃ *x*∈*p b*. *R a x*) *f g*
⟨*proof*⟩

lemma *sim-post-state-bind-Success-left*:
sim-post-state (λ*x y*. ∀ *a*∈*p x*. *R a y*) *f g* ⟹
sim-post-state R (*bind-post-state f* (λ*x*. *Success* (*p x*))) *g*
⟨*proof*⟩

lemma *rel-post-state-Failure-left*[*simp*]:
rel-post-state R Failure y ⟷ *y* = *Failure*
⟨*proof*⟩

lemma *rel-post-state-Failure-right*[*simp*]:
rel-post-state R x Failure ⟷ *x* = *Failure*
⟨*proof*⟩

lemma *rel-post-state-Success-right*:
rel-post-state R x (*Success t*) ⟷ (∃ *s*. *x* = *Success s* ∧ *rel-set R s t*)
⟨*proof*⟩

lemma *rel-post-state-Success-left*:
rel-post-state R (*Success s*) *y* ⟷ (∃ *t*. *y* = *Success t* ∧ *rel-set R s t*)
⟨*proof*⟩

lemma *rel-post-state-bind-Success-left*:
assumes ∗: ⋀*a b*. *rel-set R* (⋃ (*p* ‘ *a*)) *b* = *rel-set Q a b*

shows *rel-post-state R* (*bind-post-state f* (λ*x*. *Success* (*p x*))) *g* ⟷ *rel-post-state Q f g*
⟨*proof*⟩

lemma *sim-post-state-mono*:
sim-post-state R f g ⟹ (⋀*x y*. *R x y* ⟹ *Q x y*) ⟹ *sim-post-state Q f g*
⟨*proof*⟩

lemma *rel-set-iff-eq*:
left-total R ⟹ *right-unique R* ⟹ *rel-set R a b* ⟷ *b* = (⋃*x*∈*a*. {*y*. *R x y*})
⟨*proof*⟩

lemma *left-total-rel-exception-or-result*:
left-total A ⟹ (∀ *x*. ∃ *y*. *y* ≠ *default* ∧ *E x y*) ⟹ *left-total* (*rel-exception-or-result E A*)
⟨*proof*⟩

lemma *bind-return-eq-map-value*:
(*do* { *x* ← *f*; *return* (*F x*) }) = *map-value* (*map-exception-or-result id F*) *f*
⟨*proof*⟩

lemma *refines-bind-return-iff*:
fixes *f* :: (-, -) *res-monad* **and** *g* :: (-, -) *res-monad*
shows *refines* (*do* { *x* ← *f*; *return* (*F x*) }) (*do* { *x* ← *g*; *return* (*G x*) }) *s t R* ⟷
refines f g s t (λ(*Res r*, *s*′) (*Res q*, *t*′). *R* (*Result* (*F r*), *s*′) (*Result* (*G q*), *t*′))
⟨*proof*⟩

lemma *refines-bind-return-left-iff*:
fixes *g* :: (-, -) *res-monad*
shows *refines f* (*do* { *x* ← *g*; *return* (*G x*) }) *s t R* ⟷
refines f g s t (λ*x* (*Res q*, *t*′). *R x* (*Result* (*G q*), *t*′))
⟨*proof*⟩

lemma *refines-bind-return-right-iff*:
fixes *f* :: (-, -) *res-monad*
shows *refines* (*do* { *x* ← *f*; *return* (*G x*) }) *g s t R* ⟷
refines f g s t (λ(*Res r*, *s*′) *y*. *R* (*Result* (*G r*), *s*′) *y*)
⟨*proof*⟩

lemma *refines-runs-to-strenghten*:
refines f g s t R ⟹ *g* • *t* {‖ *P* ‖} ⟹ (⋀*a x t*′. *R a* (*x*, *t*′) ⟹ *P x t*′ ⟹ *Q a* (*x*, *t*′)) ⟹
refines f g s t Q
⟨*proof*⟩

lemma *refines-liftE-right*:
refines f (*liftE g*) *s t R* ⟷ *refines f g s t* (λ*a* (*Res b*, *s*). *R a* (*Result b*, *s*))
⟨*proof*⟩

lemma *refines-return-iff*:
refines (*return x*) (*return y*) *s t R* ⟷ *R* (*Result x*, *s*) (*Result y*, *t*)
⟨*proof*⟩

lemma *on-exit'-res-monad*:
fixes *f* :: ('*a*, '*s*) *res-monad*
shows *on-exit' f c* = *do* { *r* ← *f*; *c*; *return r* }
⟨*proof*⟩

lemma *ignoreE-case-prod*[*ignoreE-simps*]:
ignoreE (*case-prod f p*) = (*case p of* (*a*, *b*) ⇒ *ignoreE* (*f a b*))
⟨*proof*⟩

lemma *ignoreE-bind-handle*[*ignoreE-simps*]:
ignoreE (*bind-handle f g h*) = *bind-handle f* (λ*x*. *ignoreE* (*g x*)) (λ*x*. *ignoreE* (*h x*))
⟨*proof*⟩

lemma *ignoreE-assume-result-and-state*[*ignoreE-simps*]:
ignoreE (*assume-result-and-state X*) = *assume-result-and-state X*
⟨*proof*⟩

lemma *ignoreE-assert-result-and-state*[*ignoreE-simps*]:
ignoreE (*assert-result-and-state X*) = *assert-result-and-state X*
⟨*proof*⟩

lemma *ignoreE-throw-exception-or-result*[*ignoreE-simps*]:
x ≠ *default* ⟹ *ignoreE* (*throw-exception-or-result x*) = ⊥
⟨*proof*⟩

lemma *bind-handle-cong*:
assumes *eq*: *f* = *f'* *g* = *g'*
shows (⋀*x*. *x* ≠ *default* ⟹ *h x* = *h' x*) ⟹
bind-handle f g h = *bind-handle f' g' h'*
⟨*proof*⟩

lemma *bind-handle-ignoreE*[*ignoreE-simps*]: *bind-handle* (*ignoreE f*) *g h* = *bind* (*ignoreE f*) *g*
⟨*proof*⟩

lemma *bind-handle-eq-bind*:
bind-handle (*f* :: (-, -) *res-monad*) *g h* = *bind f g*
⟨*proof*⟩

lemma *bind-handle-bot-eq-ignoreE*[*ignoreE-simps*]:
bind-handle f g (λ*x*. ⊥) = *bind* (*ignoreE f*) *g*
⟨*proof*⟩

lemma *state-select-bot*: $\forall\, s.\ \exists\, t.\ (s,\ t) \in h \Longrightarrow do\ \{\ x \leftarrow state\text{-}select\ h;\ \bot\ \} = \bot$
⟨*proof*⟩

lemma *runs-to-cong*:
$f \cdot s\ ?\{\!|\ \lambda r\ t.\ P\ r\ t \longleftrightarrow Q\ r\ t\ |\!\} \Longrightarrow f \cdot s \{\!|\ P\ |\!\} \longleftrightarrow f \cdot s \{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *run-ignoreE-on-exit'*:
assumes *f-h*: $f \cdot s\ ?\{\!|\ \lambda r\ t.\ \forall\, e.\ e \neq default \longrightarrow r = Exception\ e \longrightarrow h \cdot t\ \{\!|\lambda\text{-}\ \text{-}.\ True\ |\!\}\ |\!\}$
shows $run\ (ignoreE\ (on\text{-}exit'\ f\ h))\ s = run\ (on\text{-}exit'\ (ignoreE\ f)\ (ignoreE\ h))\ s$
⟨*proof*⟩

lemma *ignoreE-on-exit'*[*ignoreE-simps*]:
$(\bigwedge s.\ f \cdot s\ ?\{\!|\ \lambda r\ t.\ \forall\, e.\ e \neq default \longrightarrow r = Exception\ e \longrightarrow h \cdot t\ \{\!|\lambda\text{-}\ \text{-}.\ True\ |\!\}\ |\!\}) \Longrightarrow$
$ignoreE\ (on\text{-}exit'\ f\ h) = on\text{-}exit'\ (ignoreE\ f)\ (ignoreE\ h)$
⟨*proof*⟩

lemma *ignoreE-on-exit*[*ignoreE-simps*]:
$\forall\, s.\ \exists\, t.\ (s,\ t) \in h \Longrightarrow ignoreE\ (on\text{-}exit\ f\ h) = on\text{-}exit\ (ignoreE\ f)\ h$
⟨*proof*⟩

lemma *sim-post-state-vmap-post-state1*:
$sim\text{-}post\text{-}state\ (\lambda x\ y.\ \forall\, x'.\ f\ x' = x \longrightarrow R\ x'\ y)\ p\ q \Longrightarrow$
$sim\text{-}post\text{-}state\ R\ (vmap\text{-}post\text{-}state\ f\ p)\ q$
⟨*proof*⟩

lemma *refines-vmap-value-left*:
$refines\ f\ g\ s\ t\ (\lambda(r,\ s)\ y.\ \forall\, x.\ r = m\ x \longrightarrow R\ (x,\ s)\ y) \Longrightarrow$
$refines\ (vmap\text{-}value\ m\ f)\ g\ s\ t\ R$
⟨*proof*⟩

lemma *refines-ignoreE-left*:
$refines\ f\ g\ s\ t\ (\lambda(r,\ s)\ y.\ \forall\, x.\ r = Result\ x \longrightarrow R\ (Result\ x,\ s)\ y) \Longrightarrow$
$refines\ (ignoreE\ f)\ g\ s\ t\ R$
⟨*proof*⟩

lemma *finally-when-throw*:
$finally\ (when\ P\ (throw\ x) >>= f) =$
$condition\ (\lambda\text{-}.\ P)\ (return\ x)\ (finally\ (f\ ()))$
⟨*proof*⟩

lemma *finally-gets-bind*:
$finally\ (do\ \{\ r \leftarrow gets\ X;\ Y\ r\ \}) = do\ \{\ r \leftarrow gets\ X;\ finally\ (Y\ r)\ \}$
⟨*proof*⟩

lemma *finally-modify-bind*:
$finally\ (do\ \{\ r \leftarrow modify\ X;\ Y\ r\ \}) = do\ \{\ r \leftarrow modify\ X;\ finally\ (Y\ r)\ \}$

⟨*proof*⟩

lemma *finally-return*: *finally* (*return x*) = *return x*
⟨*proof*⟩

lemma *f-catch-throw*: (*f* <*catch*> *throw*) = *f*
⟨*proof*⟩

end

theory *Reaches*
imports *Spec-Monad*
begin

The core notions on spec monads are

- Properties of a monad: *runs-to*, *runs-to-partial*
- Refinement / Simulation of monads: *refines*, *rel-spec* and *rel-spec-monad*.

  It is considered good style to use these more abstract concepts as much as possible.

  Next we introduce some notions that are more in a 'pointwise' spirit in the sense that they talk about a particular outcome of a monadic computation, e.g. that a particular state is a reachable outcome of a monadic computation.

  There is nothing *wrong* with these notions but we consider them as less elegant and more 'brute force'. So we encourage to think twice before using them and prefer the more abstract notions.

primrec *outcomes* :: *'a post-state* ⇒ *'a set* **where**
outcomes Failure = {}
| *outcomes* (*Success X*) = *X*

lemma *le-post-state-iff*:
p ≤ *q* ⟷ (*q* ≠ *Failure* ⟶ (*p* ≠ *Failure* ∧ *outcomes p* ⊆ *outcomes q*))
⟨*proof*⟩

lemma *outcomes-map*[*simp*]: *outcomes* (*map-post-state f x*) = *f* ‘ *outcomes x*
⟨*proof*⟩

lemma *map-post-state-eq-Failure*[*simp*]:
map-post-state f x = *Failure* ⟷ *x* = *Failure*
⟨*proof*⟩

lemma *outcomes-Sup*[*simp*]: *Failure* ∉ *F* ⟹ *outcomes* (*Sup F*) = (⋃*f*∈*F*. *outcomes f*)

⟨*proof*⟩

lemma *mem-outcomes-iff*: $x \in \mathit{outcomes}\ p \longleftrightarrow (\exists X.\ p = \mathit{Success}\ X \wedge x \in X)$
⟨*proof*⟩

lemma *outcomes-Sup-eq-if*:
$\mathit{outcomes}\ (\mathit{Sup}\ X) = (\mathit{if}\ \mathit{Failure} \in X\ \mathit{then}\ \{\}\ \mathit{else}\ \bigcup x{\in}X.\ \mathit{outcomes}\ x)$
⟨*proof*⟩

lemmas *runs-to-holds-def* = *runs-to.rep-eq*
lemmas *runs-to-partial-holds-partial-def* = *runs-to-partial.rep-eq*

16.11 *succeeds* and *reaches*

definition *succeeds* :: ('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒ '*s* ⇒ *bool* **where**
$\mathit{succeeds}\ f\ s \longleftrightarrow \mathit{run}\ f\ s \neq \top$

definition *reaches* :: ('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒ '*s* ⇒ ('*e*, '*a*) *exception-or-result* ⇒ '*s* ⇒ *bool* **where**
$\mathit{reaches}\ f\ s\ r'\ s' \longleftrightarrow (r', s') \in \mathit{outcomes}\ (\mathit{run}\ f\ s)$

There seems to a similarity between what happens in HOL with the dualism of sets as type vs. the characterisation as predicate as expressed in *Collect*. Similar *'a post-state* has a set in the *Success* case. In particular with *holds-post-state* we switch to the predicate view. Which is then continued in *runs-to* which is the predicate view and the set view with *reaches*, which is more the element wise set view similar to $x \in X$ and finally culminates in $(?f = ?g) = (\forall P\ s.\ (?f \cdot s \{\!|\ P\ |\!\}) = (?g \cdot s \{\!|\ P\ |\!\}))$. The predicate view seems to be the one we finally aim for. So maybe we could get completely rid of the set like things and start of with a predicate already in *'a post-state*?

lemma *runs-to-partial-def-old*:
$\mathit{runs\text{-}to\text{-}partial}\ f\ s\ Q \longleftrightarrow (\mathit{succeeds}\ f\ s \longrightarrow (\forall r\ t.\ \mathit{reaches}\ f\ s\ r\ t \longrightarrow Q\ r\ t))$
⟨*proof*⟩

lemma *runs-to-def-old*: $\mathit{runs\text{-}to}\ f\ s\ Q \longleftrightarrow \mathit{succeeds}\ f\ s \wedge (\forall r\ t.\ \mathit{reaches}\ f\ s\ r\ t \longrightarrow Q\ r\ t)$
⟨*proof*⟩

lemma *runs-to-succeeds-runsto-partial-conv*:
$\mathit{runs\text{-}to}\ f\ s\ Q \longleftrightarrow (\mathit{succeeds}\ f\ s \wedge \mathit{runs\text{-}to\text{-}partial}\ f\ s\ Q)$
⟨*proof*⟩

lemma *run-Success-succeeds*: $\mathit{run}\ f\ s = \mathit{Success}\ x \Longrightarrow \mathit{succeeds}\ f\ s$
⟨*proof*⟩

lemma *runs-to-partial-le-outcomes-conv*:
$\mathit{runs\text{-}to\text{-}partial}\ f\ s\ Q \longleftrightarrow (\mathit{outcomes}\ (\mathit{run}\ f\ s)) \leq \{(r,t).\ Q\ r\ t\}$

⟨*proof*⟩

lemma *not-succeeds-empty-outcomes*: ¬ *succeeds f s* ⟹ *outcomes* (*run f s*) = {}
⟨*proof*⟩

lemma *reaches-succeeds*: *reaches f s r t* ⟹ *succeeds f s*
⟨*proof*⟩

lemma *always-progress-succeeds-reaches-conv*:
always-progress f ⟷ (∀ *s*. *succeeds f s* ⟶ (∃ *r t*. *reaches f s r t*))
⟨*proof*⟩

lemma *le-succeedsD*: *g* ≤ *f* ⟹ *succeeds f s* ⟹ *succeeds g s*
⟨*proof*⟩

lemma *outcomes-succeeds-run-conv*: *outcomes* (*run f s*) = *X* ⟹ *succeeds f s* ⟹ *run f s* = *Success X*
⟨*proof*⟩

lemma *succeeds-outcomes-run-eqI*: *succeeds f s* ⟷ *succeeds g s* ⟹
(*succeeds f s* ⟹ *succeeds g s* ⟹ (*outcomes* (*run f s*) = *outcomes* (*run g s*)))
⟹
run f s = *run g s*
⟨*proof*⟩

lemma *succeeds-outcomes-spec-monad-eqI*:
(⋀*s*. *succeeds f s* ⟷ *succeeds g s*) ⟹
(⋀*s*. *succeeds f s* ⟹ *succeeds g s* ⟹ (*outcomes* (*run f s*) = *outcomes* (*run g s*)))
⟹
f = *g*
⟨*proof*⟩

lemma *succeeds-reaches-spec-monad-eqI*:
(⋀*s*. *succeeds f s* ⟷ *succeeds g s*) ⟹
(⋀*s r t*. *succeeds f s* ⟹ *succeeds g s* ⟹ (*reaches f s r t* ⟷ *reaches g s r t*)) ⟹
f = *g*
⟨*proof*⟩

lemma *succeeds-runs-to-iff*: *succeeds f s* ⟷ *runs-to f s* (λ- -. *True*)
⟨*proof*⟩

named-theorems *outcomes-spec-monad Simplification rules for outcomes of Spec monad*

16.11.1 Relational rewriting for Monads

lemma *refines-def-old*:
refines f g s t R ⟷
(*succeeds g t* ⟶ *succeeds f s* ∧

$(\forall\, r\ s'.\ reaches\ f\ s\ r\ s' \longrightarrow (\exists\, x\ t'.\ reaches\ g\ t\ x\ t' \wedge R\ (r,\ s')\ (x,\ t'))))$
⟨*proof*⟩

lemma *rel-specI*:
assumes *succeeds f s* ⟷ *succeeds g t*
assumes $\bigwedge r\ s'.\ reaches\ f\ s\ r\ s' \Longrightarrow \exists\, q\ t'.\ reaches\ g\ t\ q\ t' \wedge R\ (r,\ s')\ (q,\ t')$
assumes $\bigwedge q\ t'.\ reaches\ g\ t\ q\ t' \Longrightarrow \exists\, r\ s'.\ reaches\ f\ s\ r\ s' \wedge R\ (r,\ s')\ (q,\ t')$
shows *rel-spec f g s t R*
⟨*proof*⟩

lemma *rel-specD-succeeds*:
rel-spec f g s t R ⟹ *succeeds f s* ⟷ *succeeds g t*
⟨*proof*⟩

lemma *rel-specD-reaches1*:
rel-spec f g s t R ⟹ *reaches f s r s'* ⟹ $\exists\, q\ t'.$ *reaches g t q t'* $\wedge$ $R\ (r,\ s')\ (q,\ t')$
⟨*proof*⟩

lemma *rel-specD-reaches2*:
rel-spec f g s t R ⟹ *reaches g t q t'* ⟹ $\exists\, r\ s'.$ *reaches f s r s'* $\wedge$ $R\ (r,\ s')\ (q,\ t')$
⟨*proof*⟩

lemma *refines-iff*:
refines f g s t R ⟷
(*succeeds g t* ⟶ *succeeds f s*) ∧
(*succeeds g t* ⟶ *succeeds f s* ⟶
$(\forall\, r\ s'.\ reaches\ f\ s\ r\ s' \longrightarrow (\exists\, q\ t'.\ reaches\ g\ t\ q\ t' \wedge R\ (r,\ s')\ (q,\ t')))))$
⟨*proof*⟩

lemma *refinesI*:
assumes *succeeds g t* ⟹ *succeeds f s*
assumes $\bigwedge r\ s'.$ *succeeds g t* ⟹ *succeeds f s* ⟹ *reaches f s r s'* ⟹
$(\exists\, q\ t'.\ reaches\ g\ t\ q\ t' \wedge R\ (r,\ s')\ (q,\ t'))$
shows *refines f g s t R*
⟨*proof*⟩

lemma *refinesD-succeeds*:
refines f g s t R ⟹ *succeeds g t* ⟹ *succeeds f s*
⟨*proof*⟩

lemma *refinesD-reaches*:
refines f g s t R ⟹ *reaches f s r s'* ⟹ *succeeds g t*
⟹ $\exists\, q\ t'.$ *reaches g t q t'* $\wedge$ $R\ (r,\ s')\ (q,\ t')$
⟨*proof*⟩

lemma *refines-strengthen′*:
assumes *refines f g s t R*
assumes $\bigwedge r\ u\ q\ v.$ *reaches f s r u* ⟹ *reaches g t q v* ⟹ *succeeds f s* ⟹
succeeds g t

$\Longrightarrow R\ (r,\ u)\ (q,\ v) \Longrightarrow Q\ (r,\ u)\ (q,\ v)$
shows *refines f g s t Q*
⟨*proof*⟩

lemma *always-progressD*: *always-progress f* $\Longrightarrow$ *succeeds f s* $\Longrightarrow$ $\exists\, r\ t.$ *reaches f s r t*
⟨*proof*⟩

lemma *Ex-reaches*: *f* · *s* ⦃ *P* ⦄ $\Longrightarrow$ *always-progress f* $\Longrightarrow$ $\exists\, r\ t.$ *reaches f s r t*
⟨*proof*⟩

lemma *witness-outcomes-succeeds*: *x* ∈ *outcomes* (*run f s*) $\Longrightarrow$ *succeeds f s*
⟨*proof*⟩

lemma *runs-toD2*: *f* · *s* ⦃ *P* ⦄ $\Longrightarrow$ *reaches f s r t* $\Longrightarrow$ *P r t*
⟨*proof*⟩

lemma *runs-toD2-res*: **fixes** *f*:: (*'a*, *'s*) *res-monad*
shows *f* · *s* ⦃λ*Res r*. *P r* ⦄ $\Longrightarrow$ *reaches f s* (*Result r*) *t* $\Longrightarrow$ *P r t*
⟨*proof*⟩

lemma *rel-post-state-runI*:
assumes *succeeds f s* $\longleftrightarrow$ *succeeds g t*
assumes *succeeds f s* $\Longrightarrow$ *succeeds g t* $\Longrightarrow$ *rel-set R* (*outcomes* (*run f s*)) (*outcomes* (*run g t*))
shows *rel-post-state R* (*run f s*) (*run g t*)
⟨*proof*⟩

lemma *rel-post-state-runI'*:
assumes *succeeds f s* $\longleftrightarrow$ *succeeds g t*
assumes *succeeds f s* $\Longrightarrow$ *succeeds g t* $\Longrightarrow$ *rel-post-state R* (*run f s*) (*run g t*)
shows *rel-post-state R* (*run f s*) (*run g t*)
⟨*proof*⟩

lemma *rel-post-state-runD*:
assumes *rel-post-state R* (*run f s*) (*run g t*)
shows (*succeeds f s* $\longleftrightarrow$ *succeeds g t*) ∧
(*succeeds f s* $\longrightarrow$ *succeeds g t* $\longrightarrow$ *rel-set R* (*outcomes* (*run f s*)) (*outcomes* (*run g t*)))
⟨*proof*⟩

lemma *rel-post-state-run-iff*:
rel-post-state R (*run f s*) (*run g t*) $\longleftrightarrow$
((*succeeds f s* $\longleftrightarrow$ *succeeds g t*) ∧
(*succeeds f s* $\longrightarrow$ *succeeds g t* $\longrightarrow$ *rel-set R* (*outcomes* (*run f s*)) (*outcomes* (*run g t*))))
⟨*proof*⟩

lemma *rel-spec-monad-succeeds-iff*: *rel-spec-monad R Q f g* $\Longrightarrow$ *R s t* $\Longrightarrow$ *succeeds*

$f\ s \longleftrightarrow succeeds\ g\ t$
⟨*proof*⟩

lemma *outcomes-top-ps*[*simp*]: $outcomes\ \top = \{\}$ $outcomes\ (pure\text{-}post\text{-}state\ x) = \{x\}$
$outcomes\ \bot = \{\}$
⟨*proof*⟩

16.11.2 $\top$

lemma *outcomes-top*[*outcomes-spec-monad*]: $outcomes\ (run\ \top\ s) = \{\}$
⟨*proof*⟩

lemma *succeeds-top*[*simp*]: $succeeds\ \top\ s \longleftrightarrow False$
⟨*proof*⟩

lemma *reaches-top*[*simp*]: $reaches\ \top\ s\ r\ t \longleftrightarrow False$
⟨*proof*⟩

16.11.3 $\bot$

lemma *outcomes-bot*[*outcomes-spec-monad*]: $outcomes\ (run\ \bot\ s) = \{\}$
⟨*proof*⟩

lemma *succeeds-bot*[*simp*]: $succeeds\ \bot\ s \longleftrightarrow True$
⟨*proof*⟩

lemma *reaches-bot*[*simp*]: $reaches\ \bot\ s\ r\ t \longleftrightarrow False$
⟨*proof*⟩

16.11.4 *fail*

lemma *outcomes-fail*[*outcomes-spec-monad*]: $outcomes\ (run\ fail\ s) = \{\}$
⟨*proof*⟩

lemma *succeeds-fail-iff*[*iff*]: $\neg\ (succeeds\ fail\ f)$
⟨*proof*⟩

lemma *reaches-fail-iff*[*iff*]: $\neg\ reaches\ fail\ s\ r\ t$
⟨*proof*⟩

16.11.5 *yield*

lemma *succeeds-yield*[*simp*]: $succeeds\ (yield\ r)\ s$
⟨*proof*⟩

lemma *outcomes-yield* [*outcomes-spec-monad*]: $outcomes\ (run\ (yield\ r)\ s) = \{(r, s)\}$
⟨*proof*⟩

lemma *reaches-yield*[*simp*]: *reaches* (*yield v*) *s r t* ⟷ (*r* = *v* ∧ (*t* = *s*))
⟨*proof*⟩

16.11.6 *return*

lemma *outcomes-return* [*outcomes-spec-monad*]:
outcomes (*run* (*return v*) *s*) = {(*Result v*, *s*)}
⟨*proof*⟩

lemma *succeeds-return* [*iff*]: *succeeds* (*return v*) *s*
⟨*proof*⟩

lemma *reaches-return*[*simp*]: *reaches* (*return v*) *s r t* ⟷ (*r* = *Result v* ∧ (*t* = *s*))
⟨*proof*⟩

lemma *refines-yield-left*:
refines (*yield x*) *f s t R*
if *succeeds f t* ⟹ *reaches f t x′ s′* ∧ *R* (*x*, *s*) (*x′*, *s′*)
⟨*proof*⟩

16.11.7 *skip*

lemma *outcomes-skip* [*outcomes-spec-monad*]:
outcomes (*run* (*skip*) *s*) = {(*Result* (), *s*)}
⟨*proof*⟩

lemma *succeeds-skip*: *succeeds skip s*
⟨*proof*⟩

lemma *reaches-skip*: *reaches* (*return v*) *s r t* ⟷ (*r* = *Result* () ∧ (*t* = *s*))
⟨*proof*⟩

lemma *runs-to-skip*[*runs-to-vcg*]: *skip* · *s* ⦃ λ- *t*. *t* = *s* ⦄
⟨*proof*⟩

lemma *runs-to-partial-skip*[*runs-to-vcg*]: *skip* · *s* ?⦃ λ- *t*. *t* = *s* ⦄
⟨*proof*⟩

lemma *runs-to-skip-iff*[*runs-to-iff*]: *skip* · *s* ⦃*Q*⦄ ⟷ *Q* (*Result* ()) *s*
⟨*proof*⟩

16.11.8 *throw-exception-or-result*

lemma *outcomes-throw-exception-or-result* [*outcomes-spec-monad*]:
outcomes (*run* (*throw-exception-or-result x*) *s*) = {(*Exception x*, *s*)}
⟨*proof*⟩

lemma *succeeds-throw-exception-or-result* [*iff*]: *succeeds* (*throw-exception-or-result v*) *s*

⟨*proof*⟩

lemma *reaches-throw-exception-or-result*[*simp*]:
reaches (*throw-exception-or-result x*) *s r t* ⟷ (*r* = *Exception x* ∧ (*t* = *s*))
⟨*proof*⟩

16.11.9 *throw*

lemma *outcomes-throw* [*outcomes-spec-monad*]:
outcomes (*run* (*throw e*) *s*) = {(*Exn e*, *s*)}
⟨*proof*⟩

16.11.10 *get-state*

lemma *outcomes-get-state* [*outcomes-spec-monad*]:
outcomes (*run get-state s*) = {(*Result s*, *s*)}
⟨*proof*⟩

lemma *succeeds-get-state* [*iff*]: *succeeds get-state s*
⟨*proof*⟩

lemma *reaches-get-state*[*simp*]:
reaches get-state s r t ⟷ (*r* = *Result s* ∧ (*t* = *s*))
⟨*proof*⟩

16.11.11 *set-state*

lemma *outcomes-set-state* [*outcomes-spec-monad*]:
outcomes (*run* (*set-state t*) *s*) = {(*Result* (), *t*)}
⟨*proof*⟩

lemma *succeeds-set-state* [*iff*]: *succeeds* (*set-state t*) *s*
⟨*proof*⟩

lemma *reaches-set-state*[*simp*]: *reaches* (*set-state s′*) *s r t* ⟷ (*r* = *Result* () ∧ (*t* = *s′*))
⟨*proof*⟩

16.11.12 *select*

lemma *succeeds-holds*: *succeeds f s* ⟷ *holds-post-state* (λ*x*. *True*) (*run f s*)
⟨*proof*⟩

lemma *succeeds-select*[*simp*]: *succeeds* (*select S*) *s*
⟨*proof*⟩

lemma *select-outcomes*[*outcomes-spec-monad*]: *outcomes* (*run* (*select S*) *s*) = (λ*v*. (*Result v*, *s*)) ‘ *S*
⟨*proof*⟩

lemma *reaches-select* [*simp*]: *reaches* (*select S*) *s r t* ⟷ (*r* ∈ *Result* ' *S* ∧ *t* = *s*)
⟨*proof*⟩

16.11.13 *unknown*

lemma *succeeds-unknown*[*simp*]: *succeeds unknown s*
⟨*proof*⟩

lemma *unknown-outcomes*[*outcomes-spec-monad*]: *outcomes* (*run unknown s*) = (λ*v*. (*Result v*, *s*)) ' *UNIV*
⟨*proof*⟩

lemma *reaches-unknown* [*simp*]: *reaches unknown s r t* ⟷ (*r* ∈ *Result* ' *UNIV* ∧ *t* = *s*)
⟨*proof*⟩

16.11.14 *lift-state*

lemma *succeeds-lift-state-iff*: *succeeds* (*lift-state R f*) *s* ⟷ (∀ *s'*. *R s s'* ⟶ *succeeds f s'*)
⟨*proof*⟩

16.11.15 **const** *exec-concrete*

lemma *succeeds-exec-concrete-iff*: *succeeds* (*exec-concrete st f*) *s* ⟷ (∀ *t*. *s* = *st t* ⟶ *succeeds f t*)
⟨*proof*⟩

lemma *reaches-exec-concrete*:
assumes *succeeds*: *succeeds* (*exec-concrete st f*) *s*
shows *reaches* (*exec-concrete st f*) *s r s'* ⟷ (∃ *t t'*. *s* = *st t* ∧ *reaches f t r t'* ∧ *s'* = *st t'*)
⟨*proof*⟩

16.11.16 **const** *exec-abstract*

lemma *succeeds-exec-abstract-iff*[*simp*]: *succeeds* (*exec-abstract st f*) *s* ⟷ *succeeds f* (*st s*)
⟨*proof*⟩

lemma *reaches-exec-abstract*:
shows *reaches* (*exec-abstract st f*) *s r s'* ⟷ (∃ *t'*. *reaches f* (*st s*) *r t'* ∧ *t'* = *st s'*)
⟨*proof*⟩

16.11.17 *bind-handle*

lemma *succeeds-bind-handle*:
succeeds (*bind-handle f g h*) *s* ⟷
(*succeeds f s* ∧

$(\forall\, x\ s'.\ reaches\ f\ s\ x\ s' \longrightarrow$
$(case\ x\ of\ Exception\ e \Rightarrow succeeds\ (h\ e)\ s' \mid Result\ v \Rightarrow succeeds\ (g\ v)\ s')))$
⟨*proof*⟩

lemma *succeeds-bind-handle-res*:
succeeds (*bind-handle* (*f*::('*s*, '*a*) *res-monad*) *g h*) *s* ⟷
(*succeeds f s* ∧
(∀ *v s'*. *reaches f s* (*Result v*) *s'* ⟶
succeeds (*g v*) *s'*))
⟨*proof*⟩

lemma *outcomes-bind-handle-succeeds*: *succeeds* (*bind-handle f g h*) *s* ⟹
outcomes (*run* (*bind-handle f g h*) *s*) =
⋃ ((λ(*r*, *s'*). *case r of Exception e* => *outcomes* (*run* (*h e*) *s'*)
| *Result v* ⇒ *outcomes* (*run* (*g v*) *s'*))
‘ (*outcomes* (*run f s*)))
⟨*proof*⟩

lemma *outcomes-bind-handle-succeeds-res*: *succeeds* (*bind-handle* (*f*::('*a*, '*s*) *res-monad*) *g h*) *s* ⟹
outcomes (*run* (*bind-handle f g h*) *s*) =
⋃ ((λ(*r*, *s'*). *case r of Exception e* => {} | *Result v* ⇒ *outcomes* (*run* (*g v*) *s'*))
‘ (*outcomes* (*run f s*)))
⟨*proof*⟩

lemma *reaches-bind-handle*: *reaches* (*bind-handle f g h*) *s r t* ⟷
(*succeeds* (*bind-handle f g h*) *s* ∧
(∃ *r' s'*. *reaches f s r' s'* ∧
(*case r' of*
Exception e ⇒ *reaches* (*h e*) *s' r t*
| *Result v* ⇒ *reaches* (*g v*) *s' r t*)))
⟨*proof*⟩

lemma *reaches-bind-handle-res*: *reaches* ((*bind-handle* (*f*::('*a*, '*s*) *res-monad*) *g h*)) *s r t* ⟷
(*succeeds* (*bind-handle f g h*) *s* ∧
(∃ *v s'*. *reaches f s* (*Result v*) *s'* ∧ *reaches* (*g v*) *s' r t*))
⟨*proof*⟩

16.11.18 (⋙=)

lemma *succeeds-bind*:
succeeds (*bind f g*) *s* ⟷
(*succeeds f s* ∧ (∀ *v s'*. *reaches f s* (*Result v*) *s'* ⟶ *succeeds* (*g v*) *s'*))
⟨*proof*⟩

lemma *outcomes-bind-succeeds*: *succeeds* (*bind f g*) *s* ⟹ *outcomes* (*run* (*bind f g*) *s*) =

$\bigcup$ ((λ(r, s′). case r of Exception e => {(Exception e, s′)} | Result v ⇒ outcomes (run (g v) s′))
‘ (outcomes (run f s)))
⟨proof⟩

lemma *outcomes-bind-succeeds-res*: succeeds (bind (f::('a, 's) res-monad) g) s ⟹ outcomes (run (bind f g) s) =
$\bigcup$ ((λ(r, s′). case r of Exception e => {} | Result v ⇒ outcomes (run (g v) s′))
‘ (outcomes (run f s)))
⟨proof⟩

lemma *reaches-bind*: reaches (bind f g) s r t ⟷
(succeeds (bind f g) s ∧
(∃ r′ s′. reaches f s r′ s′ ∧
(case r′ of
Exception e ⇒ r = Exception e ∧ t = s′
| Result v ⇒ reaches (g v) s′ r t)))
⟨proof⟩

lemma *reaches-bind-res*: reaches ((bind f g)::('a, 's) res-monad) s r t ⟷
(succeeds (bind f g) s ∧
(∃ v s′. reaches f s (Result v) s′ ∧ reaches (g v) s′ r t))
⟨proof⟩

lemma *runs-toD-outcomes*: f · s ⦃ P ⦄ ⟹ (x, s) ∈ outcomes (run f s) ⟹ P x s
⟨proof⟩

lemma *run-bind-reaches-cong*:
assumes f: run f s = run f′ s
assumes g: ⋀v s′. succeeds f s ⟹ succeeds f′ s ⟹
reaches f′ s (Result v) s′ ⟹ run (g v) s′ = run (g′ v) s′
shows run (bind f g) s = run (bind f′ g′) s
⟨proof⟩

lemma *refines-bind-right′*:
succeeds g t ⟹ reaches g t (Result a) t′ ⟹
refines f (h a) s t′ R ⟹
refines f (g >>= h) s t R
⟨proof⟩

lemma *outcomes-empty-bind*:
assumes emp: outcomes (run f s) = {}
shows outcomes (run (f >>= g) s) = {}
⟨proof⟩

lemma *refines-bind-bind-exn*:
assumes f: refines f f′ s s′ Q
assumes ll: ⋀e e′ t t′.
Q (Exn e, t) (Exn e′, t′) ⟹

R (*Exn* e, t) (*Exn* e', t')
assumes *lr*: $\bigwedge e\ v'\ t\ t'$.
Q (*Exn* e, t) (*Result* v', t') $\Longrightarrow$
refines (*throw* e) ($g'\ v'$) $t\ t'\ R$
assumes *rl*: $\bigwedge v\ e'\ t\ t'$.
Q (*Result* v, t) (*Exn* e', t') $\Longrightarrow$
refines ($g\ v$) (*throw* e') $t\ t'\ R$
assumes *rr*: $\bigwedge v\ v'\ t\ t'$.
Q (*Result* v, t) (*Result* v', t') $\Longrightarrow$
refines ($g\ v$) ($g'\ v'$) $t\ t'\ R$
shows *refines* ($f \gg\!= g$) ($f' \gg\!= g'$) $s\ s'\ R$
⟨*proof*⟩

16.11.19 *assert*

lemma *outcomes-assert*[*outcomes-spec-monad*]:
outcomes (*run* (*assert* P) s) = (*if* P *then* {(*Result* (), s)} *else* {})
⟨*proof*⟩

lemma *succeeds-assert*[*simp*]: *succeeds* (*assert* P) $s \longleftrightarrow P$
⟨*proof*⟩

lemma *reaches-assert*[*simp*]: *reaches* (*assert* P) $s\ r\ t \longleftrightarrow (P \wedge r = \mathit{Result}\ () \wedge t = s)$
⟨*proof*⟩

16.11.20 *assume*

lemma *outcomes-assume*[*outcomes-spec-monad*]:
outcomes (*run* (*assume* P) s) = (*if* P *then* {(*Result* (), s)} *else* {})
⟨*proof*⟩

lemma *succeeds-assume*[*simp*]: *succeeds* (*assume* P) s
⟨*proof*⟩

lemma *reaches-assume*[*simp*]: *reaches* (*assume* P) $s\ r\ t \longleftrightarrow (P \wedge\ r = \mathit{Result}\ () \wedge t = s)$
⟨*proof*⟩

16.11.21 *assume-outcome*

lemma *outcomes-assume-outcome*[*outcomes-spec-monad*]:
outcomes (*run* (*assume-outcome* f) s) = $f\ s$
⟨*proof*⟩

lemma *succeeds-assume-outcome*[*simp*]:
succeeds (*assume-outcome* f) s
⟨*proof*⟩

lemma *reaches-assume-outcome*[*simp*]:

reaches (*assume-outcome f*) *s r t* ⟷ (*r*, *t*) ∈ *f s*
⟨*proof*⟩

16.11.22 *assume-result-and-state*

lemma *outcomes-assume-result-and-state*[*outcomes-spec-monad*]:
outcomes (*run* (*assume-result-and-state f*) *s*) = ((λ(*v*, *t*). (*Result v*, *t*)) ‘ *f s*)
⟨*proof*⟩

lemma *succeeds-assume-result-and-state*[*simp*]: *succeeds* (*assume-result-and-state f*) *s*
⟨*proof*⟩

lemma *Union-outcomes-split*:
(⋃ *x*∈*f s*.
(*outcomes* (*run* (*case x of* (*v*, *y*) ⇒ *g v y*) *s*))) = (⋃ (*v*, *y*)∈ *f s*. *outcomes* (*run* (*g v y*) *s*))
⟨*proof*⟩

lemma *reaches-assume-result-and-state* [*simp*]: *reaches* (*assume-result-and-state f*) *s r t* ⟷ (∃ *v*. *r* = *Result v* ∧ (*v*, *t*) ∈ *f s*)
⟨*proof*⟩

16.11.23 *gets*

lemma *succeeds-gets*[*simp*]: *succeeds* (*gets f*) *s*
⟨*proof*⟩

lemma *outcomes-gets*[*simp*]: *outcomes* (*run* (*gets f*) *s*) = {(*Result* (*f s*), *s*)}
⟨*proof*⟩

lemma *reaches-gets*[*simp*]: *reaches* (*gets f*) *s r t* ⟷ (*r* = *Result* (*f s*) ∧ *t* = *s*)
⟨*proof*⟩

16.11.24 *assert-result-and-state*

lemma *succeeds-assert-result-and-state*[*simp*]: *succeeds* (*assert-result-and-state f*) *s* ⟷ *f s* ≠ {}
⟨*proof*⟩

lemma *outcomes-assert-result-and-state*[*outcomes-spec-monad*]:
outcomes (*run* (*assert-result-and-state f*) *s*) = (*if f s* = {} *then* {} *else* ((λ(*v*, *t*). (*Result v*, *t*)) ‘ *f s*))
⟨*proof*⟩

lemma *reaches-assert-result-and-state* [*simp*]: *reaches* (*assert-result-and-state f*) *s r t* ⟷ (∃ *v*. *r* = *Result v* ∧ (*v*, *t*) ∈ *f s*)
⟨*proof*⟩

16.11.25 *assuming*

lemma *succeeds-assuming*[*simp*]: *succeeds* (*assuming g*) *s*
⟨*proof*⟩

lemma *outcomes-assuming*[*outcomes-spec-monad*]: *outcomes* (*run* (*assuming g*) *s*) = (*if g s then* {(*Result* (), *s*)} *else* {})
⟨*proof*⟩

lemma *reaches-assuming*[*simp*]: *reaches* (*assuming g*) *s r t* ⟷ (*g s* ∧ *r* = *Result* () ∧ *t* = *s*)
⟨*proof*⟩

16.11.26 *guard*

lemma *succeeds-guard*[*simp*]: *succeeds* (*guard g*) *s* ⟷ *g s*
⟨*proof*⟩

lemma *outcomes-guard*[*outcomes-spec-monad*]: *outcomes* (*run* (*guard g*) *s*) = (*if g s then* {(*Result* (), *s*)} *else* {})
⟨*proof*⟩

lemma *reaches-guard*[*simp*]: *reaches* (*guard g*) *s r t* ⟷ (*g s* ∧ *r* = *Result* () ∧ *t* = *s*)
⟨*proof*⟩

16.11.27 *assert-opt*

lemma *succeeds-assert-opt*[*simp*]: *succeeds* (*assert-opt x*) *s* ⟷ (∃ *v*. *x* = *Some v*)
⟨*proof*⟩

lemma *outcomes-assert-opt*[*simp*]:
outcomes (*run* (*assert-opt x*) *s*) = (*case x of Some v* ⇒ {(*Result v*, *s*)} | *None* ⇒ {})
⟨*proof*⟩

lemma *reaches-assert-opt*[*simp*]: *reaches* (*assert-opt x*) *s r t* ⟷ (∃ *v*. *x* = *Some v* ∧ *r* = *Result v* ∧ *t* = *s*)
⟨*proof*⟩

16.11.28 *gets-the*

lemma *succeeds-gets-the*[*simp*]: *succeeds* (*gets-the f*) *s* ⟷ (∃ *v*. *f s* = *Some v*)
⟨*proof*⟩

lemma *outcomes-gets-the*[*simp*]:
outcomes (*run* (*gets-the f*) *s*) = (*case f s of Some v* ⇒ {(*Result v*, *s*)} | *None* ⇒ {})
⟨*proof*⟩

lemma *reaches-gets-the*[*simp*]: *reaches* (*gets-the f*) *s r t* ⟷ (∃ *v*. *f s* = *Some v* ∧ *r* = *Result v* ∧ *t* = *s*)
⟨*proof*⟩

16.11.29 *modify*

lemma *outcomes-run-modify*[*outcomes-spec-monad*]: *outcomes* (*run* (*modify f*) *s*) = {(*Result* (), *f s*)}
⟨*proof*⟩

lemma *succeeds-modify*[*simp*]: *succeeds* (*modify f*) *s*
⟨*proof*⟩

lemma *reaches-modify*[*simp*]: *reaches* (*modify f*) *s r t* ⟷ (*r* = *Result* () ∧ *t* = *f s*)
⟨*proof*⟩

16.11.30 condition

lemma *outcomes-condition*:
outcomes (*run* (*condition c f g*) *s*) = (*if c s then outcomes* (*run f s*) *else outcomes* (*run g s*))
⟨*proof*⟩

lemma *succeeds-condition-iff*:
succeeds (*condition c f g*) *s* ⟷ *succeeds* (*if c s then f else g*) *s*
⟨*proof*⟩

lemma *reaches-condition-iff*:
reaches (*condition c f g*) *s r′ s′* ⟷ *reaches* (*if c s then f else g*) *s r′ s′*
⟨*proof*⟩

lemma *reaches-condition-True*[*simp*]:
c s ⟹ *reaches* (*condition c f g*) *s r′ s′* ⟷ *reaches f s r′ s′*
⟨*proof*⟩

lemma *reaches-condition-False*[*simp*]:
¬ *c s* ⟹ *reaches* (*condition c f g*) *s r′ s′* ⟷ *reaches g s r′ s′*
⟨*proof*⟩

lemma *succeeds-condition-True*[*simp*]:
c s ⟹ *succeeds* (*condition c f g*) *s* ⟷ *succeeds f s*
⟨*proof*⟩

lemma *succeeds-condition-False*[*simp*]:
¬(*c s*) ⟹ *succeeds* (*condition c f g*) *s* ⟷ *succeeds g s*
⟨*proof*⟩

16.11.31 *when*

lemma *succeeds-when*: *succeeds* (*when c f*) *s* ⟷ ¬*c* ∨ *succeeds f s*
⟨*proof*⟩

lemma *outcomes-when*[*outcomes-spec-monad*]:
outcomes (*run* (*when c f*) *s*) = (*if c then outcomes* (*run f s*) *else* {(*Result* (), *s*)})
⟨*proof*⟩

lemma *reaches-when-iff*:
reaches (*when c f*) *s r′ s′* ⟷ (*if c then reaches f s r′ s′ else* (*r′* = *Result* () ∧ *s′* = *s*))
⟨*proof*⟩

16.11.32 While

lemma *reaches-whileLoop-cond-false*:
reaches (*whileLoop C B r*) *s* (*Result r′*) *s′* ⟹ ¬ *C r′ s′*
⟨*proof*⟩

lemma *bind-post-state-cong*:
(⋀*r* . *r* ∈ *outcomes x* ⟹ *g r* = *g′ r*) ⟹ *bind-post-state x g* = *bind-post-state x g′*
⟨*proof*⟩

16.11.33 *map-value*

lemma *succeeds-map-value*[*simp*]: *succeeds* (*map-value f g*) *s* ⟷ *succeeds g s*
⟨*proof*⟩

lemma *outcomes-map-value*[*outcomes-spec-monad*]:
outcomes (*run* (*map-value f g*) *s*) = ((λ(*v*, *s*). (*f v*, *s*)) ‘ (*outcomes* (*run g s*)))
⟨*proof*⟩

lemma *reaches-map-value*: *reaches* (*map-value f g*) *s r t* ⟷ (∃ *r′*. *reaches g s r′ t* ∧ *r* = *f r′*)
⟨*proof*⟩

16.11.34 *liftE*

lemma *succeeds-liftE*[*simp*]: *succeeds* (*liftE f*) *s* ⟷ *succeeds f s*
⟨*proof*⟩

lemma *outcomes-liftE*[*outcomes-spec-monad*]:
outcomes (*run* (*liftE f*) *s*) =
((λ(*v*, *s*). (*map-exception-or-result* (λ*x*. *undefined*) *id v*, *s*)) ‘ (*outcomes* (*run f s*)))
⟨*proof*⟩

lemma *reaches-liftE*: *reaches* (*liftE f*) *s r t* ⟷ (∃ *r'*. *reaches f s* (*Result r'*) *t* ∧ *r* = *Result r'*)
⟨*proof*⟩

lemma *bind-cong1*:
fixes *f*::('*a*, '*s*) *res-monad*
shows ⟦ *f* = *f'*; ⋀*v s s'*. *reaches f s* (*Result v*) *s'* ⟹ *g v* = *g' v* ⟧ ⟹ *f* >>= *g* = *f'* >>= *g'*
⟨*proof*⟩

lemma *bind-liftE-cong1*:
fixes *f*::('*a*, '*s*) *res-monad*
shows ⟦ *f* = *f'*; ⋀*v s s'*. *reaches f s* (*Result v*) *s'* ⟹ *g v* = *g' v* ⟧ ⟹
(*liftE f*) >>= *g* = (*liftE f'*) >>= *g'*
⟨*proof*⟩

16.11.35 *try*

lemma *succeeds-try*[*simp*]: *succeeds* (*try f*) *s* ⟷ *succeeds f s*
⟨*proof*⟩

lemma *outcomes-try*[*outcomes-spec-monad*]:
outcomes (*run* (*try f*) *s*) = ((λ(*v*, *s*). (*unnest-exn v*, *s*)) ‘ (*outcomes* (*run f s*)))
⟨*proof*⟩

lemma *reaches-try*: *reaches* (*try f*) *s r t* ⟷ (∃ *r'*. *reaches f s r' t* ∧ *r* = *unnest-exn r'*)
⟨*proof*⟩

16.11.36 *finally*

lemma *succeeds-finally*[*simp*]: *succeeds* (*finally f*) *s* ⟷ *succeeds f s*
⟨*proof*⟩

lemma *outcomes-finally*[*outcomes-spec-monad*]:
outcomes (*run* (*finally f*) *s*) = ((λ(*v*, *s*). (*unite v*, *s*)) ‘ (*outcomes* (*run f s*)))
⟨*proof*⟩

lemma *reaches-finally*: *reaches* (*finally f*) *s r t* ⟷ (∃ *r'*. *reaches f s r' t* ∧ *r* = *unite r'*)
⟨*proof*⟩

16.11.37 (<*catch*>)

lemma *succeeds-catch*: *succeeds* (*f* <*catch*> *h*) *s* ⟷
(*succeeds f s* ∧
(∀ *x s'*. *reaches f s x s'* ⟶ (*case x of Exn e* ⇒ *succeeds* (*h e*) *s'* | *Result v* ⇒ *True*)))
⟨*proof*⟩

lemma *outcomes-catch-succeeds*: *succeeds* (*f* <*catch*> *h*) *s* ⟹
outcomes (*run* (*f* <*catch*> *h*) *s*) =
⋃ ((λ(*r*, *s*′). *case r of Exn e* => *outcomes* (*run* (*h e*) *s*′) | *Result v* ⇒ {(*Result v*, *s*′)})
‘ (*outcomes* (*run f s*)))
⟨*proof*⟩

lemma *reaches-catch*: *reaches* (*f* <*catch*> *h*) *s r t* ⟷
(*succeeds* (*f* <*catch*> *h*) *s* ∧
(∃ *r*′ *s*′. *reaches f s r*′ *s*′ ∧
(*case r*′ *of*
Exn e ⇒ *reaches* (*h e*) *s*′ *r t*
| *Result v* ⇒ *r* = *Result v* ∧ *t* = *s*′)))
⟨*proof*⟩

16.11.38 *check*

lemma *succeeds-check*[*simp*]: *succeeds* (*check e p*) *s*
⟨*proof*⟩

lemma *outcomes-check*[*outcomes-spec-monad*]: *outcomes* (*run* (*check e p*) *s*) =
(*if* (∃ *x*. *p s x*) *then* ((λ*x*. (*x*, *s*)) ‘ (*Result* ‘ {*x*. *p s x*})) *else* {(*Exn e*, *s*)})
⟨*proof*⟩

lemma *reaches-check*: *reaches* (*check e p*) *s r t* ⟷
(*t* = *s*) ∧
(*if* (∃ *x*. *p s x*) *then* (∃ *x*. *r* = *Result x* ∧ *p s x*) *else r* = *Exn e*)
⟨*proof*⟩

lemma *refines-bind-ok*:
succeeds g t ⟹ *reaches g t* (*Result a*) *t*′ ⟹
refines f (*h a*) *s t*′ *R* ⟹
refines f (*g* >>= *h*) *s t R*
⟨*proof*⟩

16.11.39 *ignoreE*

lemma *succeeds-ignoreE*[*simp*]:
succeeds (*ignoreE f*) *s* ⟷ (*succeeds f s*)
⟨*proof*⟩

lemma *outcomes-ignoreE-succeeds*: *succeeds f s* ⟹ *outcomes* (*run* (*ignoreE f*) *s*)
=
⋃ ((λ(*r*, *s*′). *case r of Exn e* => {} | *Result v* ⇒ {(*Result v*, *s*′)})
‘ (*outcomes* (*run f s*)))
⟨*proof*⟩

lemma *reaches-ignoreE*:
reaches (*ignoreE f*) *s r t* ⟷ (*succeeds f s*) ∧ (∃ *v*. *r* = *Result v* ∧ *reaches f s* (*Result v*) *t*)

⟨*proof*⟩

16.11.40 *bind-exception-or-result*

lemma *succeeds-bind-exception-or-result*:
succeeds (*bind-exception-or-result f g*) *s* ⟷
succeeds f s ∧ (∀ *v s′*. *reaches f s v s′* ⟶ *succeeds* (*g v*) *s′*)
⟨*proof*⟩

lemma *reaches-bind-exception-or-result*:
reaches (*bind-exception-or-result f g*) *s r t* ⟷
(*succeeds* (*bind-exception-or-result f g*) *s* ∧
(∃ *r′ s′*. *reaches f s r′ s′* ∧ *reaches* (*g r′*) *s′ r t*))
⟨*proof*⟩

lemma *refines-bind-exception-or-result-strong*:
assumes *f*: *refines f f′ s s′ Q*
assumes *g*: ⋀*r t r′ t′*. *Q* (*r*, *t*) (*r′*, *t′*) ⟹ *reaches f s r t* ⟹ *reaches f′ s′ r′ t′* ⟹ *refines* (*g r*) (*g′ r′*) *t t′ R*
shows *refines* (*bind-exception-or-result f g*) (*bind-exception-or-result f′ g′*) *s s′ R*
⟨*proof*⟩

16.11.41 *bind-finally*

lemma *succeeds-bind-finally*:
succeeds (*bind-finally f g*) *s* = (*succeeds f s* ∧ (∀ *v s′*. *reaches f s v s′* ⟶ *succeeds* (*g v*) *s′*))
⟨*proof*⟩

lemma *reaches-bind-finally*: *reaches* (*bind-finally f g*) *s r t* ⟷ (*succeeds* (*bind-finally f g*) *s* ∧ (∃ *r′ s′*. *reaches f s r′ s′* ∧ *reaches* (*g r′*) *s′ r t*))
⟨*proof*⟩

16.11.42 *run-bind*

lemma *succeeds-run-bind*:
succeeds (*run-bind f t g*) *s* ⟷ *succeeds f t* ∧ (∀ *r t′*. *reaches f t r t′* ⟶ *succeeds* (*g r t′*) *s*)
⟨*proof*⟩

lemma *reaches-run-bind*: *reaches* (*run-bind f t g*) *s r s′* ⟷
(*succeeds* (*run-bind f t g*) *s*) ∧
(∃ *r′ t′*. *reaches f t r′ t′* ∧ *reaches* (*g r′ t′*) *s r s′*)
⟨*proof*⟩

lemma *runs-to-partial-reaches*: *f* ⋅ *s* ?{ *reaches f s* }
⟨*proof*⟩

lemma *refines-strengthen-reaches*:

assumes *f-g*: *refines f g s t R*
assumes *reach*: $(\bigwedge x\ s'\ y\ t'.\ R\ (x,\ s')\ (y,\ t') \Longrightarrow \mathit{reaches}\ f\ s\ x\ s' \Longrightarrow \mathit{reaches}\ g\ t\ y\ t' \Longrightarrow Q\ (x,\ s')\ (y,\ t'))$
shows *refines f g s t Q*
⟨*proof*⟩

lemma *refines-bind-bind-strong′*:
assumes *f*: *refines f f′ s s′ Q*
assumes *Ex-Ex*: $(\bigwedge e\ e'\ t\ t'.$
$Q\ (\mathit{Exception}\ e,\ t)\ (\mathit{Exception}\ e',\ t') \Longrightarrow$
$e \neq \mathit{default} \Longrightarrow$
$e' \neq \mathit{default} \Longrightarrow$
$\mathit{reaches}\ f\ s\ (\mathit{Exception}\ e)\ t \Longrightarrow$
$\mathit{reaches}\ f'\ s'\ (\mathit{Exception}\ e')\ t' \Longrightarrow$
$R\ (\mathit{Exception}\ e,\ t)\ (\mathit{Exception}\ e',\ t'))$
assumes *Res-Ex*: $(\bigwedge e\ v'\ t\ t'.$
$Q\ (\mathit{Exception}\ e,\ t)\ (\mathit{Result}\ v',\ t') \Longrightarrow$
$e \neq \mathit{default} \Longrightarrow$
$\mathit{reaches}\ f\ s\ (\mathit{Exception}\ e)\ t \Longrightarrow$
$\mathit{reaches}\ f'\ s'\ (\mathit{Result}\ v')\ t' \Longrightarrow$
$\mathit{refines}\ (\mathit{yield}\ (\mathit{Exception}\ e))\ (g'\ v')\ t\ t'\ R)$
assumes *Ex-Ex*: $(\bigwedge v\ e'\ t\ t'.$
$Q\ (\mathit{Result}\ v,\ t)\ (\mathit{Exception}\ e',\ t') \Longrightarrow$
$e' \neq \mathit{default} \Longrightarrow$
$\mathit{reaches}\ f\ s\ (\mathit{Result}\ v)\ t \Longrightarrow$
$\mathit{reaches}\ f'\ s'\ (\mathit{Exception}\ e')\ t' \Longrightarrow$
$\mathit{refines}\ (g\ v)\ (\mathit{yield}\ (\mathit{Exception}\ e'))\ t\ t'\ R)$
assumes *Res-Res*: $(\bigwedge v\ v'\ t\ t'.$
$Q\ (\mathit{Result}\ v,\ t)\ (\mathit{Result}\ v',\ t') \Longrightarrow$
$\mathit{reaches}\ f\ s\ (\mathit{Result}\ v)\ t \Longrightarrow$
$\mathit{reaches}\ f'\ s'\ (\mathit{Result}\ v')\ t' \Longrightarrow$
$\mathit{refines}\ (g\ v)\ (g'\ v')\ t\ t'\ R)$
shows *refines* $(f \gg\!= g)\ (f' \gg\!= g')\ s\ s'\ R$
⟨*proof*⟩

lemma *rel-spec-monad-bind-strong*:
assumes *f-f′*: *rel-spec-monad S P f f′*
assumes *Ex-Ex*: $\bigwedge e\ e'\ s\ s'\ t\ t'.\ S\ s\ s' \Longrightarrow S\ t\ t' \Longrightarrow P\ (\mathit{Exception}\ e)\ (\mathit{Exception}\ e') \Longrightarrow e \neq \mathit{default} \Longrightarrow e' \neq \mathit{default} \Longrightarrow$
$\mathit{reaches}\ f\ s\ (\mathit{Exception}\ e)\ t \Longrightarrow \mathit{reaches}\ f'\ s'\ (\mathit{Exception}\ e')\ t' \Longrightarrow$
$Q\ (\mathit{Exception}\ e)\ (\mathit{Exception}\ e')$
assumes *Ex-Res*: $\bigwedge e\ v'\ s\ s'\ t\ t'.\ S\ s\ s' \Longrightarrow S\ t\ t' \Longrightarrow P\ (\mathit{Exception}\ e)\ (\mathit{Result}\ v') \Longrightarrow e \neq \mathit{default} \Longrightarrow$
$\mathit{reaches}\ f\ s\ (\mathit{Exception}\ e)\ t \Longrightarrow \mathit{reaches}\ f'\ s'\ (\mathit{Result}\ v')\ t' \Longrightarrow$
$\mathit{rel\text{-}spec\text{-}monad}\ S\ Q\ (\mathit{yield}\ (\mathit{Exception}\ e))\ (g'\ v')$
assumes *Res-Ex*: $\bigwedge v\ e'\ s\ s'\ t\ t'.\ S\ s\ s' \Longrightarrow S\ t\ t' \Longrightarrow P\ (\mathit{Result}\ v)\ (\mathit{Exception}\ e') \Longrightarrow e' \neq \mathit{default} \Longrightarrow$

reaches f s (Result v) t ⟹ reaches f′ s′ (Exception e′) t′ ⟹
rel-spec-monad S Q (g v) (yield (Exception e′))
assumes *Res-Res*: $\bigwedge v\ v'\ s\ s'\ t\ t'.\ S\ s\ s' \Longrightarrow S\ t\ t' \Longrightarrow P\ (Result\ v)\ (Result\ v') \Longrightarrow$
reaches f s (Result v) t ⟹ reaches f′ s′ (Result v′) t′ ⟹
rel-spec-monad S Q (g v) (g′ v′)
shows *rel-spec-monad S Q (f ⤜ g) (f′ ⤜ g′)*
⟨*proof*⟩

lemma *rel-spec-monad-bind-strong-exn*:
assumes *f-f′*: *rel-spec-monad S P f f′*
assumes *Ex-Ex*: $\bigwedge e\ e'\ s\ s'\ t\ t'.\ S\ s\ s' \Longrightarrow S\ t\ t' \Longrightarrow P\ (Exn\ e)\ (Exn\ e') \Longrightarrow$
reaches f s (Exn e) t ⟹ reaches f′ s′ (Exn e′) t′ ⟹
Q (Exn e) (Exn e′)
assumes *Ex-Res*: $\bigwedge e\ v'\ s\ s'\ t\ t'.\ S\ s\ s' \Longrightarrow S\ t\ t' \Longrightarrow P\ (Exn\ e)\ (Result\ v') \Longrightarrow$
reaches f s (Exn e) t ⟹ reaches f′ s′ (Result v′) t′ ⟹
rel-spec-monad S Q (throw e) (g′ v′)
assumes *Res-Ex*: $\bigwedge v\ e'\ s\ s'\ t\ t'.\ S\ s\ s' \Longrightarrow S\ t\ t' \Longrightarrow P\ (Result\ v)\ (Exn\ e') \Longrightarrow$
reaches f s (Result v) t ⟹ reaches f′ s′ (Exn e′) t′ ⟹
rel-spec-monad S Q (g v) (throw e′)
assumes *Res-Res*: $\bigwedge v\ v'\ s\ s'\ t\ t'.\ S\ s\ s' \Longrightarrow S\ t\ t' \Longrightarrow P\ (Result\ v)\ (Result\ v') \Longrightarrow$
reaches f s (Result v) t ⟹ reaches f′ s′ (Result v′) t′ ⟹
rel-spec-monad S Q (g v) (g′ v′)
shows *rel-spec-monad S Q (f ⤜ g) (f′ ⤜ g′)*
⟨*proof*⟩

lemma *rel-spec-monad-rel-xval-bind-strong*:
assumes *f-f′*: *rel-spec-monad S (rel-xval L P) f f′*
assumes *Res-Res*: $\bigwedge v\ v'\ s\ s'\ t\ t'.\ S\ s\ s' \Longrightarrow S\ t\ t' \Longrightarrow P\ v\ v' \Longrightarrow$
reaches f s (Result v) t ⟹ reaches f′ s′ (Result v′) t′ ⟹
rel-spec-monad S (rel-xval L R) (g v) (g′ v′)
shows *rel-spec-monad S (rel-xval L R) (f ⤜ g) (f′ ⤜ g′)*
⟨*proof*⟩

lemma *rel-spec-monad-bind-exception-or-result-strong*:
assumes *f-f′*: *rel-spec-monad S P f f′*
assumes *g-g′*: $\bigwedge r\ r'\ s\ s'\ t\ t'.\ S\ s\ s' \Longrightarrow S\ t\ t' \Longrightarrow P\ r\ r' \Longrightarrow$
reaches f s r t ⟹ reaches f′ s′ r′ t′ ⟹
rel-spec-monad S Q (g r) (g′ r′)
shows *rel-spec-monad S Q (bind-exception-or-result f g) (bind-exception-or-result f′ g′)*
⟨*proof*⟩

end

theory *Simp-Trace*
imports *AutoCorres-Utils*
begin

ATTENTION: to activate these methods use the following line:
setup ‹*Raw-Simplifier.set-trace-ops Simp-Trace.trace-ops*›

Provide a tactic wrapper to activate simplifier tracing and produce a statistic how many conditional rules were tried for how long. Also provides a shorthand for simp trace activation by adding *T* to the method name: *simpT simp-allT autoT*

⟨*ML*⟩

end

Chapter 17

Basic Stuff

theory *AutoCorres-Base*
imports
TypHeapLib
LemmaBucket-C
CTranslation
Synthesize
Cong-Tactic

Reaches
Mutual-CCPO-Recursion
Simp-Trace
begin

definition *THIN* :: *prop* ⇒ *prop* **where** *PROP THIN* (*PROP P*) ≡ *PROP P*

lemma *THIN-I*: *PROP P* ⟹ *PROP THIN* (*PROP P*)
⟨*proof*⟩

⟨*ML*⟩

named-theorems *corres-admissible* **and** *corres-top*
named-theorems *funp-intros* **and** *fun-of-rel-intros*

definition *fun-of-rel*:: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ ('*a* ⇒ '*b*) ⇒ *bool* **where**
fun-of-rel r f = (∀ *x y*. *r x y* ⟶ *y* = *f x*)

definition *funp*:: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ *bool* **where**
funp r = (∃ *f*. *fun-of-rel r f*)

lemma *funp-witness*: *fun-of-rel r f* ⟹ *funp r*
⟨*proof*⟩

lemma *funp-to-single-valuedp*: *funp r* ⟹ *single-valuedp r*

⟨*proof*⟩

lemma *fun-of-rel-xval*[*fun-of-rel-intros*]:
fun-of-rel L f-l ⟹ *fun-of-rel R f-r* ⟹ *fun-of-rel* (*rel-xval L R*) (*map-xval f-l f-r*)
⟨*proof*⟩

lemma *funp-rel-xval*[*funp-intros*, *corres-admissible*]:
assumes *L*: *funp L*
assumes *R*: *funp R*
shows *funp* (*rel-xval L R*)
⟨*proof*⟩

lemma *fun-of-rel-eq*[*fun-of-rel-intros*]: *fun-of-rel* (=) (λ*x*. *x*)
⟨*proof*⟩

lemma *fun-of-rel-bot*[*fun-of-rel-intros*]: *fun-of-rel* (λ- -. *False*) *f*
⟨*proof*⟩

lemma *funp-eq*[*funp-intros*, *corres-admissible*]: *funp* (=)
⟨*proof*⟩

lemma *admissible-mem*: *ccpo.admissible Inf* (≥) (λ*A*. *x* ∈ *A*)
⟨*proof*⟩

lemma *funp-rel-prod*[*funp-intros*, *corres-admissible*]:
assumes *L*: *funp L*
assumes *R*: *funp R*
shows *funp* (*rel-prod L R*)
⟨*proof*⟩

lemma *funp-rel-sum*[*funp-intros*, *corres-admissible*]:
assumes *L*: *funp L*
assumes *R*: *funp R*
shows *funp* (*rel-sum L R*)
⟨*proof*⟩

lemma *fun-of-rel-bottom*: *fun-of-rel* ((λ- -. *False*)) *f*
⟨*proof*⟩

lemma *funp-bottom* [*funp-intros*, *corres-admissible*]: *funp* (λ- -. *False*)
⟨*proof*⟩

lemma *fun-of-rel-prod*[*fun-of-rel-intros*]:
fun-of-rel L f-l ⟹ *fun-of-rel R f-r* ⟹ *fun-of-rel* (*rel-prod L R*) (*map-prod f-l f-r*)
⟨*proof*⟩

lemma *fun-of-rel-sum*[*fun-of-rel-intros*]:
fun-of-rel L f-l ⟹ *fun-of-rel R f-r* ⟹ *fun-of-rel* (*rel-sum L R*) (*sum-map f-l*

f-r)
⟨*proof*⟩

lemma *fun-of-relcompp*[*fun-of-rel-intros*]: *fun-of-rel F f* ⟹ *fun-of-rel G g* ⟹ *fun-of-rel* (*F OO G*) (*g o f*)
⟨*proof*⟩

lemma *funp-relcompp*[*funp-intros*, *corres-admissible*]: *funp F* ⟹ *funp G* ⟹ *funp* (*F OO G*)
⟨*proof*⟩

lemma *fun-of-rel-rel-map*[*fun-of-rel-intros*]: *fun-of-rel* (*rel-map f*) *f*
⟨*proof*⟩

lemma *funp-rel-map*[*funp-intros*, *corres-admissible*]: *funp* (*rel-map f*)
⟨*proof*⟩

lemma *ccpo-prod-gfp-gfp*:
class.ccpo
(*prod-lub Inf Inf* :: ((*'a*::*complete-lattice* ∗ *'b* :: *complete-lattice*) *set* ⇒ -))
(*rel-prod* (≥) (≥)) (*mk-less* (*rel-prod* (≥) (≥)))
⟨*proof*⟩

lemma *runs-to-partial-top*[*corres-top*]: ⊤ ⋅ *s* ?⦃ *Q* ⦄
⟨*proof*⟩

lemma *refines-top*[*corres-top*]: *refines C* ⊤ *s t R*
⟨*proof*⟩

lemma *pred-andE*[*elim*!]: ⟦ (*A and B*) *x*; ⟦ *A x*; *B x* ⟧ ⟹ *R* ⟧ ⟹ *R*
⟨*proof*⟩

lemma *pred-andI*[*intro*!]: ⟦ *A x*; *B x* ⟧ ⟹ (*A and B*) *x*
⟨*proof*⟩

definition *measure'* :: (*'a* ⇒ *'b*::*wellorder*) => (*'a* × *'a*) *set*
where *measure'* = (λ*f*. {(*a*, *b*). *f a* < *f b*})

lemma *in-measure'*[*simp*, *code-unfold*]:
((*x*,*y*) : *measure' f*) = (*f x* < *f y*)
⟨*proof*⟩

lemma *wf-measure'* [*iff*]: *wf* (*measure' f*)
⟨*proof*⟩

lemma *wf-wellorder-measure*: *wf* {(*a*, *b*). (*M a* :: *'a* :: *wellorder*) < *M b*}
⟨*proof*⟩

lemma *wf-custom-measure*:
⟦ ⋀*a b*. (*a*, *b*) ∈ *R* ⟹ *f a* < (*f* :: *'a* ⇒ *nat*) *b* ⟧ ⟹ *wf R*
⟨*proof*⟩

lemma *rel-fun-conversep'*: *rel-fun R Q f g* ⟷ *rel-fun* R^{-1-1} Q^{-1-1} *g f*
⟨*proof*⟩

end

theory *SimplBucket*
imports *AutoCorres-Base*
begin

lemma *Normal-resultE*:
⟦ Γ ⊢ ⟨*c*, *s*⟩ ⇒ *Normal t'*; ⋀*t*. ⟦ Γ ⊢ ⟨*c*, *Normal t*⟩ ⇒ *Normal t'*; *s* = *Normal t*⟧
⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

lemma *exec-While-final-cond'*:
⟦ Γ ⊢ ⟨*b*, *s*⟩ ⇒ *s'*; *b* = *While C B*; *s* = *Normal v*; *s'* = *Normal x* ⟧ ⟹ *x* ∉ *C*
⟨*proof*⟩

lemma *exec-While-final-cond*:
⟦ Γ ⊢ ⟨*While C B*, *s*⟩ ⇒ *Normal s'*⟧ ⟹ *s'* ∉ *C*
⟨*proof*⟩

lemma *exec-While-final-inv'*:
⟦ Γ ⊢ ⟨*b*, *s*⟩ ⇒ *s'*; *b* = *While C B*; *s* = *Normal v*; *s'* = *Normal x*; *I v*; ⋀*s s'*. ⟦ *I s*; Γ ⊢ ⟨*B*, *Normal s*⟩ ⇒ *Normal s'* ⟧ ⟹ *I s'* ⟧ ⟹ *I x*
⟨*proof*⟩

lemma *exec-While-final-inv*:
⟦ Γ ⊢ ⟨*While C B*, *Normal s*⟩ ⇒ *Normal s'*; *I s*; ⋀*s s'*. ⟦ *I s*; Γ ⊢ ⟨*B*, *Normal s*⟩ ⇒ *Normal s'* ⟧ ⟹ *I s'* ⟧ ⟹ *I s'*
⟨*proof*⟩

primrec
exceptions-thrown :: (*'a*, *'p*, *'e*) *com* ⇒ *bool*
where
exceptions-thrown Skip = *False*
| *exceptions-thrown* (*Seq a b*) = (*exceptions-thrown a* ∨ *exceptions-thrown b*)
| *exceptions-thrown* (*Basic a*) = *False*

| *exceptions-thrown* (*Language.Spec a*) = *False*
| *exceptions-thrown* (*Cond a b c*) = (*exceptions-thrown b* ∨ *exceptions-thrown c*)
| *exceptions-thrown* (*Catch a b*) = (*exceptions-thrown a* ∧ *exceptions-thrown b*)
| *exceptions-thrown* (*While a b*) = *exceptions-thrown b*
| *exceptions-thrown Throw* = *True*
| *exceptions-thrown* (*Call p*) = *True*
| *exceptions-thrown* (*Guard f g a*) = *exceptions-thrown a*
| *exceptions-thrown* (*DynCom a*) = (∃ *s*. *exceptions-thrown* (*a s*))

primrec
exceptions-unresolved :: ('*a*, '*p*, '*e*) *com list* ⇒ *bool*
where
exceptions-unresolved [] = *True*
| *exceptions-unresolved* (*x*#*xs*) = (*exceptions-thrown x* ∧ *exceptions-unresolved xs*)

lemma *exceptions-thrown-not-abrupt*:
⟦ Γ ⊢ ⟨*p*, *s*⟩ ⇒ *s*'; ¬ *exceptions-thrown p*; ¬ *isAbr s* ⟧ ⟹ ¬ *isAbr s*'
⟨*proof*⟩

end

theory *CCorresE*
imports
SimplBucket
begin

definition
ccorresE :: ('*t* ⇒ '*s*) ⇒ *bool* ⇒ '*ee set* ⇒ ('*p* ⇒ ('*t*, '*p*, '*ee*) *com option*)
⇒ ('*s* ⇒ *bool*) ⇒ ('*t set*)
⇒ (*unit*, *unit*, '*s*) *exn-monad* ⇒ ('*t*, '*p*, '*ee*) *com* ⇒ *bool*
where
ccorresE st check-term AF Γ *G G*' ≡
λ*m c*. ∀ *s*. *G* (*st s*) ∧ (*s* ∈ *G*') ∧ *succeeds m* (*st s*) ⟶
((∀ *t*. Γ ⊢ ⟨*c*, *Normal s*⟩ ⇒ *t* ⟶
(*case t of*
Normal s' ⇒ *reaches m* (*st s*) (*Result* ()) (*st s*')
| *Abrupt s*' ⇒ *reaches m* (*st s*) (*Exn* ()) (*st s*')
| *Fault e* ⇒ *e* ∈ *AF*
| - ⇒ *False*))
∧ (*check-term* ⟶ Γ ⊢ *c* ↓ *Normal s*))

lemma *ccorresE-cong*:
⟦ ⋀*s*. *P s* = *P′ s*;
⋀*s*. (*s* ∈ *Q*) = (*s* ∈ *Q′*);
⋀*s*. *P′ s* ⟹ *run f s* = *run f′ s*;
⋀*s x*. *s* ∈ *Q′* ⟹ Γ⊢ ⟨*g*, *Normal s*⟩ ⇒ *x* = Γ⊢ ⟨*g′*, *Normal s*⟩ ⇒ *x*
⟧ ⟹
ccorresE st ct AF Γ *P Q f g* = *ccorresE st ct AF* Γ *P′ Q′ f′ g*
⟨*proof*⟩

lemma *ccorresE-guard-imp*:
⟦ *ccorresE st ct AF* Γ *Q Q′ A B*; ⋀*s*. *P s* ⟹ *Q s*; ⋀*t*. *t* ∈ *P′* ⟹ *t* ∈ *Q′* ⟧ ⟹
ccorresE st ct AF Γ *P P′ A B*
⟨*proof*⟩

lemma *ccorresE-guard-imp-stronger*:
⟦ *ccorresE st ct AF* Γ *Q Q′ A B*;
⋀*s*. ⟦ *P* (*st s*); *s* ∈ *P′* ⟧ ⟹ *Q* (*st s*);
⋀*s*. ⟦ *P* (*st s*); *s* ∈ *P′* ⟧ ⟹ *s* ∈ *Q′* ⟧ ⟹
ccorresE st ct AF Γ *P P′ A B*
⟨*proof*⟩

lemma *ccorresE-assume-pre*:
⟦ ⋀*s*. ⟦ *G* (*st s*); *s* ∈ *G′* ⟧ ⟹
ccorresE st ct AF Γ (*G and* (λ*s′*. *s′* = *st s*)) (*G′* ∩ {*t′*. *t′* = *s*}) *A B* ⟧ ⟹
ccorresE st ct AF Γ *G G′ A B*
⟨*proof*⟩

lemma *ccorresE-Seq*:
⟦ *ccorresE st ct AF* Γ ⊤ *UNIV L L′*;
ccorresE st ct AF Γ ⊤ *UNIV R R′* ⟧ ⟹
ccorresE st ct AF Γ ⊤ *UNIV* (*do* {- ← *L*; *R* }) (*L′* ;; *R′*)
⟨*proof*⟩

lemma *ccorresE-Cond*:
⟦ *ccorresE st ct AF* Γ ⊤ *C A L′*;
ccorresE st ct AF Γ ⊤ (*UNIV* − *C*) *A R′* ⟧ ⟹
ccorresE st ct AF Γ ⊤ *UNIV A* (*Cond C L′ R′*)
⟨*proof*⟩

lemma *ccorresE-Cond-match*:
⟦ *ccorresE st ct AF* Γ *C C′ L L′*;
ccorresE st ct AF Γ (*not C*) (*UNIV* − *C′*) *R R′*;
⋀*s*. *C* (*st s*) = (*s* ∈ *C′*) ⟧ ⟹
ccorresE st ct AF Γ ⊤ *UNIV* (*condition C L R*) (*Cond C′ L′ R′*)
⟨*proof*⟩

lemma *ccorresE-Guard*:
⟦ *ccorresE st ct AF* Γ ⊤ *G X Y* ⟧ ⟹ *ccorresE st ct AF* Γ ⊤ *G X* (*Guard F G*

Y)
⟨*proof*⟩

lemma *ccorresE-Catch*:
⟦*ccorresE st ct AF* Γ ⊤ *UNIV A A′*; *ccorresE st ct AF* Γ ⊤ *UNIV B B′*⟧ ⟹
ccorresE st ct AF Γ ⊤ *UNIV* (*A* <*catch*> (λ-. *B*)) (*TRY A′ CATCH B′ END*)
⟨*proof*⟩

lemma *ccorresE-Call*:
⟦ Γ *X′* = *Some Z′*; *ccorresE st ct AF* Γ ⊤ *UNIV Z Z′* ⟧ ⟹
ccorresE st ct AF Γ ⊤ *UNIV Z* (*Call X′*)
⟨*proof*⟩

lemma *ccorresE-exec-Normal*:
⟦ *ccorresE st ct AF* Γ *G G′ B B′*; Γ⊢ ⟨*B′*, *Normal s*⟩ ⇒ *Normal t*; *s* ∈ *G′*; *G* (*st s*); *succeeds B* (*st s*) ⟧
⟹ *reaches B* (*st s*) (*Result* ()) (*st t*)
⟨*proof*⟩

lemma *ccorresE-exec-Abrupt*:
⟦ *ccorresE st ct AF* Γ *G G′ B B′*; Γ⊢ ⟨*B′*, *Normal s*⟩ ⇒ *Abrupt t*; *s* ∈ *G′*; *G* (*st s*); *succeeds B* (*st s*) ⟧
⟹ *reaches B* (*st s*) (*Exn* ()) (*st t*)
⟨*proof*⟩

lemma *ccorresE-exec-Fault*:
⟦ *ccorresE st ct AF* Γ *G G′ B B′*; Γ⊢ ⟨*B′*, *Normal s*⟩ ⇒ *Fault f*; *f* ∉ *AF*; *s* ∈ *G′*; *G* (*st s*); *succeeds B* (*st s*) ⟧ ⟹ *P*
⟨*proof*⟩

lemma *ccorresE-exec-Stuck*:
⟦ *ccorresE st ct AF* Γ *G G′ B B′*; Γ⊢ ⟨*B′*, *Normal s*⟩ ⇒ *Stuck*; *s* ∈ *G′*; *G* (*st s*); *succeeds B* (*st s*) ⟧ ⟹ *P*
⟨*proof*⟩

lemma *ccorresE-exec-cases* [*consumes 5*]:
⟦ *ccorresE st ct AF* Γ *G G′ B B′*; Γ⊢ ⟨*B′*, *Normal s*⟩ ⇒ *s′*; *s* ∈ *G′*; *G* (*st s*); *succeeds B* (*st s*);
⋀*t′*. ⟦ *s′* = *Normal t′*; *reaches B* (*st s*) (*Result* ()) (*st t′*)⟧ ⟹ *R*;
⋀*t′*. ⟦ *s′* = *Abrupt t′*; *reaches B* (*st s*) (*Exn* ()) (*st t′*) ⟧ ⟹ *R*;
⋀*f*. ⟦ *s′* = *Fault f*; *f* ∈ *AF* ⟧ ⟹ *R*
⟧ ⟹ *R*
⟨*proof*⟩

lemma *ccorresE-terminates*:
⟦ *ccorresE st ct AF* Γ ⊤ *UNIV B B′*; *succeeds B* (*st s*); *ct* ⟧ ⟹ Γ ⊢ *B′* ↓ *Normal s*
⟨*proof*⟩

lemma *exec-While-final-inv'*:
assumes *exec*: $\Gamma\vdash \langle b, x\rangle \Rightarrow s'$
shows
$[\![b = \mathit{While}\ C\ B;\ x = \mathit{Normal}\ s;$
$\bigwedge s.\ [\![s \notin C]\!] \Longrightarrow I\ s\ (\mathit{Normal}\ s);$
$\bigwedge t\ t'.\ [\![t \in C; \Gamma\vdash \langle B, \mathit{Normal}\ t\rangle \Rightarrow \mathit{Normal}\ t';\ I\ t'\ s']\!] \Longrightarrow I\ t\ s';$
$\bigwedge t\ t'.\ [\![t \in C; \Gamma\vdash \langle B, \mathit{Normal}\ t\rangle \Rightarrow \mathit{Abrupt}\ t']\!] \Longrightarrow I\ t\ (\mathit{Abrupt}\ t');$
$\bigwedge t.\ [\![t \in C; \Gamma\vdash \langle B, \mathit{Normal}\ t\rangle \Rightarrow \mathit{Stuck}]\!] \Longrightarrow I\ t\ \mathit{Stuck};$
$\bigwedge t\ f.\ [\![t \in C; \Gamma\vdash \langle B, \mathit{Normal}\ t\rangle \Rightarrow \mathit{Fault}\ f]\!] \Longrightarrow I\ t\ (\mathit{Fault}\ f)]\!]$
$\Longrightarrow I\ s\ s'$
⟨*proof*⟩

lemma *exec-While-final-inv*:
$[\![\Gamma\vdash \langle \mathit{While}\ C\ B, \mathit{Normal}\ s\rangle \Rightarrow s';$
$\bigwedge s.\ [\![s \notin C]\!] \Longrightarrow I\ s\ (\mathit{Normal}\ s);$
$\bigwedge t\ t'.\ [\![t \in C; \Gamma\vdash \langle B, \mathit{Normal}\ t\rangle \Rightarrow \mathit{Normal}\ t';\ I\ t'\ s']\!] \Longrightarrow I\ t\ s';$
$\bigwedge t\ t'.\ [\![t \in C; \Gamma\vdash \langle B, \mathit{Normal}\ t\rangle \Rightarrow \mathit{Abrupt}\ t']\!] \Longrightarrow I\ t\ (\mathit{Abrupt}\ t');$
$\bigwedge t.\ [\![t \in C; \Gamma\vdash \langle B, \mathit{Normal}\ t\rangle \Rightarrow \mathit{Stuck}]\!] \Longrightarrow I\ t\ \mathit{Stuck};$
$\bigwedge t\ f.\ [\![t \in C; \Gamma\vdash \langle B, \mathit{Normal}\ t\rangle \Rightarrow \mathit{Fault}\ f]\!] \Longrightarrow I\ t\ (\mathit{Fault}\ f)]\!]$
$\Longrightarrow I\ s\ s'$
⟨*proof*⟩

lemma *ccorresE-termination'*:
assumes *no-fail*: *succeeds* (*whileLoop CC BB r*) *s*
and *s-match*: $s = \mathit{st}\ s' \wedge \mathit{CC} = (\lambda\text{-}.\ C) \wedge \mathit{BB} = (\lambda\text{-}.\ B)$
and *corres*: *ccorresE st ct AF* Γ ⊤ *UNIV B B'*
and *cond-match*: $\bigwedge s.\ C\ (\mathit{st}\ s) = (s \in C')$
and *ct*: *ct*
shows $\Gamma\vdash \mathit{While}\ C'\ B' \downarrow \mathit{Normal}\ s'$
⟨*proof*⟩

lemma *ccorresE-termination*:
assumes *no-fail*: *succeeds* (*whileLoop* (λ-. *C*) (λ-. *B*) *r*) *s*
and *s-match*: $s = \mathit{st}\ s'$
and *corres*: *ccorresE st ct AF* Γ ⊤ *UNIV B B'*
and *cond-match*: $\bigwedge s.\ C\ (\mathit{st}\ s) = (s \in C')$
and *ct*: *ct*
shows $\Gamma\vdash \mathit{While}\ C'\ B' \downarrow \mathit{Normal}\ s'$
⟨*proof*⟩

lemma *ccorresE-While*:
assumes *body-refines*: *ccorresE st ct AF* Γ ⊤ *UNIV B B'*
and *cond-match*: $\bigwedge s.\ C\ (\mathit{st}\ s) = (s \in C')$
shows *ccorresE st ct AF* Γ *G G'* (*whileLoop* (λ-. *C*) (λ-. *B*) ()) (*While C' B'*)
⟨*proof*⟩

lemma *ccorresE-get*:
$(\bigwedge s.\ \mathit{ccorresE}\ \mathit{st}\ \mathit{ct}\ \mathit{AF}\ \Gamma\ (P\ \mathit{and}\ (\lambda s'.\ s' = s))\ Q\ (L\ s)\ R) \Longrightarrow \mathit{ccorresE}\ \mathit{st}\ \mathit{ct}\ \mathit{AF}$

$\Gamma\ P\ Q\ ((get\text{-}state) >>= L)\ R$
⟨*proof*⟩

lemma *ccorresE-fail*:
ccorresE st ct AF Γ P Q fail R
⟨*proof*⟩

lemma *ccorresE-DynCom*:
$[\![\bigwedge t.\ [\![t \in P']\!] \Longrightarrow ccorresE\ st\ ct\ AF\ \Gamma\ P\ (P' \cap \{t'.\ t' = t\})\ A\ (B\ t)]\!] \Longrightarrow ccorresE\ st\ ct\ AF\ \Gamma\ P\ P'\ A\ (DynCom\ B)$
⟨*proof*⟩

lemma *ccorresE-Catch-nothrow*:
$[\![ccorresE\ st\ ct\ AF\ \Gamma\ \top\ UNIV\ A\ A'; \neg\ exceptions\text{-}thrown\ A']\!] \Longrightarrow$
ccorresE st ct AF Γ ⊤ UNIV A (TRY A′ CATCH B′ END)
⟨*proof*⟩

context *stack-heap-state*
begin

definition *with-fresh-stack-ptr* :: $nat \Rightarrow ('s \Rightarrow 'a\ list\ set) \Rightarrow ('a::mem\text{-}type\ ptr \Rightarrow ('e::default, 'v, 's)\ spec\text{-}monad) \Rightarrow ('e::default, 'v, 's)\ spec\text{-}monad$
where
with-fresh-stack-ptr n I c ≡
do {
$p \leftarrow assume\text{-}result\text{-}and\text{-}state\ (\lambda s.\ \{(p,\ t).\ \exists d\ vs\ bs.\ (p,\ d) \in stack\text{-}allocs\ n\ \mathcal{S}\ TYPE('a::mem\text{-}type)\ (htd\ s) \wedge$
$vs \in I\ s \wedge length\ vs = n \wedge length\ bs = n * size\text{-}of\ TYPE('a) \wedge$
$t = hmem\text{-}upd\ (fold\ (\lambda i.\ heap\text{-}update\text{-}padding\ (p +_p int\ i)\ (vs!i)\ (take\ (size\text{-}of\ TYPE('a))\ (drop\ (i * size\text{-}of\ TYPE('a))\ bs)))\ [0..<n])\ (htd\text{-}upd\ (\lambda\text{-}.\ d)\ s)\})$;
on-exit (*c p*)
$(\{(s,t).\ \exists bs.\ length\ bs = n * size\text{-}of\ TYPE('a) \wedge t = hmem\text{-}upd\ (heap\text{-}update\text{-}list\ (ptr\text{-}val\ p)\ bs)\ (htd\text{-}upd\ (stack\text{-}releases\ n\ p)\ s)\})$
}

lemma *monotone-with-fresh-stack-ptr-le*[*partial-function-mono*]:
assumes [*partial-function-mono*]: $\bigwedge p.\ monotone\ R\ (\leq)\ (\lambda f.\ c\ f\ p)$
shows *monotone R* (≤) $(\lambda f.\ with\text{-}fresh\text{-}stack\text{-}ptr\ n\ I\ (c\ f))$
⟨*proof*⟩

lemma *monotone-with-fresh-stack-ptr-ge*[*partial-function-mono*]:
assumes [*partial-function-mono*]: $\bigwedge p.\ monotone\ R\ (\geq)\ (\lambda f.\ c\ f\ p)$
shows *monotone R* (≥) $(\lambda f.\ with\text{-}fresh\text{-}stack\text{-}ptr\ n\ I\ (c\ f))$
⟨*proof*⟩

$\langle ML \rangle$

end

end

Chapter 18

L1 phase

theory *L1Defs*
imports *CCorresE*
begin

type-synonym *'s L1-monad* = (*unit*, *unit*, *'s*) *exn-monad*
definition *L1-seq* (*A* :: *'s L1-monad*) (*B* :: *'s L1-monad*) ≡ (*A* >>= (λ-. *B*)) :: *'s L1-monad*
definition *L1-skip* ≡ *return* () :: *'s L1-monad*
definition *L1-modify m* ≡ (*modify m*) :: *'s L1-monad*
definition *L1-condition c* (*A* :: *'s L1-monad*) (*B* :: *'s L1-monad*) ≡ *condition c A B*
definition *L1-catch* (*A* :: *'s L1-monad*) (*B* :: *'s L1-monad*) ≡ (*A* <*catch*> (λ-. *B*))
definition *L1-while c* (*A* :: *'s L1-monad*) ≡ (*whileLoop* (λ-. *c*) (λ-. *A*) ())
definition *L1-throw* ≡ *throw* () :: *'s L1-monad*
definition *L1-spec r* ≡ *state-select r* :: *'s L1-monad*
definition *L1-assume f* ≡ *assume-result-and-state f* :: *'s L1-monad*
definition *L1-guard c* ≡ *guard c* :: *'s L1-monad*
definition *L1-init v* ≡ (*do* { *x* ← *select UNIV*; *modify* (*v* (λ-. *x*)) }) :: *'s L1-monad*
definition *L1-call scope-setup* (*dest-fn* :: *'s L1-monad*) *scope-teardown result-exn return-xf* ≡
  do {
    s ← *get-state*;
    ((*do* { *modify scope-setup*;
        dest-fn })
      <*catch*> (λ-. *L1-seq* (*modify* (λ*t*. *result-exn* (*scope-teardown s t*) *t*)) *L1-throw*));
    t ← *get-state*;
    modify (*scope-teardown s*);
    modify (*return-xf t*)
  }

definition *L1-fail* ≡ *fail* :: *'s L1-monad*
definition *L1-set-to-pred S* ≡ λ*s*. *s* ∈ *S*
definition *L1-rel-to-fun R* = (λ*s*. *Pair* () ‘ *Image R* {*s*})

lemma *L1-rel-to-fun-alt*: *L1-rel-to-fun R* = (λ*s*. *Pair* () ‘ {*s'*. (*s*, *s'*) ∈ *R*})
⟨*proof*⟩

lemmas *L1-defs* = *L1-seq-def L1-skip-def L1-modify-def L1-condition-def*
L1-catch-def L1-while-def L1-throw-def L1-spec-def L1-assume-def L1-guard-def
L1-fail-def L1-init-def L1-set-to-pred-def L1-rel-to-fun-def

lemmas *L1-defs'* =
L1-seq-def L1-skip-def L1-modify-def L1-condition-def L1-catch-def
L1-while-def L1-throw-def L1-spec-def L1-assume-def
L1-guard-def
L1-fail-def L1-init-def L1-set-to-pred-def L1-rel-to-fun-def

declare *L1-set-to-pred-def* [*simp*]
declare *L1-rel-to-fun-def* [*simp*]

definition *L1-guarded*:: (*'s* ⇒ *bool*) ⇒ *'s L1-monad* ⇒ *'s L1-monad*
where
L1-guarded g f = *L1-seq* (*L1-guard g*) *f*

locale *L1-functions* =
fixes P:: *unit ptr* ⇒ *'s L1-monad*
begin
definition *L1-dyn-call g scope-setup* (*dest* :: *'s* ⇒ *unit ptr*) *scope-teardown result-exn return-xf* ≡
L1-guarded g (*gets dest* >>= (λ*p*. *L1-call scope-setup* (P *p*) *scope-teardown result-exn return-xf*))
end

definition
L1corres :: *bool* ⇒ (*'p* ⇒ (*'s*, *'p*, *strictc-errortype*) *com option*)
⇒ *'s L1-monad* ⇒ (*'s*, *'p*, *strictc-errortype*) *com* ⇒ *bool*
where
L1corres check-term Γ ≡
λ*A C*. ∀ *s*. *succeeds A s* ⟶
((∀ *t*. Γ ⊢ ⟨*C*, *Normal s*⟩ ⇒ *t* ⟶
(*case t of*
Normal s' ⇒ *reaches A s* (*Result* ()) *s'*
| *Abrupt s'* ⇒ *reaches A s* (*Exn* ()) *s'*
| *Fault e* ⇒ *e* ∈ {*AssumeError*, *StackOverflow*}
| - ⇒ *False*))

$\wedge$ (*check-term* $\longrightarrow \Gamma \vdash C \downarrow$ *Normal s*))

definition
L1corres' :: *bool* $\Rightarrow$ (*'p* $\Rightarrow$ (*'s*, *'p*, *strictc-errortype*) *com option*) $\Rightarrow$ (*'s* $\Rightarrow$ *bool*)
$\Rightarrow$ *'s L1-monad* $\Rightarrow$ (*'s*, *'p*, *strictc-errortype*) *com* $\Rightarrow$ *bool*
where
L1corres' check-term Γ *P* $\equiv$
$\lambda A\ C.\ \forall s.\ (P\ s) \wedge$ *succeeds A s* $\longrightarrow$
((*∀ t.* $\Gamma \vdash \langle C,$ *Normal s*$\rangle \Rightarrow t \longrightarrow$
(*case t of*
Normal s' $\Rightarrow$ *reaches A s* (*Result* ()) *s'*
| *Abrupt s'* $\Rightarrow$ *reaches A s* (*Exn* ()) *s'*
| *Fault e* $\Rightarrow$ *e* $\in$ {*AssumeError*, *StackOverflow*}
| - $\Rightarrow$ *False*))
$\wedge$ (*check-term* $\longrightarrow \Gamma \vdash C \downarrow$ *Normal s*))

lemma *L1corres-alt-def*: *L1corres ct* Γ = *ccorresE* ($\lambda x.\ x$) *ct* {*AssumeError*, *StackOverflow*} Γ $\top$ *UNIV*
⟨*proof*⟩

lemma *L1corres'-alt-def*: *L1corres' ct* Γ *P* = *ccorresE* ($\lambda x.\ x$) *ct* {*AssumeError*, *StackOverflow*} Γ *P UNIV*
⟨*proof*⟩

lemma *admissible-nondet-ord-L1corres* [*corres-admissible*]:
ccpo.admissible Inf ($\geq$) ($\lambda A.$ *L1corres ct* Γ *A C*)
⟨*proof*⟩

lemma *L1corres-top* [*corres-top*]: *L1corres ct P* $\top$ *C*
⟨*proof*⟩

lemma *L1corres-guard-DynCom*:
$[\![\bigwedge s.\ s \in g \Longrightarrow$ *L1corres ct* Γ (*B s*) (*B' s*)$]\!] \Longrightarrow$
L1corres ct Γ (*L1-seq* (*L1-guard* ($\lambda s.\ s \in g$)) (*gets B* $\gg\!=$ ($\lambda b.\ b$))) (*Guard f g* (*DynCom B'*))
⟨*proof*⟩

lemma *L1corres'-guard-DynCom-conseq*:
assumes *conseq*: $\bigwedge s.\ P\ s \Longrightarrow g\ s \Longrightarrow s \in g'$
assumes *corres*: $\bigwedge s.\ P\ s \Longrightarrow g\ s \Longrightarrow$ *L1corres' ct* Γ ($\lambda s.\ P\ s \wedge g\ s$) (*B s*) (*B' s*)
shows *L1corres' ct* Γ *P* (*L1-seq* (*L1-guard g*) (*gets B* $\gg\!=$ ($\lambda b.\ b$))) (*Guard f g'* (*DynCom B'*))
⟨*proof*⟩

lemma *L1corres′-guard-DynCom*:
⟦⋀*s*. *P s* ⟹ *s* ∈ *g* ⟹ *L1corres′ ct* Γ (λ*s*. *P s* ∧ *s* ∈ *g*) (*B s*) (*B′ s*)⟧ ⟹
L1corres′ ct Γ *P* (*L1-seq* (*L1-guard* (λ*s*. *s* ∈ *g*)) (*gets B* ≫= (λ*b*. *b*))) (*Guard f g* (*DynCom B′*))
⟨*proof*⟩

lemma *L1corres-DynCom*:
assumes *corres-f*: ⋀*s*. *L1corres ct* Γ (*g s*) (*f s*)
shows *L1corres ct* Γ (*gets g* ≫= (λ*b*. *b*)) (*DynCom f*)
⟨*proof*⟩

lemma *L1corres′-DynCom*:
assumes *corres-f*: ⋀*s*. *P s* ⟹ *L1corres′ ct* Γ *P* (*g s*) (*f s*)
shows *L1corres′ ct* Γ *P* (*gets g* ≫= (λ*b*. *b*)) (*DynCom f*)
⟨*proof*⟩

lemma *L1corres′-DynCom-fix-state*:
assumes *corres-f*: ⋀*s*. *P s* ⟹ *L1corres′ ct* Γ (λ*s′*. *P s′* ∧ *s′* = *s*) (*g s*) (*f s*)
shows *L1corres′ ct* Γ *P* (*gets g* ≫= (λ*b*. *b*)) (*DynCom f*)
⟨*proof*⟩

lemma *L1corres′-guard′*:
⟦ *L1corres′ ct* Γ (λ*s*. *P s* ∧ *s* ∈ *g*) *B B′* ⟧ ⟹
L1corres′ ct Γ *P* (*L1-seq* (*L1-guard* (λ*s*. *s* ∈ *g*)) *B*) (*Guard f g B′*)
⟨*proof*⟩

lemma *L1corres′-guarded*:
⟦ *L1corres′ ct* Γ (λ*s*. *P s* ∧ *s* ∈ *g*) *B B′* ⟧ ⟹
L1corres′ ct Γ *P* (*L1-guarded* (λ*s*. *s* ∈ *g*) *B*) (*Guard f g B′*)
⟨*proof*⟩

lemma *L1corres′-Guard-maybe-guard*:
L1corres′ ct Γ *P B* (*Guard f g B′*) ⟹ *L1corres′ ct* Γ *P B* (*maybe-guard f g B′*)
⟨*proof*⟩

lemma *L1corres′-guarded-DynCom-conseq*:
assumes *conseq*: ⋀*s*. *P s* ⟹ *g s* ⟹ *s* ∈ *g′*
assumes *corres-B*: ⋀*s*. *P s* ⟹ *g s* ⟹ *L1corres′ ct* Γ (λ*s*. *P s* ∧ *g s*) (*B s*) (*B′ s*)
shows *L1corres′ ct* Γ *P* (*L1-guarded g* (*gets B* ≫= (λ*b*. *b*))) (*maybe-guard f g′* (*DynCom B′*))
⟨*proof*⟩

lemma *L1corres′-guarded-DynCom*:
assumes *corres-B*: ⋀*s*. *P s* ⟹ *s* ∈ *g* ⟹ *L1corres′ ct* Γ (λ*s*. *P s* ∧ *s* ∈ *g*) (*B s*) (*B′ s*)
shows *L1corres′ ct* Γ *P* (*L1-guarded* (*L1-set-to-pred g*) (*gets B* ≫= (λ*b*. *b*))) (*maybe-guard f g* (*DynCom B′*))

⟨*proof*⟩

lemma *L1corres′-conseq*:
assumes *corres-Q*: *L1corres′ ct* Γ *Q B B′*
assumes *conseq*: ⋀*s*. *P s* ⟹ *Q s*
shows *L1corres′ ct* Γ *P B B′*
⟨*proof*⟩

lemma *L1corres-to-L1corres′*: *L1corres ct* Γ = *L1corres′ ct* Γ ⊤
⟨*proof*⟩

lemma *L1corres-guarded-DynCom-conseq*:
assumes *conseq*: ⋀*s*. *g s* ⟹ *s* ∈ *g′*
assumes *corres-B*: ⋀*s*. *g s* ⟹ *L1corres ct* Γ (*B s*) (*B′ s*)
shows *L1corres ct* Γ (*L1-guarded g* (*gets B* ≫= (λ*b*. *b*))) (*maybe-guard f g′* (*DynCom B′*))
⟨*proof*⟩

lemma *L1corres-guarded-DynCom*:
assumes *corres-B*: ⋀*s*. *s* ∈ *g* ⟹ *L1corres ct* Γ (*B s*) (*B′ s*)
shows *L1corres ct* Γ (*L1-guarded* (*L1-set-to-pred g*) (*gets B* ≫= (λ*b*. *b*))) (*maybe-guard f g* (*DynCom B′*))
⟨*proof*⟩

definition
L1-call-simpl check-term Gamma proc
= *do* {*s* ← *get-state*;
assert (*check-term* ⟶ *Gamma* ⊢ *Call proc* ↓ *Normal s*);
xs ← *select* {*t*. *Gamma* ⊢ ⟨*Call proc*, *Normal s*⟩ ⇒ *t*};
case xs :: (-, *strictc-errortype*) *xstate of*
Normal s ⇒ *set-state s*
| *Abrupt s* ⇒ *do* {*set-state s*; *throw* () }
| *Fault ft* ⇒ *fail*
| *Stuck* ⇒ *fail*
}

lemma *L1corres-call-simpl*:
L1corres ct Γ (*L1-call-simpl ct* Γ *proc*) (*Call proc*)
⟨*proof*⟩

lemma *L1corres-skip*:
L1corres ct Γ *L1-skip SKIP*
⟨*proof*⟩

lemma *L1corres-throw*:
L1corres ct Γ *L1-throw Throw*

⟨*proof*⟩

lemma *L1corres-seq*:
⟦ *L1corres ct* Γ *L L′*; *L1corres ct* Γ *R R′* ⟧ ⟹
L1corres ct Γ (*L1-seq L R*) (*L′* ;; *R′*)
⟨*proof*⟩

lemma *L1corres-modify*:
L1corres ct Γ (*L1-modify m*) (*Basic m*)
⟨*proof*⟩

lemma *L1corres-condition*:
⟦ *L1corres ct* Γ *L L′*; *L1corres ct* Γ *R R′* ⟧ ⟹
L1corres ct Γ (*L1-condition* (*L1-set-to-pred c*) *L R*) (*Cond c L′ R′*)
⟨*proof*⟩

lemma *L1corres-catch*:
⟦ *L1corres ct* Γ *L L′*; *L1corres ct* Γ *R R′* ⟧ ⟹
L1corres ct Γ (*L1-catch L R*) (*Catch L′ R′*)
⟨*proof*⟩

lemma *L1corres-while*:
⟦ *L1corres ct* Γ *B B′* ⟧ ⟹
L1corres ct Γ (*L1-while* (*L1-set-to-pred c*) *B*) (*While c B′*)
⟨*proof*⟩

lemma *L1corres-guard*:
⟦ *L1corres ct* Γ *B B′* ⟧ ⟹
L1corres ct Γ (*L1-seq* (*L1-guard* (*L1-set-to-pred c*)) *B*) (*Guard f c B′*)
⟨*proof*⟩

lemma *L1corres-spec*:
L1corres ct Γ (*L1-spec x*) (*com.Spec x*)
⟨*proof*⟩

lemma *L1-init-alt-def*:
L1-init upd ≡ *L1-spec* {(*s*, *t*). ∃ *v*. *t* = *upd* (λ-. *v*) *s*}
⟨*proof*⟩

lemma *L1corres-init*:
L1corres ct Γ (*L1-init upd*) (*lvar-nondet-init upd*)
⟨*proof*⟩

lemma *L1corres-guarded-spec*:
L1corres ct Γ (*L1-spec R*) (*guarded-spec-body F R*)
⟨*proof*⟩

lemma *L1corres-assume*:

L1corres ct Γ (*L1-assume* (*L1-rel-to-fun R*)) (*guarded-spec-body AssumeError R*)
⟨*proof*⟩

lemma *pred-conj-apply*[*simp*]: (*P and Q*) *s* ⟷ *P s* ∧ *Q s*
⟨*proof*⟩

lemma *L1corres-call*:
⟦ *L1corres ct* Γ *dest-fn* (*Call dest*) ⟧ ⟹
L1corres ct Γ
(*L1-call scope-setup dest-fn scope-teardown-norm scope-teardown-exn f*)
(*call-exn scope-setup dest scope-teardown-norm scope-teardown-exn* (λ- *t*. *Basic* (*f t*)))
⟨*proof*⟩

lemma (**in** *L1-functions*) *L1corres-dyn-call-conseq*:
assumes *conseq*: ⋀*s*. *g s* ⟹ *s* ∈ *g′*
assumes *corres-dest*: ⋀*s*. *g s* ⟹ *L1corres ct* Γ (*P* (*dest s*)) (*Call* (*dest s*))
shows
L1corres ct Γ
(*L1-dyn-call g scope-setup dest scope-teardown-norm scope-teardown-exn result*)
(*dynCall-exn f g′ scope-setup dest scope-teardown-norm scope-teardown-exn* (λ- *t*. *Basic* (*result t*)))
⟨*proof*⟩

lemma (**in** *L1-functions*) *L1corres-dyn-call-same-guard*:
assumes *eq*: *L1-set-to-pred g* ≡ *g′*
assumes *corres-dest*: ⋀*s*. *g′ s* ⟹ *L1corres ct* Γ (*P* (*dest s*)) (*Call* (*dest s*))
shows
L1corres ct Γ
(*L1-dyn-call g′ scope-setup dest scope-teardown-norm scope-teardown-exn result*)
(*dynCall-exn f g scope-setup dest scope-teardown-norm scope-teardown-exn* (λ- *t*. *Basic* (*result t*)))
⟨*proof*⟩

lemma (**in** *L1-functions*) *L1corres-dyn-call-add-and-select-guard*:
assumes *eq*: *L1-set-to-pred g* ≡ *g′*
assumes *corres-dest*: ⋀*s*. *G s* ⟹ *L1corres ct* Γ (*P* (*dest s*)) (*Call* (*dest s*))
shows
L1corres ct Γ
(*L1-dyn-call* (*G and g′*) *scope-setup dest scope-teardown-norm scope-teardown-exn result*)
(*dynCall-exn f g scope-setup dest scope-teardown-norm scope-teardown-exn* (λ- *t*. *Basic* (*result t*)))
⟨*proof*⟩

lemma *L1-seq-guard-merge*: *L1-seq* (*L1-guard P*) (*L1-seq* (*L1-guard Q*) *c*) = *L1-seq* (*L1-guard* (*P and Q*)) *c*
⟨*proof*⟩

lemma *and-unfold*: (*and*) = (λ*P Q s*. *P s* ∧ *Q s*)
⟨*proof*⟩

lemma *L1-seq-guard-eq*: (⋀*s*. *P s* = *Q s*) ⟹ *L1-seq* (*L1-guard P*) *c* = *L1-seq* (*L1-guard Q*) *c*
⟨*proof*⟩

lemma *foldr-and-commute*: ⋀*s*. *foldr* (*and*) *gs* (*P and g*) *s* = (*g s* ∧ *foldr* (*and*) *gs P s*)
⟨*proof*⟩

lemma *L1corres-fail*:
L1corres ct Γ *L1-fail X*
⟨*proof*⟩

lemma *L1corres-prepend-unknown-var'*:
⟦ *L1corres ct* Γ *A C*; ⋀*s*::'*s*::*type*. *X* (λ-::'*a*::*type*. (*X' s*)) *s* = *s* ⟧ ⟹ *L1corres ct* Γ (*L1-seq* (*L1-init X*) *A*) *C*
⟨*proof*⟩

lemma *L1-catch-seq-join*: *no-throw* (λ-. *True*) *A* ⟹ *L1-seq A* (*L1-catch B C*) = (*L1-catch* (*L1-seq A B*) *C*)
⟨*proof*⟩

lemma *no-throw-L1-init* [*simp*]: *no-throw P* (*L1-init f*)
⟨*proof*⟩

lemma *L1corres-prepend-unknown-var*:
⟦ *L1corres ct* Γ (*L1-catch A B*) *C*; ⋀*s*. *X* (λ-::'*d*::*type*. (*X' s*)) *s* = *s* ⟧
⟹ *L1corres ct* Γ (*L1-catch* (*L1-seq* (*L1-init X*) *A*) *B*) *C*
⟨*proof*⟩

lemma *L1corres-Call*:
⟦ Γ *X'* = *Some Z'*; *L1corres ct* Γ *Z Z'* ⟧ ⟹
L1corres ct Γ *Z* (*Call X'*)
⟨*proof*⟩

lemma *L1-call-corres* [*fundef-cong*]:
⟦ *scope-setup* = *scope-setup'*;
dest-fn = *dest-fn'*;
scope-teardown = *scope-teardown'*;

return-xf = return-xf' ⟧ ⟹
L1-call scope-setup dest-fn scope-teardown return-xf =
L1-call scope-setup' dest-fn' scope-teardown' return-xf'
⟨*proof*⟩

lemma *L1-corres-cleanup*:
L1corres ct Γ (*do* {*y* <− *state-select* {(*s*,*t*). *t* = *cleanup s*};
return ()
})
(*Basic cleanup*)
⟨*proof*⟩

lemma *L1-corres-spec-cleanup*:
L1corres ct Γ (*do* { *y* <− *state-select cleanup*;
return ()
})
(*com.Spec cleanup*)
⟨*proof*⟩

lemma *L1-corres-cleanup-throw*:
L1corres ct Γ (*do* { - <− *state-select* {(*s*,*t*). *t* = *cleanup s*};
throw ()
})
(*Basic cleanup*;; *THROW*)
⟨*proof*⟩

lemma *L1-corres-spec-cleanup-throw*:
L1corres ct Γ (*do*{ - <− *state-select cleanup*;
throw ()
})
(*com.Spec cleanup*;; *THROW*)
⟨*proof*⟩

lemma *on-exit-unit-def*: (*on-exit f cleanup*::(*unit*, *unit*, *'s*) *exn-monad*) =
bind-handle f
(λ*v*. *bind* (*state-select cleanup*) (λ-. *return* ()))
(λ*e*. *bind* (*state-select cleanup*)(λ-. *throw* ()))
⟨*proof*⟩

lemma *on-exit-catch-conv*: *on-exit f cleanup* =
do {
r ← (*f* <*catch*> (λ*e*. *state-select cleanup* >>= (λ-. *throw e*)));
state-select cleanup;
return r
}
⟨*proof*⟩

lemma *L2corres-on-exit'*:
assumes *m-c*: *L1corres ct* Γ *m c*
shows *L1corres ct* Γ (*on-exit m* {(*s*,*t*). *t* = *cleanup s*}) (*On-Exit c* (*Basic cleanup*))
⟨*proof*⟩

lemma *L2corres-on-exit*:
assumes *m-c*: *L1corres ct* Γ *m c*
shows *L1corres ct* Γ (*on-exit m cleanup*) (*On-Exit c* (*com.Spec cleanup*))
⟨*proof*⟩

definition
refines-simpl :: *bool* ⇒ ('*p* ⇒ ('*s*, '*p*, *strictc-errortype*) *com option*) ⇒
('*s*, '*p*, *strictc-errortype*) *com* ⇒
(('*e*::*default*, '*a*, '*t*) *spec-monad*) ⇒
'*s* ⇒ '*t* ⇒ (('*s*, *strictc-errortype*) *xstate* ⇒ (('*e*, '*a*) *exception-or-result* ∗ '*t*) ⇒ *bool*) ⇒ *bool* **where**
refines-simpl ct Γ *c m s t R* ≡
succeeds m t ⟶
((∀ *s'*. Γ ⊢ ⟨*c*, *Normal s*⟩ ⇒ *s'* ⟶
(*s'* ∈ {*Fault AssumeError*, *Fault StackOverflow*} ∨
(∃ *r t'*. *reaches m t r t'* ∧ *R s'* (*r*, *t'*)))) ∧
(*ct* ⟶ Γ ⊢ *c* ↓ *Normal s*))

lemma *refines-simplI*:
assumes *termi*: *succeeds m t* ⟹ *ct* ⟹ Γ ⊢ *c* ↓ *Normal s*
assumes *sim*: ⋀*s'*. *succeeds m t* ⟹ Γ ⊢ ⟨*c*, *Normal s*⟩ ⇒ *s'* ⟹ *s'* ∉ {*Fault AssumeError*, *Fault StackOverflow*}
⟹ ∃ *r t'*. *reaches m t r t'* ∧ *R s'* (*r*, *t'*)
shows *refines-simpl ct* Γ *c m s t R*
⟨*proof*⟩

definition
rel-L1 :: ('*s*, *strictc-errortype*) *xstate* ⇒ ('*e*, '*a*) *xval* × '*s* ⇒ *bool* **where**
rel-L1 ≡ λ*s* (*r*, *t*). (*case s of*
Normal s' ⇒ (∃ *x*. *r* = *Result x*) ∧ *t* = *s'*
| *Abrupt s'* ⇒ (∃ *x*. *r* = *Exn x*) ∧ *t* = *s'*
| *Fault e* ⇒ *False*
| *Stuck* ⇒ *False*)

lemma *rel-L1-unit*:
rel-L1 = (λ*s* (*r*, *t*). (*case s of*
Normal s' ⇒ *r* = *Result* () ∧ *t* = *s'*
| *Abrupt s'* ⇒ *r* = *Exn* () ∧ *t* = *s'*
| *Fault e* ⇒ *False*
| *Stuck* ⇒ *False*))
⟨*proof*⟩

lemma *rel-L1-conv* [*simp*]:
rel-L1 (*Normal s*) (*r*, *t*) = ((∃ *x*. *r* = *Result x*) ∧ *t* = *s*)
rel-L1 (*Abrupt s*) (*r*, *t*) = ((∃ *x*. *r* = *Exn x*) ∧ *t* = *s*)
rel-L1 (*Fault e*) *x* = *False*
rel-L1 Stuck x = *False*
⟨*proof*⟩

lemma *refines-simpl-rel-L1I*:
assumes *termi*: *succeeds m t* ⟹ *ct* ⟹ Γ ⊢ *c* ↓ *Normal s*
assumes *sim-Normal*: ⋀*s'*. *succeeds m t* ⟹ Γ ⊢ ⟨*c*, *Normal s*⟩ ⇒ *Normal s'*
⟹ ∃ *r*. *reaches m t* (*Result r*) *s'*
assumes *sim-Abrupt*: ⋀*s'*. *succeeds m t* ⟹ Γ ⊢ ⟨*c*, *Normal s*⟩ ⇒ *Abrupt s'*
⟹ ∃ *r*. *reaches m t* (*Exn r*) *s'*
assumes *sim-Fault*: ⋀*e*. *succeeds m t* ⟹ Γ ⊢ ⟨*c*, *Normal s*⟩ ⇒ *Fault e*
⟹ *e* ∈ {*AssumeError*, *StackOverflow*}
assumes *sim-Stuck*: *succeeds m t* ⟹ Γ ⊢ ⟨*c*, *Normal s*⟩ ⇒ *Stuck*
⟹ *False*
shows *refines-simpl ct* Γ *c m s t rel-L1*
⟨*proof*⟩

lemma *L1corres-refines-simpl*:
L1corres ct Γ *m c* ⟹ *refines-simpl ct* Γ *c m s s rel-L1*
⟨*proof*⟩

lemma *refines-simpl-L1corres*:
assumes ⋀*s*. *refines-simpl ct* Γ *c m s s rel-L1*
shows *L1corres ct* Γ *m c*
⟨*proof*⟩

theorem *L1corres-refines-simpl-conv*:
L1corres ct Γ *m c* ⟷ (∀ *s*. *refines-simpl ct* Γ *c m s s rel-L1*)
⟨*proof*⟩

lemma *refines-simpl-DynCom*:
refines-simpl ct Γ (*c s*) *m s t R* ⟹ *refines-simpl ct* Γ (*DynCom c*) *m s t R*
⟨*proof*⟩

lemma *refines-simpl-StackOverflow*:
assumes *c*: *s* ∈ *g* ⟹ *refines-simpl ct* Γ *c m s t R*
shows *refines-simpl ct* Γ (*Guard StackOverflow g c*) *m s t R*
⟨*proof*⟩

lemma *refines-simpl-rel-L1-bind*:
fixes *m1*:: ('*e*, '*a*, '*s*) *exn-monad*
fixes *m2*:: '*a* ⇒ ('*e*, '*b*, '*s*) *exn-monad*
assumes *c1*: *refines-simpl ct* Γ *c1 m1 s s rel-L1*
assumes *c2*: ⋀*r s'*. *succeeds m1 s* ⟹ Γ ⊢ ⟨*c1*, *Normal s*⟩ ⇒ *Normal s'* ⟹

reaches m1 s (Result r) s' $\Longrightarrow$
refines-simpl ct Γ c2 (m2 r) s' s' rel-L1
shows *refines-simpl ct Γ (c1;;c2) (m1 >>= m2) s s rel-L1*
⟨*proof*⟩

lemma *refines-simpl-rel-L1-catch*:
assumes *L*: *refines-simpl ct Γ L' L s s rel-L1*
assumes *R*: $\bigwedge$*s. refines-simpl ct Γ R' R s s rel-L1*
shows *refines-simpl ct Γ (Catch L' R') (L1-catch L R) s s rel-L1*
⟨*proof*⟩

lemmas *refines-simpl-cleanup = L1corres-refines-simpl [OF L1-corres-cleanup]*
lemmas *refines-simpl-cleanup-throw = L1corres-refines-simpl [OF L1-corres-cleanup-throw]*
lemmas *refines-simpl-spec-cleanup = L1corres-refines-simpl [OF L1-corres-spec-cleanup]*
lemmas *refines-simpl-spec-cleanup-throw = L1corres-refines-simpl [OF L1-corres-spec-cleanup-throw]*

lemma *refines-simpl-rel-L1-on-exit'*:
fixes *m*:: *'s L1-monad*
assumes *m-c*: *refines-simpl ct Γ c m s s rel-L1*
shows *refines-simpl ct Γ (On-Exit c (Basic cleanup)) (on-exit m {(s,t). t = cleanup s}) s s rel-L1*
⟨*proof*⟩

lemma *refines-simpl-rel-L1-on-exit*:
fixes *m*:: *'s L1-monad*
assumes *m-c*: *refines-simpl ct Γ c m s s rel-L1*
shows *refines-simpl ct Γ (On-Exit c (com.Spec cleanup)) (on-exit m cleanup) s s rel-L1*
⟨*proof*⟩

named-theorems *L1corres-with-fresh-stack-ptr*

context *stack-heap-state*
begin
lemma *refines-simpl-rel-L1-with-fresh-stack-ptr*:
fixes *m*:: *'a::mem-type ptr* $\Rightarrow$ *'s L1-monad*
assumes *c-m*: $\bigwedge$*p s. refines-simpl ct Γ (c p) (m p) s s rel-L1*
shows *refines-simpl ct Γ (With-Fresh-Stack-Ptr n I c) (with-fresh-stack-ptr n I m) s s rel-L1*
⟨*proof*⟩

lemma *L1corres-with-fresh-stack-ptr*[*L1corres-with-fresh-stack-ptr*]:
fixes *m*:: *'a::mem-type ptr* $\Rightarrow$ *'s L1-monad*
assumes *c-m*: $\bigwedge$*p. L1corres ct Γ (m p) (c p)*

shows *L1corres ct* Γ (*with-fresh-stack-ptr n I m*) (*With-Fresh-Stack-Ptr n I c*)
⟨*proof*⟩
end

definition *UNDEFINED-FUNCTION* ≡ *False*

definition
undefined-function-body :: ('*a*, *int*, *strictc-errortype*) *com*
where
undefined-function-body ≡ *Guard UndefinedFunction* {*x*. *UNDEFINED-FUNCTION*} *SKIP*

definition
init-return-undefined-function-body::((*'a* ⇒ *'a*) ⇒ ((*'g*, *'l*, *'e*, *'z*) *state-scheme* ⇒ (*'g*, *'l*, *'e*, *'z*) *state-scheme*))
⇒ ((*'g*, *'l*, *'e*, *'z*) *state-scheme*, *int*, *strictc-errortype*) *com*
where
init-return-undefined-function-body ret ≡ *Seq* (*lvar-nondet-init ret*) (*Guard UndefinedFunction* {*x*. *UNDEFINED-FUNCTION*} *SKIP*)

lemma *L1corres-undefined-call*:
L1corres ct Γ ((*L1-seq* (*L1-guard* (*L1-set-to-pred* {*x*. *UNDEFINED-FUNCTION*})) *L1-skip*)) (*Call X'*)
⟨*proof*⟩

lemma *L1-UNDEFINED-FUNCTION-fail*: (*L1-guard* (*L1-set-to-pred* {*x*. *UNDEFINED-FUNCTION*})) = *L1-fail*
⟨*proof*⟩

lemma *L1-seq-fail*: *L1-seq L1-fail X* = *L1-fail*
⟨*proof*⟩

lemma *L1-seq-init-fail*: (*L1-seq* (*L1-init ret*) *L1-fail*) = *L1-fail*
⟨*proof*⟩

lemma *L1-corres-L1-fail*: *L1corres ct* Γ *L1-fail X*
⟨*proof*⟩

lemma *L1corres-init-return-undefined-call*:
L1corres ct Γ (*L1-seq* (*L1-init ret*) ((*L1-seq* (*L1-guard* (*L1-set-to-pred* {*x*. *UNDEFINED-FUNCTION*})) *L1-skip*))) (*Call X'*)
⟨*proof*⟩

named-theorems *L1unfold*

named-theorems *L1except*

lemma *signed-bounds-one-to-nat*: $n <s 1 \Longrightarrow 0 \leq s\ n \Longrightarrow unat\ n = 0$
⟨*proof*⟩

lemma *signed-bounds-to-nat-boundsF*: $n <s numeral\ B \Longrightarrow 0 \leq s\ n \Longrightarrow unat\ n < numeral\ B$
⟨*proof*⟩

lemma *word-bounds-to-nat-boundsF*: $(n::'a::len\ word) < numeral\ B \Longrightarrow 0 \leq s\ n \Longrightarrow unat\ n < numeral\ B$
⟨*proof*⟩

lemma *word-bounds-one-to-nat*: $(n::'a::len\ word) < 1 \Longrightarrow 0 \leq s\ n \Longrightarrow unat\ n = 0$
⟨*proof*⟩

lemma *monotone-L1-seq-le* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≤) *X*
assumes *mono-Y*: *monotone R* (≤) *Y*
shows *monotone R* (≤)
(λ*f*. (*L1-seq* (*X f*) (*Y f*)))
⟨*proof*⟩

lemma *monotone-L1-seq-ge* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≥) *X*
assumes *mono-Y*: *monotone R* (≥) *Y*
shows *monotone R* (≥)
(λ*f*. (*L1-seq* (*X f*) (*Y f*)))
⟨*proof*⟩

lemma *monotone-L1-catch-le* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≤) *X*
assumes *mono-Y*: *monotone R* (≤) *Y*
shows *monotone R* (≤)
(λ*f*. (*L1-catch* (*X f*) (*Y f*)))
⟨*proof*⟩

lemma *monotone-L1-catch-ge* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≥) *X*
assumes *mono-Y*: *monotone R* (≥) *Y*
shows *monotone R* (≥)
(λ*f*. (*L1-catch* (*X f*) (*Y f*)))
⟨*proof*⟩

lemma *monotone-L1-condition-le* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≤) *X*
assumes *mono-Y*: *monotone R* (≤) *Y*
shows *monotone R* (≤)
(λ*f*. (*L1-condition C* (*X f*) (*Y f*)))
⟨*proof*⟩

lemma *monotone-L1-condition-ge* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≥) *X*
assumes *mono-Y*: *monotone R* (≥) *Y*
shows *monotone R* (≥)
(λ*f*. (*L1-condition C* (*X f*) (*Y f*)))
⟨*proof*⟩

lemma *monotone-L1-guarded-le* [*partial-function-mono*]:
assumes *mono-X* [*partial-function-mono*]: *monotone R* (≤) *X*
shows *monotone R* (≤)
(λ*f*. (*L1-guarded C* (*X f*)))
⟨*proof*⟩

lemma *monotone-L1-guarded-ge* [*partial-function-mono*]:
assumes *mono-X* [*partial-function-mono*]: *monotone R* (≥) *X*
shows *monotone R* (≥)
(λ*f*. (*L1-guarded C* (*X f*)))
⟨*proof*⟩

lemma *monotone-L1-while-le* [*partial-function-mono*]:
assumes *mono-B*: *monotone R* (≤) (λ*f*. *B f*)
shows *monotone R* (≤) (λ*f*. *L1-while C* (*B f*))
⟨*proof*⟩

lemma *monotone-L1-while-ge* [*partial-function-mono*]:
assumes *mono-B*: *monotone R* (≥) (λ*f*. *B f*)
shows *monotone R* (≥) (λ*f*. *L1-while C* (*B f*))
⟨*proof*⟩

lemma *monotone-L1-call-le* [*partial-function-mono*]:
assumes *X*[*partial-function-mono*]: *monotone R* (≤) *X*
shows *monotone R* (≤)
(λ*f*. *L1-call scope-setup* (*X f*) *scope-teardown result-exn return-xf*)
⟨*proof*⟩

lemma *monotone-L1-call-ge* [*partial-function-mono*]:
assumes *X*[*partial-function-mono*]: *monotone R* (≥) *X*

shows *monotone R* (≥)
(λ*f*. *L1-call scope-setup* (*X f*) *scope-teardown result-exn return-xf*)
⟨*proof*⟩

end

18.1 Peep-hole L1 optimisations

theory *L1Peephole*
imports *L1Defs*
begin

named-theorems *L1opt*

lemma *L1-seq-assoc* [*L1opt*]: (*L1-seq* (*L1-seq X Y*) *Z*) = (*L1-seq X* (*L1-seq Y Z*))
⟨*proof*⟩

lemma *L1-seq-skip* [*L1opt*]:
L1-seq A L1-skip = *A*
L1-seq L1-skip A = *A*
⟨*proof*⟩

lemma *L1-condition-true* [*L1opt*]: *L1-condition* (λ-. *True*) *A B* = *A*
⟨*proof*⟩

lemma *L1-condition-false* [*L1opt*]: *L1-condition* (λ-. *False*) *A B* = *B*
⟨*proof*⟩

lemma *L1-condition-same* [*L1opt*]: *L1-condition C A A* = *A*
⟨*proof*⟩

lemma *L1-fail-seq* [*L1opt*]: *L1-seq L1-fail X* = *L1-fail*
⟨*proof*⟩

lemma *L1-throw-seq* [*L1opt*]: *L1-seq L1-throw X* = *L1-throw*
⟨*proof*⟩

lemma *L1-fail-propagates* [*L1opt*]:
L1-seq L1-skip L1-fail = *L1-fail*
L1-seq (*L1-modify M*) *L1-fail* = *L1-fail*
L1-seq (*L1-spec S*) *L1-fail* = *L1-fail*
L1-seq (*L1-guard G*) *L1-fail* = *L1-fail*
L1-seq (*L1-init I*) *L1-fail* = *L1-fail*
L1-seq L1-fail L1-fail = *L1-fail*
⟨*proof*⟩

lemma *L1-condition-distrib*:

L1-seq (*L1-condition C L R*) *X* = *L1-condition C* (*L1-seq L X*) (*L1-seq R X*)
⟨*proof*⟩

lemmas *L1-fail-propagate-condition* [*L1opt*] = *L1-condition-distrib* [**where** *X*=*L1-fail*]

lemma *L1-fail-propagate-catch* [*L1opt*]:
(*L1-seq* (*L1-catch L R*) *L1-fail*) = (*L1-catch* (*L1-seq L L1-fail*) (*L1-seq R L1-fail*))
⟨*proof*⟩

lemma *L1-guard-false* [*L1opt*]:
L1-guard (λ-. *False*) = *L1-fail*
⟨*proof*⟩

lemma *L1-guard-true* [*L1opt*]:
L1-guard (λ-. *True*) = *L1-skip*
⟨*proof*⟩

lemma *L1-condition-fail-lhs* [*L1opt*]:
L1-condition C L1-fail A = *L1-seq* (*L1-guard* (λ*s*. ¬ *C s*)) *A*
⟨*proof*⟩

lemma *L1-condition-fail-rhs* [*L1opt*]:
L1-condition C A L1-fail = *L1-seq* (*L1-guard C*) *A*
⟨*proof*⟩

lemma *L1-catch-fail* [*L1opt*]: *L1-catch L1-fail A* = *L1-fail*
⟨*proof*⟩

lemma *L1-while-fail* [*L1opt*]: *L1-while C L1-fail* = *L1-guard* (λ*s*. ¬ *C s*)
⟨*proof*⟩

lemma *whileLoop-succeeds-terminates-infinite*:
assumes *run* (*whileLoop* (λ-. *C*) (λ-. *skip*) ()) *s* ≠ ⊤
shows *C s* ⟹ *False*
⟨*proof*⟩

lemma *run-whileLoop-infinite*: *run* (*whileLoop* (λ-. *C*) (λ-. *skip*) ()) *s* = *run* (*guard* (λ*s*. ¬ *C s*)) *s*
⟨*proof*⟩

lemma *whileLoop-infinite*: *whileLoop* (λ-. *C*) (λ-. *skip*) () = *guard* (λ*s*. ¬ *C s*)
⟨*proof*⟩

lemma *L1-while-infinite* [*L1opt*]: *L1-while C L1-skip* = *L1-guard* (λ*s*. ¬ *C s*)
⟨*proof*⟩

lemma *L1-while-false* [*L1opt*]:
  L1-while (λ-. *False*) *B* = *L1-skip*
  ⟨*proof*⟩

declare *ucast-id* [*L1opt*]
declare *scast-id* [*L1opt*]
declare *L1-set-to-pred-def* [*L1opt*]
declare *L1-rel-to-fun-def* [*L1opt*]

lemma *in-set-to-pred* [*L1opt*]: (λ*s*. *s* ∈ {*x*. *P x*}) = *P*
  ⟨*proof*⟩

lemma *in-set-if-then* [*L1opt*]: (*s* ∈ (*if P then A else B*)) = (*if P then* (*s* ∈ *A*) *else* (*s* ∈ *B*))
  ⟨*proof*⟩

lemma *Pair-unit-Image*[*L1opt*]: *Pair* () ‘ *S* ‘‘ {*x*} = {(*u*, *x'*). (*x*, *x'*) ∈ *S*}
  ⟨*proof*⟩

lemmas *if-simps* =
  if-x-Not if-Not-x if-cancel if-True if-False if-bool-simps

declare *empty-iff* [*L1opt*]
declare *UNIV-I* [*L1opt*]
declare *singleton-iff* [*L1opt*]
declare *if-simps* [*L1opt*]
declare *simp-thms* [*L1opt*]

lemma *L1-call-stop-cong*: (*L1-call f n g r*) = (*L1-call f n g r*)
  ⟨*proof*⟩

lemma *L1-merge-assignments* : (*L1-seq* (*L1-modify f*)
                (*L1-seq* (*L1-modify g*) *X*)) ≡ *L1-seq* (*L1-modify* (λ*s*. *g* (*f s*))) *X*
  ⟨*proof*⟩

end

theory *SimplConv*
imports *L1Peephole*

begin

named-theorems *L1unfold*
declare *creturn-def* [*L1unfold*]
declare *creturn-void-def* [*L1unfold*]
declare *cbreak-def* [*L1unfold*]
declare *cgoto-def* [*L1unfold*]
declare *whileAnno-def* [*L1unfold*]
declare *ccatchbrk-def* [*L1unfold*]
declare *ccatchgoto-def* [*L1unfold*]
declare *ccatchreturn-def* [*L1unfold*]
declare *cexit-def* [*L1unfold*]

lemma *switch-alt-defs* [*L1unfold*]:
switch x [] ≡ *SKIP*
switch v ((*a*, *b*) # *vs*) ≡ *Cond* {*s*. *v s* ∈ *a*} *b* (*switch v vs*)
⟨*proof*⟩

lemma *sless-positive* [*simp*]:
⟦ *a* < *n*; *n* ≤ (2 ^(*len-of TYPE*('*a*) − 1)) − 1 ⟧ ⟹ (*a* :: ('*a*::{*len*}) *word*) <*s* *n*
⟨*proof*⟩

lemma *sle-positive* [*simp*]:
⟦ *a* ≤ *n*; *n* ≤ (2 ^(*len-of TYPE*('*a*) − 1)) − 1 ⟧ ⟹ (*a* :: ('*a*::{*len*}) *word*) <=*s* *n*
⟨*proof*⟩

lemmas [*L1except*] =
L1-set-to-pred-def in-set-to-pred in-set-if-then
L1-rel-to-fun-def Pair-unit-Image
L1-seq-assoc

end

theory *CorresXF*
imports
CCorresE
begin

definition *corresXF-simple st xf P M M′* ≡
 ∀ *s*. (*P s* ∧ *succeeds M* (*st s*)) ⟶ (∀ *r′ t′*. *reaches M′ s r′ t′* ⟶
 reaches M (*st s*) (*xf r′ t′*) (*st t′*)) ∧ *succeeds M′ s*

definition *corresXF st ret-xf ex-xf P A C* ≡
 ∀ *s*. *P s* ∧ *succeeds A* (*st s*) ⟶
 (∀ *r t*. *reaches C s r t* ⟶
 (*case r of*
 Exn r ⇒ *reaches A* (*st s*) (*Exn* (*ex-xf r t*)) (*st t*)
 | *Result r* ⇒ *reaches A* (*st s*) (*Result* (*ret-xf r t*)) (*st t*)))
 ∧ *succeeds C s*

definition *rel-XF st ret-xf ex-xf Q* ≡ λ(*r*, *t*) (*r′*, *t′*).
 t′ = *st t* ∧
 rel-xval (λ*e e′*. *e′* = *ex-xf e t*) (λ*v v′*. *v′* = *ret-xf v t*) *r r′* ∧
 Q r t

lemma *corresXF-refines-iff*:
 corresXF st ret-xf ex-xf P A C ⟷
 (∀ *s*. *P s* ⟶ *refines C A s* (*st s*) (*rel-XF st ret-xf ex-xf* (λ- -. *True*)))
 ⟨*proof*⟩

definition *corresXF-post st ret-xf ex-xf P Q A C* ≡
 ∀ *s*. *P s* ∧ *succeeds A* (*st s*) ⟶
 (∀ *r t*. *reaches C s r t* ⟶ *Q s r t* ∧
 (*case r of*
 Exn r ⇒ *reaches A* (*st s*) (*Exn* (*ex-xf r t*)) (*st t*)
 | *Result r* ⇒ *reaches A* (*st s*) (*Result* (*ret-xf r t*)) (*st t*)))
 ∧ *succeeds C s*

lemma *corresXF-post-refines-iff*:
 corresXF-post st ret-xf ex-xf P Q A C ⟷
 (∀ *s*. *P s* ⟶ *refines C A s* (*st s*) (*rel-XF st ret-xf ex-xf* (*Q s*)))
 ⟨*proof*⟩

lemma *corresXF-post-to-corresXF*:
 corresXF-post st ret-xf ex-xf P Q A C ⟹ *corresXF st ret-xf ex-xf P A C*
 ⟨*proof*⟩

lemma *corresXF-corres-XF-post-conv*:
 corresXF st ret-xf ex-xf P A C = *corresXF-post st ret-xf ex-xf P* (λ- - -. *True*) *A C*
 ⟨*proof*⟩

lemma *corresXF-simple-corresXF*:
(*corresXF-simple st*
(λ*x s*. *case x of*
Exn r ⇒ *Exn* (*ex-state r s*)
| *Result r* ⇒ (*Result* (*ret-state r s*))) *P M M′*)
= (*corresXF st ret-state ex-state P M M′*)
⟨*proof*⟩

lemma *corresXF-simpleI*: ⟦
⋀*s′ t′ r′*. ⟦*P s′*; *succeeds M* (*st s′*); *reaches M′ s′ r′ t′*⟧
⟹ *reaches M* (*st s′*) (*xf r′ t′*) (*st t′*);
⋀*s′*. ⟦*P s′*; *succeeds M* (*st s′*) ⟧ ⟹ *succeeds M′ s′*
⟧ ⟹ *corresXF-simple st xf P M M′*
⟨*proof*⟩

lemma *corresXF-I*: ⟦
⋀*s′ t′ r′*. ⟦*P s′*; *succeeds M* (*st s′*); *reaches M′ s′* (*Result r′*) *t′*⟧
⟹ *reaches M* (*st s′*) (*Result* (*ret-state r′ t′*)) (*st t′*);
⋀*s′ t′ r′*. ⟦*P s′*; *succeeds M* (*st s′*); *reaches M′ s′* (*Exn r′*) *t′*⟧
⟹ *reaches M* (*st s′*) (*Exn* (*ex-state r′ t′*)) (*st t′*);
⋀*s′*. ⟦*P s′*; *succeeds M* (*st s′*) ⟧ ⟹ *succeeds M′ s′*
⟧ ⟹ *corresXF st ret-state ex-state P M M′*
⟨*proof*⟩

lemma *ccpo-prod-gfp-gfp*:
class.ccpo
(*prod-lub Inf Inf* :: (('*a*::*complete-lattice* ∗ '*b* :: *complete-lattice*) *set* ⇒ -))
(*rel-prod* (≥) (≥)) (*mk-less* (*rel-prod* (≥) (≥)))
⟨*proof*⟩

lemma *admissible-mem*: *ccpo.admissible Inf* (≥) (λ*A*. *x* ∈ *A*)
⟨*proof*⟩

lemma *admissible-nondet-ord-corresXF*:
ccpo.admissible Inf (≥) (λ*A*. *corresXF st R E P A C*)
⟨*proof*⟩

lemma *corresXF-top*: *corresXF st ret-xf ex-xf P* ⊤ *C*
⟨*proof*⟩

lemma *admissible-nondet-ord-corresXF-post*:
ccpo.admissible Inf (≥) (λ*A*. *corresXF-post st R E P Q A C*)
⟨*proof*⟩

lemma *corresXF-post-top*: *corresXF-post st ret-xf ex-xf P Q* $\top$ *C*
⟨*proof*⟩

lemma *corresXF-assume-pre*:
⟦ $\bigwedge$*s s'*. ⟦ *P s'*; *s* = *st s'* ⟧ $\Longrightarrow$ *corresXF st xf-normal xf-exception P L R* ⟧ $\Longrightarrow$ *corresXF st xf-normal xf-exception P L R*
⟨*proof*⟩

lemma *corresXF-assume-fix-pre*:
⟦ $\bigwedge$*s s'*. ⟦ *P s'*; *s* = *st s'* ⟧ $\Longrightarrow$ *corresXF st xf-normal xf-exception* (λ*s*. *s* = *s'* ∧ *P s*) *L R* ⟧ $\Longrightarrow$ *corresXF st xf-normal xf-exception P L R*
⟨*proof*⟩

lemma *corresXF-guard-imp*:
⟦ *corresXF st xf-normal xf-exception Q f g*; $\bigwedge$*s*. *P s* $\Longrightarrow$ *Q s* ⟧
$\Longrightarrow$ *corresXF st xf-normal xf-exception P f g*
⟨*proof*⟩

lemma *corresXF-return*:
⟦ $\bigwedge$*s*. ⟦ *P s* ⟧ $\Longrightarrow$ *xf-normal b s* = *a* ⟧ $\Longrightarrow$
corresXF st xf-normal xf-exception P (*return a*) (*return b*)
⟨*proof*⟩

lemma *corresXF-gets*:
⟦ $\bigwedge$*s*. *P s* $\Longrightarrow$ *ret* (*g s*) *s* = *f* (*st s*) ⟧ $\Longrightarrow$
corresXF st ret ex P (*gets f*) (*gets g*)
⟨*proof*⟩

lemma *corresXF-insert-guard*:
⟦ *corresXF st ret ex Q A C*; $\bigwedge$*s*. ⟦ *P s* ⟧ $\Longrightarrow$ *G* (*st s*) $\longrightarrow$ *Q s* ⟧ $\Longrightarrow$
corresXF st ret ex P (*guard G* >>= (λ-. *A*)) *C*
⟨*proof*⟩

lemma *corresXF-exec-abs-guard*:
corresXF st ret-xf ex-xf (λ*s*. *P s* ∧ *G* (*st s*)) (*A* ()) *C* $\Longrightarrow$ *corresXF st ret-xf ex-xf P* (*guard G* >>= *A*) *C*
⟨*proof*⟩

lemma *corresXF-simple-exec*:
⟦ *corresXF-simple st xf P A B*; *reaches B s r' s'*; *succeeds A* (*st s*); *P s* ⟧
$\Longrightarrow$ *reaches A* (*st s*) (*xf r' s'*) (*st s'*)
⟨*proof*⟩

lemma *corresXF-simple-fail*:
⟦ *corresXF-simple st xf P A B*; ¬ *succeeds B s*; *P s* ⟧
$\Longrightarrow$ ¬ *succeeds A* (*st s*)
⟨*proof*⟩

lemma *corresXF-simple-no-fail*:

⟦ *corresXF-simple st xf P A B*; *succeeds A* (*st s*); *P s* ⟧
⟹ *succeeds B s*
⟨*proof*⟩

lemma *corresXF-exec-normal*:
⟦ *corresXF st ret ex P A B*; *reaches B s* (*Result r′*) *s′*; *succeeds A* (*st s*); *P s* ⟧
⟹ *reaches A* (*st s*) (*Result* (*ret r′ s′*)) (*st s′*)
⟨*proof*⟩

lemma *corresXF-exec-except*:
⟦ *corresXF st ret ex P A B*; *reaches B s* (*Exn r′*) *s′*; *succeeds A* (*st s*); *P s* ⟧
⟹ *reaches A* (*st s*) (*Exn* (*ex r′ s′*)) (*st s′*)
⟨*proof*⟩

lemma *corresXF-exec-fail*:
⟦ *corresXF st ret ex P A B*; ¬ *succeeds B s*; *P s* ⟧
⟹ ¬ *succeeds A* (*st s*)
⟨*proof*⟩

lemma *corresXF-intermediate*:
⟦ *corresXF st ret-xf ex-xf P A′ C*;
corresXF id (λ*r s*. *r*) (λ*r s*. *r*) (λ*s*. ∃ *x*. *s* = *st x* ∧ *P x*) *A A′* ⟧ ⟹
corresXF st ret-xf ex-xf P A C
⟨*proof*⟩

lemma *corresXF-join*:
⟦ *corresXF st V E P L L′*; ⋀*x y*. *corresXF st V′ E* (*P′ x y*) (*R x*) (*R′ y*);
⋀*s*. *Q s* ⟹ *L′* ⋆ *s* ?{ λ*r t*. *case r of Exn* - ⇒ ⊤ | *Result v* ⇒ *P′* (*V v t*) *v t* }?;
⋀*s*. *Q s* ⟹ *P s* ⟧ ⟹
corresXF st V′ E Q (*L* >>= *R*) (*L′* >>= *R′*)
⟨*proof*⟩

lemma *corresXF-join-xf-state-independent-same-state*:
⟦ *corresXF* (λ*s*. *s*) (λ*r s*. *V r*) (λ*r s*. *E r*) *P L L′*;
⋀*y*. *corresXF* (λ*s*. *s*) (λ*r s*. *V′ r*) (λ*r s*. *E r*) (*P′* (*V y*)) (*R* (*V y*)) (*R′ y*);
⋀*s*. *Q s* ⟹ *L* ⋆ *s* ?{ λ*r t*. *case r of Exn* - ⇒ ⊤ | *Result v* ⇒ *P′ v t*}?; ⋀*s*. *Q s*
⟹ *P s* ⟧ ⟹
corresXF (λ*s*. *s*) (λ*r s*. *V′ r*) (λ*r s*. *E r*) *Q* (*L* >>= *R*) (*L′* >>= *R′*)
⟨*proof*⟩

lemma *corresXF-except*:
⟦ *corresXF st V E P L L′*; ⋀*x y*. *corresXF st V E′* (*P′ x y*) (*R x*) (*R′ y*);
⋀*s*. *Q s* ⟹ *L′* ⋆ *s* ?{ λ*r s*. *case r of Exn r* ⇒ *P′* (*E r s*) *r s* | *Result* - ⇒ ⊤ }?;
⋀*s*. *Q s* ⟹ *P s* ⟧ ⟹
corresXF st V E′ Q (*L* <*catch*> *R*) (*L′* <*catch*> *R′*)
⟨*proof*⟩

lemma *corresXF-cond*:
⟦ *corresXF st V E P L L′*; *corresXF st V E P R R′*; ⋀*s*. *P s* ⟹ *A* (*st s*) = *A′ s*

$\rrbracket \Longrightarrow$
  corresXF st V E P (*condition A L R*) (*condition A′ L′ R′*)
 ⟨*proof*⟩

lemma *refines-assume-succeeds*: (*succeeds g t* $\Longrightarrow$ *refines f g s t R*) $\Longrightarrow$ *refines f g s t R*
 ⟨*proof*⟩

lemma *corresXF-while*:
 assumes *body-corres*: $\bigwedge x\ y.$ *corresXF st ret ex* ($\lambda s.$ *P x s* $\wedge$ *y* = *ret x s*) (*A y*) (*B x*)
 and *cond-match*: $\bigwedge s\ r.$ *P r s* $\Longrightarrow$ *C r s* = *C′* (*ret r s*) (*st s*)
 and *pred-inv*: $\bigwedge r\ s.$ *P r s* $\Longrightarrow$ *C r s* $\Longrightarrow$ *succeeds* (*whileLoop C′ A* (*ret r s*)) (*st s*) $\Longrightarrow$
 B r • *s ?*{⦇ $\lambda r\ s.$ *case r of Exn - ⇒ True | Result r ⇒ P r s* ⦈}
 and *init-match*: $\bigwedge s.$ *P′ x s* $\Longrightarrow$ *y* = *ret x s*
 and *pred-imply*: $\bigwedge s.$ *P′ x s* $\Longrightarrow$ *P x s*
shows *corresXF st ret ex* (*P′ x*) (*whileLoop C′ A y*) (*whileLoop C B x*)
 ⟨*proof*⟩

lemma *corresXF-name-pre*:
 ⟦ $\bigwedge s'.$ *corresXF st ret ex* ($\lambda s.$ *P s* $\wedge$ *s* = *s′*) *A C* ⟧ $\Longrightarrow$
  corresXF st ret ex P A C
 ⟨*proof*⟩

lemma *corresXF-guarded-while-body*:
 corresXF st ret ex P A B $\Longrightarrow$
  corresXF st ret ex P
  (*do*{ *r* ← *A*; - ← *guard* (*G r*); *return r* }) *B*
 ⟨*proof*⟩

lemma *whileLoop-succeeds-terminates-guard-body*:
 assumes *B-succeeds*: $\bigwedge i\ s.$ *succeeds* (*B i*) *s* $\Longrightarrow$ *succeeds* (*B′ i*) *s*
 assumes *B-reaches*: $\bigwedge i\ s\ r\ t.$ *reaches* (*B′ i*) *s r t* $\Longrightarrow$ *succeeds* (*B i*) *s* $\Longrightarrow$ *reaches* (*B i*) *s r t*
 assumes *termi*: *run* (*whileLoop C B I*) *s* $\neq \top$
 shows *run* (*whileLoop C B′ I*) *s* $\neq \top$
 ⟨*proof*⟩

lemma *whileLoop-succeeds-guard-body*:
 assumes *B-succeeds*: $\bigwedge i\ s.$ *succeeds* (*B i*) *s* $\Longrightarrow$ *succeeds* (*B′ i*) *s*
 assumes *B-reaches*: $\bigwedge i\ s\ r\ t.$ *reaches* (*B′ i*) *s r t* $\Longrightarrow$ *succeeds* (*B i*) *s* $\Longrightarrow$ *reaches* (*B i*) *s r t*
 assumes *termi*: *succeeds* (*whileLoop C B I*) *s*
 shows *succeeds* (*whileLoop C B′ I*) *s*
 ⟨*proof*⟩

lemma *corresXF-guarded-while*:

assumes *body-corres*: $\bigwedge x\ y.\ corresXF\ st\ ret\ ex\ (\lambda s.\ P\ x\ s \wedge y = ret\ x\ s)\ (A\ y)\ (B\ x)$
and *cond-match*: $\bigwedge s\ r.\ [\![P\ r\ s;\ G\ (ret\ r\ s)\ (st\ s)]\!] \Longrightarrow C\ r\ s = C'\ (ret\ r\ s)\ (st\ s)$
and *pred-inv*: $\bigwedge r\ s.\ P\ r\ s \Longrightarrow C\ r\ s \Longrightarrow succeeds\ (whileLoop\ C'\ A\ (ret\ r\ s))\ (st\ s) \Longrightarrow G\ (ret\ r\ s)\ (st\ s) \Longrightarrow B\ r \bullet s\ ?\{\!|\ \lambda r\ s.\ case\ r\ of\ Exn\ \text{-} \Rightarrow True \mid Result\ r \Rightarrow G\ (ret\ r\ s)\ (st\ s) \longrightarrow P\ r\ s\ |\!\}$
and *pred-imply*: $\bigwedge s.\ [\![G\ y\ (st\ s);\ P'\ x\ s]\!] \Longrightarrow P\ x\ s$
and *init-match*: $\bigwedge s.\ [\![G\ y\ (st\ s);\ P'\ x\ s]\!] \Longrightarrow y = ret\ x\ s$
shows *corresXF st ret ex* $(P'\ x)$

```
(do {
   - ← guard (G y);
   whileLoop C' (λi. (do {
      r ← A i;
      - ← guard (G r);
      return r
    })) y
 })
(whileLoop C B x)
```

⟨*proof*⟩

definition *ac-corres st check-termination AF* Γ *rx ex G* ≡

```
λA B. ∀ s. (G s ∧ succeeds A (st s)) ⟶
      (∀ t. Γ ⊢ ⟨B, Normal s⟩ ⇒ t ⟶
         (case t of
           Normal s' ⇒ reaches A (st s) (Result (rx s')) (st s')
         | Abrupt s' ⇒ reaches A (st s) (Exn (ex s')) (st s')
         | Fault e ⇒ e ∈ AF
         | - ⇒ False))
       ∧ (check-termination ⟶ Γ ⊢ B ↓ Normal s)
```

lemma *ccorresE-corresXF-merge*:

```
⟦ ccorresE st1 ct AF Γ ⊤ G1 M B;
  corresXF st2 rx ex G2 A M;
  ⋀s. st s = st2 (st1 s);
  ⋀r s. rx' s = rx r (st1 s);
  ⋀r s. ex' s = ex r (st1 s);
  ⋀s. G s ⟶ (s ∈ G1 ∧ G2 (st1 s)) ⟧ ⟹
 ac-corres st ct AF Γ rx' ex' G A B
```

⟨*proof*⟩

lemma *corresXF-corresXF-merge*:
⟦ *corresXF st rx ex P A B*; *corresXF st′ rx′ ex′ P′ B C* ⟧ ⟹
corresXF (*st o st′*) (λ*rv s*. *rx* (*rx′ rv s*) (*st′ s*))
(λ*rv s*. *ex* (*ex′ rv s*) (*st′ s*)) (λ*s*. *P′ s* ∧ *P* (*st′ s*)) *A C*
⟨*proof*⟩

lemma *ac-corres-guard-imp*:
⟦ *ac-corres st ct AF G rx ex P A C*; ⋀*s*. *P′ s* ⟹ *P s* ⟧ ⟹ *ac-corres st ct AF G rx ex P′ A C*
⟨*proof*⟩

lemma *corresXF-modify-local*:
⟦ ⋀*s*. *st s* = *st* (*M s*); ⋀*s*. *P s* ⟹ *ret* () (*M s*) = *x* ⟧
⟹ *corresXF st ret ex P* (*return x*) (*modify M*)
⟨*proof*⟩

lemma *corresXF-modify-global*:
⟦ ⋀*s*. *P s* ⟹ *M* (*st s*) = *st* (*M′ s*) ⟧ ⟹
corresXF st ret ex P (*modify M*) (*modify M′*)
⟨*proof*⟩

lemma *corresXF-select-modify*:
⟦ ⋀*s*. *P s* ⟹ *st s* = *st* (*M s*); ⋀*s*. *P s* ⟹ *ret* () (*M s*) ∈ *x* ⟧ ⟹
corresXF st ret ex P (*select x*) (*modify M*)
⟨*proof*⟩

lemma *corresXF-select-select*:
⟦ ⋀*s a*. *st s* = *st* (*M* (*a*::(*'a* ⇒ (*'a*::{*type*}))) *s*);
⋀*s x*. ⟦ *P s*; *x* ∈ *b*⟧ ⟹ *ret x s* ∈ *a* ⟧ ⟹
corresXF st ret ex P (*select a*) (*select b*)
⟨*proof*⟩

lemma *corresXF-modify-gets*:
⟦ ⋀*s*. *P s* ⟹ *st s* = *st* (*M s*); ⋀*s*. *P s* ⟹ *ret* () (*M s*) = *f* (*st* (*M s*)) ⟧ ⟹
corresXF st ret ex P (*gets f*) (*modify M*)
⟨*proof*⟩

lemma *corresXF-guard*:
⟦ ⋀*s*. *P s* ⟹ *G′ s* = *G* (*st s*) ⟧ ⟹ *corresXF st ret ex P* (*guard G*) (*guard G′*)
⟨*proof*⟩

lemma *corresXF-fail*:
corresXF st return-xf exception-xf P fail X
⟨*proof*⟩

lemma *corresXF-spec*:

⟦ ⋀s s'. ((s, s') ∈ A') = ((st s, st s') ∈ A); surj st ⟧
  ⟹ corresXF st ret ex P (state-select A) (state-select A')
⟨proof⟩

lemma *corresXF-throw*:
⟦ ⋀s. P s ⟹ E B s = A ⟧ ⟹ corresXF st V E P (throw A) (throw B)
⟨proof⟩

lemma *corresXF-append-gets-abs*:
assumes *corres*: corresXF st ret ex P L R
and *consistent*: ⋀s. P s ⟹ R • s ?{λr s. case r of Exn - ⇒ ⊤ | Result v ⇒ M (ret v s) (st s) = ret' v s }
shows corresXF st ret' ex P (L >>= (λr. gets (M r))) R
⟨proof⟩

lemma *corresXF-skipE*:
corresXF st ret ex P skip skip
⟨proof⟩

lemma *corresXF-id*:
corresXF id (λr s. r) (λr s. r) P M M
⟨proof⟩

lemma *corresXF-cong*:
⟦ ⋀s. st s = st' s;
⋀s r. ret-xf r s = ret-xf' r s;
⋀s r. ex-xf r s = ex-xf' r s;
⋀s. P s = P' s;
⋀s s'. P' s' ⟹ run A s = run A' s;
⋀s. P' s ⟹ run C s = run C' s ⟧ ⟹
corresXF st ret-xf ex-xf P A C = corresXF st' ret-xf' ex-xf' P' A' C'
⟨proof⟩

lemma *corresXF-exec-abs-select*:
⟦ x ∈ Q; x ∈ Q ⟹ corresXF id rx ex P (A x) A' ⟧ ⟹ corresXF id rx ex P (select Q >>= A) A'
⟨proof⟩

end

18.2 Hoare-Triples for L1 (internal use)

theory *L1Valid*
imports *L1Defs*
begin

definition

validE :: $('s \Rightarrow bool) \Rightarrow ('e, 'a, 's)$ *exn-monad* $\Rightarrow$
$('a \Rightarrow 's \Rightarrow bool) \Rightarrow$
$('e \Rightarrow 's \Rightarrow bool) \Rightarrow bool$

where *validE P f Q E* $\equiv \forall s.\ P\ s \longrightarrow f \cdot s\ ?\{\!|\ \lambda v\ s.\ case\ v\ of\ Result\ r \Rightarrow Q\ r\ s \mid Exn\ e \Rightarrow E\ e\ s\ |\!\}$

open-bundle *validE-syntax*
begin
notation *validE* (‹(‹*open-block notation*=‹*mixfix L1 Hoare triple*››{|-|}/ - /({|-|},/ {|-|}))›)
end

lemma *hoareE-TrueI*: {|*P*|} *f* {|λ- -. *True*|}, {|λ*r* -. *True*|}
⟨*proof*⟩

lemma *combine-validE*: ⟦ {| *P* |} *x* {| *Q* |},{| *E* |};
{| *P'* |} *x* {| *Q'* |},{| *E'* |} ⟧ ⟹
{| *P and P'* |} *x* {| λ*r*. (*Q r*) *and* (*Q' r*) |},{|λ*r*. (*E r*) *and* (*E' r*) |}
⟨*proof*⟩

lemma *L1-skip-wp*: {| *P* () |} *L1-skip* {| *P* |}, {| *Q* |}
⟨*proof*⟩

lemma *L1-modify-wp*: {| λ*s*. *P* () (*f s*) |} *L1-modify f* {| *P* |}, {| *Q* |}
⟨*proof*⟩

lemma *L1-spec-wp*: {| λ*s*. ∀*t*. (*s*, *t*) ∈ *f* ⟶ *P* () *t* |} *L1-spec f* {| *P* |}, {| *Q* |}
⟨*proof*⟩

lemma *L1-assume-wp*: {| λ*s*. ∀*t*. ((), *t*) ∈ *f s* ⟶ *P* () *t* |} *L1-assume f* {| *P* |}, {| *Q* |}
⟨*proof*⟩

lemma *L1-init-wp*: {| λ*s*. ∀*x*. *P* () (*f* (λ-. *x*) *s*) |} *L1-init f* {| *P* |}, {| *Q* |}
⟨*proof*⟩

lemma *L1-skip-lp*: ⟦ ⋀*s*. *P s* ⟹ *Q* () *s* ⟧ ⟹ {|*P*|} *L1-skip* {|*Q*|}, {|*E*|}
⟨*proof*⟩

lemma *L1-skip-lp-same-pre-post*: {|*P*|} *L1-skip* {|λ-. *P*|}, {|λ-. *P*|}
⟨*proof*⟩

lemma *L1-guard-lp*: ⟦ ⋀*s*. *P s* ⟹ *Q* () *s* ⟧ ⟹ {|*P*|} *L1-guard e* {|*Q*|}, {|*E*|}

⟨*proof*⟩

lemma *L1-guard-lp-same-pre-post*: ⦃*P*⦄ *L1-guard e* ⦃λ-. *P*⦄, ⦃λ-. *P*⦄
⟨*proof*⟩

lemma *L1-guarded-lp-same-pre-post*: ⦃*P*⦄ *c* ⦃λ-. *P*⦄, ⦃λ-. *P*⦄
⟹ ⦃*P*⦄ *L1-guarded g c* ⦃λ-. *P*⦄, ⦃λ-. *P*⦄
⟨*proof*⟩

lemma *L1-guarded-lp-gets*: (⋀*p*. ⦃*P*⦄ (*c p*) ⦃λ-. *P*⦄, ⦃λ-. *P*⦄)
⟹ ⦃*P*⦄ *L1-guarded g* (*gets dest* ≫= (λ*p*. *c p*)) ⦃λ-. *P*⦄, ⦃λ-. *P*⦄
⟨*proof*⟩

lemma *L1-fail-lp*: ⦃*P*⦄ *L1-fail* ⦃*Q*⦄, ⦃*E*⦄
⟨*proof*⟩

lemma *L1-fail-lp-same-pre-post*: ⦃*P*⦄ *L1-fail* ⦃λ-. *P*⦄, ⦃λ-. *P*⦄
⟨*proof*⟩

lemma *L1-throw-lp*: ⟦ ⋀*s*. *P s* ⟹ *E* () *s* ⟧ ⟹ ⦃*P*⦄ *L1-throw* ⦃*Q*⦄, ⦃*E*⦄
⟨*proof*⟩

lemma *L1-throw-lp-same-pre-post*: ⦃*P*⦄ *L1-throw* ⦃λ-. *P*⦄, ⦃λ-. *P*⦄
⟨*proof*⟩

lemma *L1-spec-lp*: ⟦ ⋀*s r*. ⟦ (*s*, *r*) ∈ *e*; *P s* ⟧ ⟹ *Q* () *r* ⟧ ⟹ ⦃*P*⦄ *L1-spec e*
⦃*Q*⦄, ⦃*E*⦄
⟨*proof*⟩

lemma *L1-spec-lp-same-pre-post*: ⟦ ⋀*s r*. ⟦ (*s*, *r*) ∈ *e*; *P s* ⟧ ⟹ *P r* ⟧
⟹ ⦃*P*⦄ *L1-spec e* ⦃λ-. *P*⦄, ⦃λ-. *P*⦄
⟨*proof*⟩

lemma *L1-modify-lp*: ⟦ ⋀*s*. *P s* ⟹ *Q* () (*f s*) ⟧ ⟹ ⦃*P*⦄ *L1-modify f* ⦃*Q*⦄, ⦃*E*⦄
⟨*proof*⟩

lemma *L1-modify-lp-same-pre-post*: ⟦ ⋀*s*. *P s* ⟹ *P* (*f s*) ⟧ ⟹ ⦃*P*⦄ *L1-modify f*
⦃λ-. *P*⦄, ⦃λ-. *P*⦄
⟨*proof*⟩

lemma *L1-call-lp*:
⟦ ⋀*s r*. *P s* ⟹ *Q* () (*return-xf r* (*scope-teardown s r*));
⋀*s r*. *P s* ⟹ *E* () (*result-exn* (*scope-teardown s r*) *r*)⟧ ⟹

$\{\!|P|\!\}$ *L1-call scope-setup dest-fn scope-teardown result-exn return-xf* $\{\!|Q|\!\}$, $\{\!|E|\!\}$
⟨*proof*⟩

lemma *L1-call-lp-same-pre-post*:
⟦ $\bigwedge s\ r.\ P\ s \Longrightarrow P\ (\mathit{return\text{-}xf}\ r\ (\mathit{scope\text{-}teardown}\ s\ r))$;
$\bigwedge s\ r.\ P\ s \Longrightarrow P\ (\mathit{result\text{-}exn}\ (\mathit{scope\text{-}teardown}\ s\ r)\ r)$⟧ ⟹
$\{\!|P|\!\}$ *L1-call scope-setup dest-fn scope-teardown result-exn return-xf* $\{\!|\lambda\text{-}.\ P|\!\}$, $\{\!|\lambda\text{-}.\ P|\!\}$
⟨*proof*⟩

lemma *L1-seq-lp*: ⟦
$\{\!|P1|\!\}$ *A* $\{\!|Q1|\!\}$,$\{\!|E1|\!\}$;
$\{\!|P2|\!\}$ *B* $\{\!|Q2|\!\}$,$\{\!|E2|\!\}$;
$\bigwedge s.\ P\ s \Longrightarrow P1\ s$;
$\bigwedge s.\ Q1\ ()\ s \Longrightarrow P2\ s$;
$\bigwedge s.\ Q2\ ()\ s \Longrightarrow Q\ ()\ s$;
$\bigwedge s.\ E1\ ()\ s \Longrightarrow E\ ()\ s$;
$\bigwedge s.\ E2\ ()\ s \Longrightarrow E\ ()\ s$
⟧ ⟹ $\{\!|P|\!\}$ *L1-seq A B* $\{\!|Q|\!\}$, $\{\!|E|\!\}$
⟨*proof*⟩

lemma *L1-seq-lp-same-pre-post*: ⟦
$\{\!|P|\!\}$ *A* $\{\!|\lambda\text{-}.\ P|\!\}$,$\{\!|\lambda\text{-}.\ P|\!\}$;
$\{\!|P|\!\}$ *B* $\{\!|\lambda\text{-}.\ P|\!\}$,$\{\!|\lambda\text{-}.\ P|\!\}$
⟧ ⟹ $\{\!|P|\!\}$ *L1-seq A B* $\{\!|\lambda\text{-}.\ P|\!\}$, $\{\!|\lambda\text{-}.\ P|\!\}$
⟨*proof*⟩

lemma *L1-condition-lp*:
⟦ $\{\!|P1|\!\}$ *A* $\{\!|Q1|\!\}$,$\{\!|E1|\!\}$;
$\{\!|P2|\!\}$ *B* $\{\!|Q2|\!\}$,$\{\!|E2|\!\}$;
$\bigwedge s.\ P\ s \Longrightarrow P1\ s$;
$\bigwedge s.\ P\ s \Longrightarrow P2\ s$;
$\bigwedge s.\ Q1\ ()\ s \Longrightarrow Q\ ()\ s$;
$\bigwedge s.\ Q2\ ()\ s \Longrightarrow Q\ ()\ s$;
$\bigwedge s.\ E1\ ()\ s \Longrightarrow E\ ()\ s$;
$\bigwedge s.\ E2\ ()\ s \Longrightarrow E\ ()\ s$ ⟧ ⟹
$\{\!|P|\!\}$ *L1-condition c A B* $\{\!|Q|\!\}$, $\{\!|E|\!\}$
⟨*proof*⟩

lemma *L1-condition-lp-same-pre-post*:
⟦$\{\!|P|\!\}$ *A* $\{\!|\lambda\text{-}.\ P|\!\}$,$\{\!|\lambda\text{-}.\ P|\!\}$;
$\{\!|P|\!\}$ *B* $\{\!|\lambda\text{-}.\ P|\!\}$,$\{\!|\lambda\text{-}.\ P|\!\}$
⟧ ⟹
$\{\!|P|\!\}$ *L1-condition c A B* $\{\!|\lambda\text{-}.\ P|\!\}$, $\{\!|\lambda\text{-}.\ P|\!\}$
⟨*proof*⟩

lemma *L1-catch-lp*:
⟦ $\{\!|P1|\!\}$ *A* $\{\!|Q1|\!\}$,$\{\!|E1|\!\}$;

$\{\!|P2|\!\}\ B\ \{\!|Q2|\!\},\{\!|E2|\!\};$
$\bigwedge s.\ P\ s \Longrightarrow P1\ s;$
$\bigwedge s.\ E1\ ()\ s \Longrightarrow P2\ s;$
$\bigwedge s.\ Q1\ ()\ s \Longrightarrow Q\ ()\ s;$
$\bigwedge s.\ Q2\ ()\ s \Longrightarrow Q\ ()\ s;$
$\bigwedge s.\ E2\ ()\ s \Longrightarrow E\ ()\ s\ ]\!] \Longrightarrow$
$\{\!|P|\!\}\ L1\text{-}catch\ A\ B\ \{\!|Q|\!\},\ \{\!|E|\!\}$
⟨*proof*⟩

lemma *L1-catch-lp-same-pre-post*:
$[\![\{\!|P|\!\}\ A\ \{\!|\lambda\text{-}.\ P|\!\},\{\!|\lambda\text{-}.\ P|\!\};$
$\{\!|P|\!\}\ B\ \{\!|\lambda\text{-}.\ P|\!\},\{\!|\lambda\text{-}.\ P|\!\}$
$]\!] \Longrightarrow$
$\{\!|P|\!\}\ L1\text{-}catch\ A\ B\ \{\!|\lambda\text{-}.\ P|\!\},\ \{\!|\lambda\text{-}.\ P|\!\}$
⟨*proof*⟩

lemma *L1-init-lp*: $[\![\ \bigwedge s.\ P\ s \Longrightarrow \forall x.\ Q\ ()\ (f\ (\lambda\text{-}.\ x)\ s)\ ]\!] \Longrightarrow \{\!|P|\!\}\ L1\text{-}init\ f\ \{\!|Q|\!\},$
$\{\!|E|\!\}$
⟨*proof*⟩

lemma *L1-init-lp-same-pre-post*: $[\![\ \bigwedge s.\ P\ s \Longrightarrow \forall x.\ P\ (f\ (\lambda\text{-}.\ x)\ s)\ ]\!] \Longrightarrow \{\!|P|\!\}$
$L1\text{-}init\ f\ \{\!|\lambda\text{-}.\ P|\!\},\ \{\!|\lambda\text{-}.\ P|\!\}$
⟨*proof*⟩

lemma *validE-weaken*:
$[\![\ \{\!|P'|\!\}\ A\ \{\!|Q'|\!\},\{\!|E'|\!\};\ \bigwedge s.\ P\ s \Longrightarrow P'\ s;\ \bigwedge r\ s.\ Q'\ r\ s \Longrightarrow Q\ r\ s;\ \bigwedge r\ s.\ E'\ r\ s \Longrightarrow$
$E\ r\ s\ ]\!] \Longrightarrow \{\!|P|\!\}\ A\ \{\!|Q|\!\},\{\!|E|\!\}$
⟨*proof*⟩

lemma *L1-while-lp*:
assumes *body-lp*: $\{\!|\ P'\ |\!\}\ B\ \{\!|\ Q'\ |\!\},\ \{\!|\ E'\ |\!\}$
and *p-impl*: $\bigwedge s.\ P\ s \Longrightarrow P'\ s$
and *q-impl*: $\bigwedge s.\ Q'\ ()\ s \Longrightarrow Q\ ()\ s$
and *e-impl*: $\bigwedge s.\ E'\ ()\ s \Longrightarrow E\ ()\ s$
and *inv*: $\bigwedge s.\ Q'\ ()\ s \Longrightarrow P'\ s$
and *inv'*: $\bigwedge s.\ P'\ s \Longrightarrow Q'\ ()\ s$
shows $\{\!|\ P\ |\!\}\ L1\text{-}while\ c\ B\ \{\!|\ Q\ |\!\},\{\!|\ E\ |\!\}$
⟨*proof*⟩

lemma *L1-while-lp-same-pre-post*:
assumes *body-lp*: $\{\!|\ P\ |\!\}\ B\ \{\!|\ \lambda\text{-}.\ P|\!\},\ \{\!|\ \lambda\text{-}.\ P\ |\!\}$
shows $\{\!|\ P\ |\!\}\ L1\text{-}while\ c\ B\ \{\!|\lambda\text{-}.\ P\ |\!\},\{\!|\ \lambda\text{-}.\ P\ |\!\}$
⟨*proof*⟩

lemma *on-exit-lp-same-pre-post*:
assumes *cleanup*: $\bigwedge s\ t.\ (s,t) \in cleanup \Longrightarrow\ P\ t = P\ s$
assumes *c*: $\{\!|P|\!\}\ c\ \{\!|\lambda\text{-}.\ P|\!\},\ \{\!|\lambda\text{-}.\ P|\!\}$

shows $\{\!|P|\!\}$ *on-exit c cleanup* $\{\!|\lambda\text{-}.\ P|\!\}$, $\{\!|\lambda\text{-}.\ P|\!\}$
⟨*proof*⟩

lemma *seqE*:
assumes *f-valid*: $\{\!|A|\!\}\ f\ \{\!|B|\!\}$,$\{\!|E|\!\}$
assumes *g-valid*: $\bigwedge x.\ \{\!|B\ x|\!\}\ g\ x\ \{\!|C|\!\}$,$\{\!|E|\!\}$
shows $\{\!|A|\!\}$ *do* { $x \leftarrow f$; $g\ x$ } $\{\!|C|\!\}$,$\{\!|E|\!\}$
⟨*proof*⟩

named-theorems *with-fresh-stack-ptr-lp-same-pre-post*
context *stack-heap-state*
begin
lemma *with-fresh-stack-ptr-lp-same-pre-post*[*with-fresh-stack-ptr-lp-same-pre-post*]:

assumes *c*: $\bigwedge p.\ \{\!|P|\!\}\ (c\ p)\ \{\!|\lambda\text{-}.\ P|\!\}$, $\{\!|\lambda\text{-}.\ P|\!\}$
assumes *htd-indep*: $\bigwedge s\ g.\ P\ (\textit{htd-upd}\ g\ s) = P\ s$
assumes *hmem-indep*: $\bigwedge s\ f.\ P\ (\textit{hmem-upd}\ f\ s) = P\ s$
shows $\{\!|P|\!\}$ *with-fresh-stack-ptr n g c* $\{\!|\lambda\text{-}.\ P|\!\}$, $\{\!|\lambda\text{-}.\ P|\!\}$
⟨*proof*⟩
end

lemma *validE-weaken-dependent*:
assumes *dep-spec*: $\bigwedge s.\ P\ s \Longrightarrow \{\!|P'\ s|\!\}\ A\ \{\!|Q'\ s|\!\}$, $\{\!|E'\ s|\!\}$
assumes *weaken-pre*: $\bigwedge s.\ P\ s \Longrightarrow P'\ s\ s$
assumes *weaken-norm*: $(\bigwedge s\ r\ t.\ P'\ s\ s \Longrightarrow Q'\ s\ r\ t \Longrightarrow Q\ r\ t)$
assumes *weaken-exn*: $(\bigwedge s\ r\ t.\ P'\ s\ s \Longrightarrow E'\ s\ r\ t \Longrightarrow E\ r\ t)$
shows $\{\!|P|\!\}\ A\ \{\!|Q|\!\}$, $\{\!|E|\!\}$
⟨*proof*⟩

lemma *validE-weaken-dependent-same*:
assumes *dep-spec*: $\bigwedge s.\ P\ s \Longrightarrow \{\!|P'\ s|\!\}\ A\ \{\!|\lambda\text{-}.\ P'\ s|\!\}$, $\{\!|\lambda\text{-}.\ P'\ s|\!\}$
assumes *weaken-post*: $(\bigwedge s\ t.\ P'\ s\ t \Longrightarrow P\ t)$
assumes *weaken-pre*: $\bigwedge s.\ P\ s \Longrightarrow P'\ s\ s$
shows $\{\!|P|\!\}\ A\ \{\!|\lambda\text{-}.\ P|\!\}$, $\{\!|\lambda\text{-}.\ P|\!\}$
⟨*proof*⟩

end

Chapter 19

L2 phase: local variable abstraction with lambdas

theory *L2Defs*
imports *CorresXF L1Defs L1Peephole L1Valid*
begin

type-synonym $('s, 'a, 'e)$ *L2-monad* $= ('e, 'a, 's)$ *exn-monad*

definition *L2-unknown* $(ns :: nat\ list) \equiv$ *select UNIV* $:: ('s, 'a, 'e)$ *L2-monad*
definition *L2-seq* $(A :: ('s, 'a, 'e)$ *L2-monad*$) (B :: 'a \Rightarrow ('s, 'b, 'e)$ *L2-monad*$) \equiv (A >>= B) :: ('s, 'b, 'e)$ *L2-monad*
definition *L2-modify* $m \equiv$*modify* $m :: ('s,$ *unit*$, 'e)$ *L2-monad*
definition *L2-gets* f $(names :: nat\ list) \equiv$ *gets* $f :: ('s, 'a, 'e)$ *L2-monad*
definition *L2-condition* c $(A :: ('s, 'a, 'e)$ *L2-monad*$) (B :: ('s, 'a, 'e)$ *L2-monad*$)$ $\equiv$ *condition* c A B
definition *L2-catch* $(A :: ('s, 'a, 'e)$ *L2-monad*$) (B :: 'e \Rightarrow ('s, 'a, 'ee)$ *L2-monad*$)$ $\equiv (A <catch> B)$
definition *L2-try* $(A :: ('s, 'a, 'e + 'a)$ *L2-monad*$) \equiv$ *try* A
definition *L2-while* c $(A :: 'a \Rightarrow ('s, 'a, 'e)$ *L2-monad*$)$ i $(ns :: nat\ list) \equiv$ *whileLoop* c A $i :: ('s, 'a, 'e)$ *L2-monad*
definition *L2-throw* x $(ns :: nat\ list) \equiv$ *throw* $x :: ('s, 'a, 'e)$ *L2-monad*
definition *L2-spec* $r \equiv$ (*state-select* $r >>= (\lambda$-. *select UNIV*$)) :: ('s, 'a, 'e)$ *L2-monad*
definition *L2-assume* $r \equiv$ (*assume-result-and-state* r) $:: ('s, 'a, 'e)$ *L2-monad*
definition *L2-guard* $c \equiv$ (*guard* c) $:: ('s,$ *unit*$, 'e)$ *L2-monad*
definition *L2-guarded* P $c \equiv$ *L2-seq* (*L2-guard* P) $(\lambda$-. c) — used to guard a function-pointer call
definition *L2-fail* $\equiv$ *fail* $:: ('s, 'a, 'e)$ *L2-monad*
definition *L2-call* x *emb* $(ns :: nat\ list) \equiv$ *map-value* (*map-exn emb*) $x :: ('s, 'a, 'e)$ *L2-monad*

abbreviation *L2-skip* $\equiv$ *L2-gets* $(\lambda$-. $())$ []
definition *L2-VARS* f $(names::nat\ list) \equiv f$ — Auxilliary construct to preserve names, like in *L2-unknown*, *L2-gets*, ...

definition *gets-theE* :: (*'s* ⇒ *'a option*) ⇒ (*'e*, *'a*, *'s*) *exn-monad* **where**
gets-theE ≡ λ*x*. *gets-the x* — Lifting into exception monad

definition *L2-folded-gets f names* ≡ *L2-gets f names* :: (*'s*, *'a*, *'e*) *L2-monad*
definition *L2-voidcall x emb ns* ≡ *L2-seq* (*L2-call x emb ns*) (λ*ret*. *L2-skip*) :: (*'s*, *unit*, *'e*) *L2-monad*
definition *L2-modifycall x m emb ns* ≡ *L2-seq* (*L2-call x emb ns*) (λ*ret*. *L2-modify* (*m ret*)) :: (*'s*, *unit*, *'e*) *L2-monad*
definition *L2-returncall x f emb ns* ≡ *L2-seq* (*L2-call x emb ns*) (λ*ret*. *L2-folded-gets* (*f ret*) []) :: (*'s*, *'a*, *'e*) *L2-monad*

definition *L2-seq-guard P A* ≡ *L2-seq* (*L2-guard P*) *A*
definition *L2-seq-gets c n A* ≡ *L2-seq* (*L2-gets* (λ-. *c*) *n*) *A*
definition *L2-seq-unknown ns A* ≡ *L2-seq* (*L2-unknown ns*) *A*
definition *L2-seq-condition c L R X* ≡ *L2-seq* (*L2-condition c L R*) *X*

definition *SANITIZE-NAMES prj ns ns'* = *True*

lemma *sanitize-namesI*: *SANITIZE-NAMES prj ns ns'*
⟨*proof*⟩

definition *DYN-CALL* :: *prop* ⇒ *prop* **where** *PROP DYN-CALL* (*PROP P*) ≡ *PROP P*

lemma *DYN-CALL-I*: *PROP P* ⟹ *PROP DYN-CALL* (*PROP P*)
⟨*proof*⟩

lemma *DYN-CALL-D*: *PROP DYN-CALL* (*PROP P*) ⟹ *PROP P*
⟨*proof*⟩

lemma *runs-to-partial-top*[*corres-top*]: ⊤ ∙ *s* ?{‖ *Q* ‖}
⟨*proof*⟩

lemma *admissible-runs-to-partial*[*corres-admissible*]: *ccpo.admissible Inf* (≥) (λ*f* . *f* ∙ *s* ?{‖*Q*‖})
⟨*proof*⟩

lemma *reaches-gets-theE* [*simp*]:
reaches (*gets-theE M*) *s rv s'* ⟷ (∃ *v'*. *rv* = *Result v'* ∧ *s'* = *s* ∧ *M s* = *Some v'*)
⟨*proof*⟩

lemma *seucceeds-gets-theE* [*simp*]:
succeeds (*gets-theE M*) *s* = (∃ *v*. *M s* = *Some v*)

⟨*proof*⟩

lemma *L2-folded-gets-bind*:
L2-seq (*L2-folded-gets* (λ-. *x*) *ns*) *f* = *f x*
⟨*proof*⟩

lemma *L2-call-throw-names*: *L2-call x emb ns* = (*x* <*catch*> (λ*r*. *L2-throw* (*emb r*) *ns*))
⟨*proof*⟩

lemmas *L2-remove-scaffolding-1* =
L2-folded-gets-bind [*THEN eq-reflection*]
L2-returncall-def
L2-modifycall-def
L2-voidcall-def

lemmas *L2-remove-scaffolding-2* =
L2-remove-scaffolding-1
L2-folded-gets-def

abbreviation (*input*) *L2-guarded-while G C B i n* ≡ *L2-seq* (*L2-guard* (*G i*))
(λ-. *L2-while C* (λ*i*. *L2-seq* (*B i*) (λ*r*. *L2-seq* (*L2-guard* (*G r*)) (λ-. *L2-gets* (λ-. *r*) *n*))) *i n*)

lemmas *L2-defs* = *L2-unknown-def L2-seq-def*
L2-modify-def L2-gets-def L2-condition-def L2-catch-def L2-while-def
L2-throw-def L2-spec-def L2-assume-def L2-guard-def L2-fail-def L2-folded-gets-def
L2-voidcall-def L2-modifycall-def L2-returncall-def
L2-try-def

lemma *L2-defs'*:
L2-unknown n ≡ *unknown*
L2-seq A' B' ≡ *A'* >>= *B'*
L2-modify m ≡ *modify m*
L2-gets f n ≡ *gets f*
L2-condition c L R ≡ *condition c L R*
L2-catch A B ≡ (*A* <*catch*> *B*)
L2-try A ≡ *try A*
L2-while c' B'' i n ≡ *whileLoop c' B'' i*
L2-throw x n ≡ *throw x*
L2-spec r ≡ (*state-select r* >>= (λ-. *select UNIV*))
L2-assume r' ≡ (*assume-result-and-state r'*)
L2-guard c ≡ *guard c*
L2-fail ≡ *fail*
⟨*proof*⟩

definition
L2corres :: $('s \Rightarrow 't) \Rightarrow ('s \Rightarrow 'r) \Rightarrow ('s \Rightarrow 'e) \Rightarrow ('s \Rightarrow bool)$
$\Rightarrow ('e, 'r, 't)\ exn\text{-}monad \Rightarrow (unit, unit, 's)\ exn\text{-}monad \Rightarrow bool$
where
L2corres st ret-xf ex-xf P A C
$\equiv$ *corresXF st* (λ-. *ret-xf*) (λ-. *ex-xf*) *P A C*

lemma *admissible-nondet-ord-L2corres* [*corres-admissible*]:
ccpo.admissible Inf ($\geq$) (λA. *L2corres st ret-xf ex-xf P A C*)
⟨*proof*⟩

lemma *L2corres-top* [*corres-top*]: *L2corres st ret-xf ex-xf P* $\top$ *C*
⟨*proof*⟩

definition
L2-call-L1 arg-xf gs ret-xf l1body
= *do* {
s $\leftarrow$ *get-state*;
t $\leftarrow$ *select* {*t*. *gs t* = *s* $\wedge$ *arg-xf t*};
run-bind l1body t ($\lambda r'\ t'$.
(*case r' of*
Exception - $\Rightarrow$ *fail*
| *Result* - $\Rightarrow$ *do* {*set-state* (*gs t'*); *return* (*ret-xf t'*)}))
}

lemma *L2corres-L2-call-L1*:
L2corres gs ret-xf ex-xf arg-xf
(*L2-call-L1 arg-xf gs ret-xf f*) *f*
⟨*proof*⟩

lemma *L2corres-L2-call-simpl*:
l1-f $\equiv$ *simpl-f* $\Longrightarrow$
L2corres gs ret-xf ex-xf arg-xf
(*L2-call-L1 arg-xf gs ret-xf simpl-f*) *l1-f*
⟨*proof*⟩

lemma *L2corres-modify-global*:
⟦ $\bigwedge s$. *P s* $\Longrightarrow$ *M* (*st s*) = *st* (*M' s*) ⟧ $\Longrightarrow$
L2corres st ret ex P (*L2-modify M*) (*L1-modify M'*)
⟨*proof*⟩

lemma *L2corres-modify-unknown*:
⟦ ⋀*s*. *P s* ⟹ *st s* = *st* (*M s*) ⟧ ⟹
L2corres st ret ex P (*L2-unknown ns*) (*L1-modify M*)
⟨*proof*⟩

lemma *L2corres-spec-unknown*:
⟦ ⋀*s a*. *st s* = *st* (*M* (*a*::(*'a* ⇒ (*'a*::{*type*}))) *s*) ⟧ ⟹
L2corres st ret ex P (*L2-unknown ns*) (*L1-init M*)
⟨*proof*⟩

lemma *L2corres-modify-gets*:
⟦ ⋀*s*. *P s* ⟹ *st s* = *st* (*M s*); ⋀*s*. *P s* ⟹ *ret* (*M s*) = *f* (*st s*) ⟧ ⟹
L2corres st ret ex P (*L2-gets* (λ*s*. *f s*) *n*) (*L1-modify M*)
⟨*proof*⟩

lemma *L2corres-gets-skip*:
L2corres st ret ex P L2-skip L1-skip
⟨*proof*⟩

lemma *L2corres-guard*:
⟦ ⋀*s*. *P s* ⟹ *G'* *s* = *G* (*st s*) ⟧ ⟹
L2corres st return-xf exception-xf P (*L2-guard G*) (*L1-guard G'*)
⟨*proof*⟩

lemma *L2corres-fail*:
L2corres st return-xf exception-xf P L2-fail X
⟨*proof*⟩

lemma *L2corres-spec*:
⟦ ⋀*s s'*. ((*s*, *s'*) ∈ *A'*) = ((*st s*, *st s'*) ∈ *A*); *surj st* ⟧
⟹ *L2corres st return-xf exception-xf P* (*L2-spec A*) (*L1-spec A'*)
⟨*proof*⟩

lemma *L2corres-assume*:
⟦ ⋀*s s'*. ((), *s'*) ∈ *A' s*) = ((), *st s'*) ∈ *A* (*st s*)) ⟧
⟹ *L2corres st return-xf exception-xf P* (*L2-assume A*) (*L1-assume A'*)
⟨*proof*⟩

lemma *L2corres-seq*:
⟦ *L2corres st return-xf exception-xf P A A'*;
⋀*x*. *L2corres st return-xf' exception-xf* (*P'* *x*) (*B x*) *B'*;
{|*R*|} *A'* {|λ- *s*. *P'* (*return-xf s*) *s*|}, {|λ- -. *True*|};
⋀*s*. *R s* ⟹ *P s* ⟧ ⟹
L2corres st return-xf' exception-xf R (*L2-seq A B*) (*L1-seq A' B'*)
⟨*proof*⟩

lemma *L2corres-guard-imp*:
⟦ *L2corres st ret-state ex-state Q M M'*; *pred-imp P Q* ⟧ ⟹
L2corres st ret-state ex-state P M M'
⟨*proof*⟩

lemma *L2corres-guarded''*:
assumes *bdy*: ⋀*s' s*. *g' s'* ⟹ *s* = *st s'* ⟹ *P s'* ⟹ *L2corres st ret ex P* (*c* (*dest s*)) (*c'* (*dest' s'*))
assumes *g-g'*: ⋀*s'*. *P s'* ⟹ *g' s'* = *g* (*st s'*)
assumes *dest*: ⋀*s' s*. *g' s'* ⟹ *s* = *st s'* ⟹ *P s'* ⟹ *dest' s'* = *dest* (*st s'*)
shows *L2corres st ret ex P* (*L2-guarded g* (*L2-seq* (*L2-gets dest* []) *c*)) (*L1-guarded g'* (*gets dest'* ≫= *c'*))
⟨*proof*⟩

lemma *L2corres-catch*:
⟦ *L2corres st V E P L L'*;
⋀*x*. *L2corres st V E'* (*P' x*) (*R x*) *R'*;
{|*Q*|} *L'* {|λ- -. *True*|}, {|λ- *s*. *P'* (*E s*) *s*|}; ⋀*s*. *Q s* ⟹ *P s* ⟧ ⟹
L2corres st V E' Q (*L2-catch L R*) (*L1-catch L' R'*)
⟨*proof*⟩

lemma *L2corres-throw*:
⟦ ⋀*s*. *P s* ⟹ *E s* = *A* ⟧ ⟹ *L2corres st V E P* (*L2-throw A n*) (*L1-throw*)
⟨*proof*⟩

lemma *L2corres-conseq*:
assumes *corres*: *L2corres st return-xf exception-xf P C C'*
assumes *conseq*: ⋀*s*. *Q s* ⟹ *P s*
shows *L2corres st return-xf exception-xf Q C C'*
⟨*proof*⟩

lemma *L2corres-cond*:
⟦ *L2corres st return-xf exception-xf P A A'*;
L2corres st return-xf exception-xf P' B B';
⋀*s*. *R s* ⟹ *P s*;
⋀*s*. *R s* ⟹ *P' s*;
⋀*s*. *R s* ⟹ *c' s* = *c* (*st s*) ⟧ ⟹
L2corres st return-xf exception-xf R (*L2-condition c A B*) (*L1-condition c' A' B'*)
⟨*proof*⟩

lemma *L2corres-inject-return*:
⟦ *L2corres st V E P L R*; {|*P'*|} *R* {|λ- *s*. *W* (*V s*) = *V' s*|}, {| λ- -. *True* |}; ⋀*s*. *P' s* ⟹ *P s*⟧ ⟹
L2corres st V' E P' (*L2-seq L* (λ*x*. *L2-gets* (λ-. *W x*) *n*)) *R*
⟨*proof*⟩

lemma *L2corres-inject-return'*:
⟦ *L2corres st V E P L R*;*Gets* = (λ*x*. *L2-gets* (λ-. *W x*) *n*);{|*P'*|} *R* {|λ- *s*. *W* (*V s*) = *V' s*|}, {| λ- -. *True*|}; ⋀*s*. *P' s* ⟹ *P s*
⟧ ⟹
L2corres st V' E P' (*L2-seq L Gets*) *R*
⟨*proof*⟩

lemma *L2corres-skip*:
L2corres st return-xf exception-xf P L2-skip L1-skip
⟨*proof*⟩

lemma *L2corres-while*:
assumes *body-corres*: ⋀*x*. *L2corres st ret ex* (*P' x*) (*A x*) *B*
and *inv-holds*: {|λ*s*. *P* (*ret s*) *s* |} *B* {|λ- *s*. *P* (*ret s*) *s* |}, {|λ- -. *True*|}
and *cond-match*: ⋀*s*. *P* (*ret s*) *s* ⟹ *c' s* = *c* (*ret s*) (*st s*)
and *pred-imply*: ⋀*s x*. *P x s* ⟹ *P' x s*
and *pred-extract*: ⋀*s*. *P x s* ⟹ *ret s* = *x*
and *pred-imply2*: ⋀*s*. *Q x s* ⟹ *P x s*
shows *L2corres st ret ex* (*Q x*) (*L2-while c A x n*) (*L1-while c' B*)
⟨*proof*⟩

lemma *L2corres-returncall*:
⟦ *L2corres st ret' ex' P' Z dest-fn*;
⋀*s*. *P s* ⟹ *P'* (*scope-setup s*);
⋀*t s*. *st* (*return-xf t* (*scope-teardown s t*)) = *st t*;
⋀*t s*. *st* (*result-exn* (*scope-teardown s t*) *t*) = *st t*;
⋀*s*. *st* (*scope-setup s*) = *st s*;
⋀*t s*. *st* (*scope-teardown s t*) = *st t*;
⋀*t s*. *P s* ⟹ *ret* (*return-xf t* (*scope-teardown s t*)) = *F* (*ret' t*) (*st t*) ;
⋀*t s*. *P s* ⟹ (*ex* (*result-exn* (*scope-teardown s t*) *t*)) = *emb* (*ex' t*) ⟧ ⟹
L2corres st ret ex P (*L2-returncall Z F emb ns*)
(*L1-call scope-setup dest-fn scope-teardown result-exn return-xf*)
⟨*proof*⟩

lemma *L2corres-dest-fn-simp*:
assumes *dest-fn*: ⋀*s*. *P s* ⟹ *dest-fn* = *dest-fn'*
assumes *corres*: *L2corres st ret ex P X*
(*L1-call scope-setup* (*measure-call dest-fn'*) *scope-teardown result-exn return-xf*)
shows *L2corres st ret ex P X*
(*L1-call scope-setup* (*measure-call dest-fn*) *scope-teardown result-exn return-xf*)
⟨*proof*⟩

lemma *L2corres-l2-propagate-fixed-cong*:
(⋀*s*. *P s* = *P' s*) ⟹ (⋀*s*. *P' s* ⟹ *A* = *A'*) ⟹ *L2corres st ret ex P A C* = *L2corres st ret ex P' A' C*
⟨*proof*⟩

lemma *L2corres-modifycall*:
⟦ *L2corres st ret′ ex′ P′ Z dest-fn*;
⋀*s*. *P s* ⟹ *P′* (*scope-setup s*);
⋀*s r*. *P r* ⟹ *F* (*ret′ s*) (*st s*) = *st* (*return-xf s* (*scope-teardown r s*));
⋀*s*. *st* (*scope-setup s*) = *st s*;
⋀*t s*. *st* (*scope-teardown s t*) = *st t*;
⋀*t s*. *st* (*result-exn* (*scope-teardown s t*) *t*) = *st t*;
⋀*t s*. *P s* ⟹ *ex* (*result-exn* (*scope-teardown s t*) *t*) = *emb* (*ex′ t*)⟧ ⟹
L2corres st ret ex P (*L2-modifycall Z F emb ns*)
(*L1-call scope-setup dest-fn scope-teardown result-exn return-xf*)
⟨*proof*⟩

lemma *L2corres-voidcall*:
⟦ *L2corres st ret′ ex′ P′ Z dest-fn*;
⋀*s*. *P s* ⟹ *P′* (*scope-setup s*);
⋀*s r*. *st* (*return-xf s* (*scope-teardown r s*)) = *st s*;
⋀*s*. *st* (*scope-setup s*) = *st s*;
⋀*t s*. *st* (*scope-teardown s t*) = *st t*;
⋀*t s*. *st* (*result-exn* (*scope-teardown s t*) *t*) = *st t*;
⋀*t s*. *P s* ⟹ *ex* (*result-exn* (*scope-teardown s t*) *t*) = *emb* (*ex′ t*)⟧ ⟹
L2corres st ret ex P (*L2-voidcall Z emb ns*)
(*L1-call scope-setup dest-fn scope-teardown result-exn return-xf*)
⟨*proof*⟩

lemma *L2corres-call*:
⟦ *L2corres st ret′ ex′ P′ Z dest-fn*;
⋀*s*. *P s* ⟹ *P′* (*scope-setup s*);
⋀*s r*. *st* (*return-xf s* (*scope-teardown r s*)) = *st s*;
⋀*s r*. *ret* (*return-xf s* (*scope-teardown r s*)) = *ret′ s*;
⋀*s*. *st* (*scope-setup s*) = *st s*;
⋀*t s*. *st* (*scope-teardown s t*) = *st t*;
⋀*t s*. *st* (*result-exn* (*scope-teardown s t*) *t*) = *st t*;
⋀*t s*. *P s* ⟹ *ex* (*result-exn* (*scope-teardown s t*) *t*) = *emb* (*ex′ t*) ⟧ ⟹
L2corres st ret ex P (*L2-call Z emb ns*)
(*L1-call scope-setup dest-fn scope-teardown result-exn return-xf*)
⟨*proof*⟩

lemma (**in** *L1-functions*) *L2corres-dyn-call*:
L2corres st ret ex P X
(*L1-guarded g* (*gets dest* >>= (λ*p*. *L1-call scope-setup* (*𝒫 p*) *scope-teardown result-exn return-xf*))) ⟹
L2corres st ret ex P X (*L1-dyn-call g scope-setup dest scope-teardown result-exn return-xf*)
⟨*proof*⟩

lemma *L2-gets-bind*: $[\![\bigwedge s\ s'.\ V\ s = V\ s']\!] \Longrightarrow$ *L2-seq* (*L2-gets* $V\ n$) $f = f$ (V *undefined*)
⟨*proof*⟩

lemma *L2corres-folded-gets*:
$[\![\bigwedge a.$ *L2corres st ret ex* (P *and* ($\lambda s.\ a = c$ (*st* s))) ($X\ a$) $Y]\!] \Longrightarrow$
L2corres st ret ex P (*L2-seq* (*L2-folded-gets* c *ns*) X) Y
⟨*proof*⟩

lemma *L2-call-cong* [*fundef-cong*, *cong*]:
$f = f' \Longrightarrow$ *L2-call* f = *L2-call* f'
⟨*proof*⟩

lemma *L2-call-liftE* [*simp*]:
L2-call (*liftE* x) *emb ns* ≡ *liftE* x
⟨*proof*⟩

lemma *L2-call-fail* [*simp*]: *L2-call fail emb ns* = *fail*
⟨*proof*⟩

lemma *L2-call-L2-gets* [*simp*]: *L2-call* (*L2-gets* $x\ n$) *emb ns* = *L2-gets* $x\ n$
⟨*proof*⟩

lemma *L2-split-fixup-1*:
(*L2-seq* A ($\lambda x.$ *case* y *of* (a, b) ⇒ $B\ a\ b\ x$)) =
(*case* y *of* (a, b) ⇒ *L2-seq* A ($\lambda x.\ B\ a\ b\ x$))
⟨*proof*⟩
lemma *L2-split-fixup-2*:
(*L2-seq* (*case* y *of* (a, b) ⇒ $B\ a\ b$) A) =
(*case* y *of* (a, b) ⇒ *L2-seq* ($B\ a\ b$) A)
⟨*proof*⟩

lemma *L2-split-L2-seq-fixup-both*:
(*L2-seq* (*case* y *of* (a,b) ⇒ $A\ a\ b$) (*case* y *of* (a, b) ⇒ $B\ a\ b$)) =
(*case* y *of* (a, b) ⇒ *L2-seq* ($A\ a\ b$) ($B\ a\ b$))
⟨*proof*⟩

lemma *L2-split-L2-seq-gets-fixup-1*:
(*L2-seq-gets* (*case* y *of* (a,b) ⇒ $A\ a\ b$) n (*case* y *of* (a, b) ⇒ $B\ a\ b$)) =
(*case* y *of* (a, b) ⇒ *L2-seq-gets* ($A\ a\ b$) n ($B\ a\ b$))
⟨*proof*⟩

lemma *L2-split-fixup-3*:
(*case* (x, y) *of* (a, b) ⇒ $P\ a\ b$) = $P\ x\ y$

⟨*proof*⟩
lemma *L2-split-fixup-4*:
case-prod (λ*a* (*b* :: *'a* × *'b*). *P a*) = *case-prod* (λ*a*. *case-prod* (λ(*x* :: *'a*) (*y* :: *'b*). *P a*))
⟨*proof*⟩
lemma *L2-split-fixup-f*:
(*f* (*case y of* (*a*, *b*) ⇒ *G a b*) =
(*case y of* (*a*, *b*) ⇒ *f* (*G a b*)))
⟨*proof*⟩
lemma *L2-split-fixup-g*:
case-prod (λ*a* (*b* :: *'a* × *'b*). *P a b*) = *case-prod* (λ*a*. *case-prod* (λ(*x* :: *'a*) (*y* :: *'b*). *P a* (*x*, *y*)))
⟨*proof*⟩

lemma *liftE-fixup*: (λ*x*. *liftE* (*case x of* (*a*, *b*) ⇒ *f a b*)) = (λ(*a*,*b*). *liftE* (*f a b*))
⟨*proof*⟩

lemma *finally-fixup*: (λ*x*. *finally* (*case x of* (*a*, *b*) ⇒ *f a b*)) = (λ(*a*,*b*). *finally* (*f a b*))
⟨*proof*⟩

lemma *try-fixup*: (λ*x*. *try* (*case x of* (*a*, *b*) ⇒ *f a b*)) = (λ(*a*,*b*). *try* (*f a b*))
⟨*proof*⟩

lemma *L2-gets-const-split-fixup*: *L2-gets* (λ-. *case y of* (*a*, *b*) ⇒ *G a b*) = (*case y of* (*a*, *b*) ⇒ *L2-gets* (λ-. *G a b*))
⟨*proof*⟩

definition *STOP* :: *'a*::{} ⇒ *'a*
where *STOP P* ≡ *P*

lemma *STOP-cong*: *STOP P* ≡ *STOP P*
⟨*proof*⟩

lemma *do-STOP*: *P* ≡ *STOP P*
⟨*proof*⟩

definition *STOP-UNBIND* :: *'a*::{} ⇒ *'a*
where *STOP-UNBIND P* ≡ *P*

lemma *STOP-UNBIND-cong*: *STOP-UNBIND P* ≡ *STOP-UNBIND P*
⟨*proof*⟩

lemma *do-STOP-UNBIND*: *P* ≡ *STOP-UNBIND P*
⟨*proof*⟩

definition *FUSE* :: *'a*::{} ⇒ *'a*
where *FUSE P* ≡ *P*

lemma *FUSE-cong*: $FUSE\ P \equiv FUSE\ P$
⟨*proof*⟩

lemma *do-FUSE*: $P \equiv FUSE\ P$
⟨*proof*⟩

lemma *FUSE-STOP*: $X \equiv X' \Longrightarrow FUSE\ X \equiv STOP\ X'$
⟨*proof*⟩

lemma *fixup-accumulate*:
$(\lambda(x,\ a).\ (case\ a\ of\ (x1,\ x2) \Rightarrow C\ x1\ x2)\ x) = (\lambda(x,\ x1,\ x2).\ C\ x1\ x2\ x)$
⟨*proof*⟩

lemma *case-prod-apply-distrib*:
$(case\ x\ of\ (a,\ b) \Rightarrow f\ a\ b)\ n = \ (case\ x\ of\ (a,\ b) \Rightarrow f\ a\ b\ n\ )$
⟨*proof*⟩

lemma *L2-guard-fixup1*: $L2\text{-}guard\ (\lambda s.\ case\ y\ of\ (a,\ b) \Rightarrow G\ a\ b\ s) = (case\ y\ of\ (a,\ b) \Rightarrow L2\text{-}guard\ (G\ a\ b))$
⟨*proof*⟩

lemmas *L2-split-fixups-base* =

L2-split-fixup-3
L2-split-fixup-4
liftE-fixup
finally-fixup
try-fixup

L2-guard-fixup1
L2-split-fixup-f [**where** *f*=*L2-guard*]
L2-split-fixup-f [**where** *f*=*L2-gets*]
L2-split-fixup-f [**where** *f*=*L2-modify*]

L2-gets-const-split-fixup

L2-split-fixup-g [**where** $P{=}\lambda a\ b.\ L2\text{-}gets\ (P\ a\ b)\ n$ **for** $P\ n$]
L2-split-fixup-g [**where** $P{=}\lambda a\ b.\ L2\text{-}guard\ (P\ a\ b)$ **for** P]
L2-split-fixup-g [**where** $P{=}\lambda a\ b.\ L2\text{-}modify\ (P\ a\ b)$ **for** P]
L2-split-fixup-g [**where** $P{=}\lambda a\ b.\ L2\text{-}spec\ (P\ a\ b)$ **for** P]
L2-split-fixup-g [**where** $P{=}\lambda a\ b.\ L2\text{-}assume\ (P\ a\ b)$ **for** P]
L2-split-fixup-g [**where** $P{=}\lambda a\ b.\ L2\text{-}throw\ (P\ a\ b)\ n$ **for** $P\ n$]

L2-split-fixup-g [**where** $P{=}\lambda a\ b.\ L2\text{-}seq\ (L\ a\ b)\ (R\ a\ b)$ **for** $L\ R$]
L2-split-fixup-g [**where** $P{=}\lambda a\ b.\ L2\text{-}while\ (C\ a\ b)\ (B\ a\ b)\ (I\ a\ b)\ n$ **for** $C\ B\ I\ n$]
L2-split-fixup-g [**where** $P{=}\lambda a\ b.\ L2\text{-}unknown\ n$ **for** n]
L2-split-fixup-g [**where** $P{=}\lambda a\ b.\ L2\text{-}catch\ (L\ a\ b)\ (R\ a\ b)$ **for** $L\ R$]
L2-split-fixup-g [**where** $P{=}\lambda a\ b.\ L2\text{-}condition\ (C\ a\ b)\ (L\ a\ b)\ (R\ a\ b)$ **for** $C\ L\ R$]

L2-split-fixup-g [**where** *P*=λ*a b*. *L2-call* (*M a b*) **for** *M*]
L2-split-fixup-g [**where** *P*=λ*a b*. *liftE* (*M a b*) **for** *M*]
L2-split-fixup-g [**where** *P*=λ*a b*. *finally* (*M a b*) **for** *M*]
L2-split-fixup-g [**where** *P*=λ*a b*. *try* (*M a b*) **for** *M*]

lemmas *L2-split-fixups* =
L2-split-fixup-1
L2-split-fixup-2
L2-split-fixups-base

lemmas *L2-split-fixups'* =
L2-split-L2-seq-fixup-both
L2-split-L2-seq-gets-fixup-1
L2-split-fixup-g [**where** *P*=λ*a b*. *L2-seq-gets* (*L a b*) (*R a b*) **for** *L R*]
fixup-accumulate
L2-split-fixups-base
L2-split-fixup-f [**where** *f*=*STOP*]
case-prod-apply-distrib

lemmas *L2-split-fixups-congs* =
prod.case-cong

named-theorems *L2opt*

lemma *monotone-nondet-monad-L2-seq-le* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≤) *X*
assumes *mono-Y*: ⋀*r*. *monotone R* (≤) (λ*f*. *Y f r*)
shows *monotone R* (≤)
(λ*f*. (*L2-seq* (*X f*) (*Y f*)))
⟨*proof*⟩

lemma *monotone-nondet-monad-L2-seq-ge* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≥) *X*
assumes *mono-Y*: ⋀*r*. *monotone R* (≥) (λ*f*. *Y f r*)
shows *monotone R* (≥)
(λ*f*. (*L2-seq* (*X f*) (*Y f*)))
⟨*proof*⟩

lemma *monotone-nondet-monad-L2-try-le* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≤) *X*
shows *monotone R* (≤)
(λ*f*. (*L2-try* (*X f*)))
⟨*proof*⟩

lemma *monotone-nondet-monad-L2-try-ge* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≥) *X*
shows *monotone R* (≥)

(λf. (L2-try (X f)))
⟨proof⟩

lemma *monotone-nondet-monad-L2-catch-le* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≤) *X*
assumes *mono-Y*: ⋀r. *monotone R* (≤) (λf. *Y f r*)
shows *monotone R* (≤)
(λf. (L2-catch (X f) (Y f)))
⟨proof⟩

lemma *monotone-nondet-monad-L2-catch-ge* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≥) *X*
assumes *mono-Y*: ⋀r. *monotone R* (≥) (λf. *Y f r*)
shows *monotone R* (≥)
(λf. (L2-catch (X f) (Y f)))
⟨proof⟩

lemma *monotone-nondet-monad-L2-condition-le* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≤) *X*
assumes *mono-Y*: *monotone R* (≤) *Y*
shows *monotone R* (≤)
(λf. (L2-condition C (X f) (Y f)))
⟨proof⟩

lemma *monotone-nondet-monad-L2-condition-ge* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (≥) *X*
assumes *mono-Y*: *monotone R* (≥) *Y*
shows *monotone R* (≥)
(λf. (L2-condition C (X f) (Y f)))
⟨proof⟩

lemma *monotone-nondet-monad-L2-guarded-le* [*partial-function-mono*]:
assumes *mono-X*[*partial-function-mono*]: *monotone R* (≤) *X*
shows *monotone R* (≤)
(λf. L2-guarded C (X f))
⟨proof⟩

lemma *monotone-nondet-monad-L2-guarded-ge* [*partial-function-mono*]:
assumes *mono-X*[*partial-function-mono*]: *monotone R* (≥) *X*
shows *monotone R* (≥)
(λf. L2-guarded C (X f))
⟨proof⟩

lemma *monotone-nondet-monad-L2-while-le* [*partial-function-mono*]:
assumes *mono-B*: ⋀r. *monotone R* (≤) (λf. *B f r*)
shows *monotone R* (≤) (λf. L2-while C (B f) I ns)
⟨proof⟩

lemma *monotone-nondet-monad-L2-while-ge* [*partial-function-mono*]:
assumes *mono-B*: $\bigwedge r.$ *monotone R* ($\geq$) ($\lambda f.$ *B f r*)
shows *monotone R* ($\geq$) ($\lambda f.$ *L2-while C* (*B f*) *I ns*)
⟨*proof*⟩

lemma *monotone-nondet-monad-map-exn-le* [*partial-function-mono*]:
assumes *X*[*partial-function-mono*]: *monotone R* ($\leq$) *X*
shows *monotone R* ($\leq$) ($\lambda f.$ *map-value* (*map-exn emb*) (*X f*))
⟨*proof*⟩

lemma *monotone-nondet-monad-map-exn-ge* [*partial-function-mono*]:
assumes *X*[*partial-function-mono*]: *monotone R* ($\geq$) *X*
shows *monotone R* ($\geq$) ($\lambda f.$ *map-value* (*map-exn emb*) (*X f*))
⟨*proof*⟩

lemma *monotone-nondet-monad-L2-call-le* [*partial-function-mono*]:
assumes *X*[*partial-function-mono*]: *monotone R* ($\leq$) *X*
shows *monotone R* ($\leq$)
($\lambda f.$ *L2-call* (*X f*) *emb ns*)
⟨*proof*⟩

lemma *monotone-nondet-monad-L2-call-ge* [*partial-function-mono*]:
assumes *X*[*partial-function-mono*]: *monotone R* ($\geq$) *X*
shows *monotone R* ($\geq$)
($\lambda f.$ *L2-call* (*X f*) *emb ns*)
⟨*proof*⟩

19.1 Some Relators

lemma *exists-sum-unit-eq*: ($\exists\, l'$::*unit*. *Inl x* = *Inl l'* $\wedge$ *Inr y* = *Inr l'*) = (*x*=() $\wedge$ *y*=())
⟨*proof*⟩

lemmas *unit-convs* = *exists-sum-unit-eq*

definition *rel-Nonlocal* = ($\lambda e\ v.$ *case e of Nonlocal x* $\Rightarrow$ *v* = *x* | - $\Rightarrow$ *False*)

lemma *rel-Nonlocal-conv*: (*rel-Nonlocal e v*) = (*e* = *Nonlocal v*)
⟨*proof*⟩

lemma *fun-of-rel-rel-Nonlocal*[*fun-of-rel-intros*]: *fun-of-rel rel-Nonlocal the-Nonlocal*
⟨*proof*⟩

lemma *funp-rel-Nonlocal*[*funp-intros*, *corres-admissible*]: *funp rel-Nonlocal*
⟨*proof*⟩

lemma *rel-sum-rel-Nonlocal-case-InlI*: *e* = *Nonlocal v'* $\Longrightarrow$ *rel-sum rel-Nonlocal* (=) (*case e of Nonlocal v* $\Rightarrow$ *Inl* (*Nonlocal v*) | - $\Rightarrow$ *Inr x*) (*Inl v'*)

⟨*proof*⟩

lemma *rel-xval-rel-Nonlocal-case-ExnI*: $e = Nonlocal\ v' \Longrightarrow rel\text{-}xval\ rel\text{-}Nonlocal\ (=)\ (case\ e\ of\ Nonlocal\ v \Rightarrow Exn\ (Nonlocal\ v)\ |\ \text{-} \Rightarrow Result\ x)\ (Exn\ v')$
⟨*proof*⟩

lemma *rel-sum-rel-Nonlocal-case-InrI*: $is\text{-}local\ e \Longrightarrow x = v' \Longrightarrow rel\text{-}sum\ rel\text{-}Nonlocal\ (=)\ (case\ e\ of\ Nonlocal\ v \Rightarrow Inl\ (Nonlocal\ v)\ |\ \text{-} \Rightarrow Inr\ x)\ (Inr\ v')$
⟨*proof*⟩

lemma *rel-xval-rel-Nonlocal-case-ResultI*: $is\text{-}local\ e \Longrightarrow x = v' \Longrightarrow rel\text{-}xval\ rel\text{-}Nonlocal\ (=)\ (case\ e\ of\ Nonlocal\ v \Rightarrow Exn\ (Nonlocal\ v)\ |\ \text{-} \Rightarrow Result\ x)\ (Result\ v')$
⟨*proof*⟩

lemma *rel-sum-rel-Nonlocal-map-sumI*: $v = (map\text{-}sum\ (\lambda x.\ x)\ f\ (case\ e\ of\ Nonlocal\ x \Rightarrow Inl\ x\ |\ \text{-} \Rightarrow Inr\ x)) \Longrightarrow$
$rel\text{-}sum\ rel\text{-}Nonlocal\ (=)\ (map\text{-}sum\ Nonlocal\ f\ (case\ e\ of\ Nonlocal\ x \Rightarrow Inl\ x\ |\ \text{-} \Rightarrow Inr\ x))\ v$
⟨*proof*⟩

lemma *rel-xval-rel-Nonlocal-map-xvalI*: $v = (map\text{-}xval\ (\lambda x.\ x)\ f\ (case\ e\ of\ Nonlocal\ x \Rightarrow Exn\ x\ |\ \text{-} \Rightarrow Result\ x)) \Longrightarrow$
$rel\text{-}xval\ rel\text{-}Nonlocal\ (=)\ (map\text{-}xval\ Nonlocal\ f\ (case\ e\ of\ Nonlocal\ x \Rightarrow Exn\ x\ |\ \text{-} \Rightarrow Result\ x))\ v$
⟨*proof*⟩

definition $rel\text{-}Inr\ v\ v' \equiv (v = Inr\ v')$

lemma *fun-of-rel-rel-Inr*[*fun-of-rel-intros*]:
$fun\text{-}of\text{-}rel\ rel\text{-}Inr\ (\lambda x.\ case\ x\ of\ Inr\ v \Rightarrow v\ |\ Inl\ \text{-} \Rightarrow undefined)$
⟨*proof*⟩

lemma *funp-rel-Inr*[*funp-intros*, *corres-admissible*]: *funp rel-Inr*
⟨*proof*⟩

lemma *rel-InrI*: $v = Inr\ v' \Longrightarrow rel\text{-}Inr\ v\ v'$
⟨*proof*⟩

lemma *rel-Inr-apply*: $rel\text{-}Inr\ x\ y = (x = Inr\ y)$
⟨*proof*⟩

lemma *rel-Inr-trivial* [*simp*]: $rel\text{-}Inr\ (Inr\ v)\ v$
⟨*proof*⟩

definition *rel-liftE*:: $('e, 'a)\ xval \Rightarrow 'a\ val \Rightarrow bool$ **where** $rel\text{-}liftE\ v\ v' \equiv (v =$

Result (*the-Res v'*))

lemma *fun-of-rel-rel-liftE*[*fun-of-rel-intros*]:
fun-of-rel rel-liftE (λ*x*. *case x of Result v* ⇒ *Result v* | *Exn* - ⇒ *undefined*)
⟨*proof*⟩

lemma *funp-rel-liftE*[*funp-intros*, *corres-admissible*]: *funp rel-liftE*
⟨*proof*⟩

lemma *rel-liftEI*: *v* = *Result* (*the-Res v'*) ⟹ *rel-liftE v v'*
⟨*proof*⟩

lemma *rel-liftE-apply*: *rel-liftE x y* = (*x* = *Result* (*the-Res y*))
⟨*proof*⟩

lemma *rel-liftE-trivial* [*simp*]: *rel-liftE* (*Result v*) (*Result v*)
⟨*proof*⟩

lemma *rel-liftE-rel-exception-or-result-conv*: *rel-liftE* = *rel-exception-or-result* (λ*None* ⇒ λ-. *True* | *Some* - ⇒ λ-. *False*) (=)
⟨*proof*⟩

lemma *rel-liftE-Result-Result-iff*[*simp*]: *rel-liftE* (*Result v*) (*Result v'*) ⟷ *v* = *v'*
⟨*proof*⟩

definition *rel-liftE' v v'* ≡ (*v* = *Inr v'*)

lemma *fun-of-rel-rel-liftE'*[*fun-of-rel-intros*]:
fun-of-rel rel-liftE' (λ*x*. *case x of Inr v* ⇒ *v* | *Inl* - ⇒ *undefined*)
⟨*proof*⟩

lemma *funp-rel-liftE'*[*funp-intros*, *corres-admissible*]: *funp rel-liftE'*
⟨*proof*⟩

lemma *rel-liftE'I*: *v* = *Inr v'* ⟹ *rel-liftE' v v'*
⟨*proof*⟩

lemma *rel-liftE'-apply*: *rel-liftE' x y* = (*x* = *Inr y*)
⟨*proof*⟩

lemma *rel-liftE'-trivial* [*simp*]: *rel-liftE'* (*Inr v*) *v*
⟨*proof*⟩

lemma *rel-liftE'-Inr-iff*[*simp*]: *rel-liftE'* (*Inr v*) *v'* ⟷ *v* = *v'*
⟨*proof*⟩

lemma *rel-liftE'-rel-liftE-conv*: *rel-liftE'* = *rel-map to-xval OO rel-liftE OO rel-map*

the-Res
⟨*proof*⟩

lemma *rel-liftE-rel-liftE'-conv*: *rel-liftE* = *rel-map from-xval OO rel-liftE' OO rel-map Result*
⟨*proof*⟩

lemma *rel-liftE'-Inr*: *rel-liftE'* (*Inr x*) *x*
⟨*proof*⟩

definition *rel-throwE'* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ '*a* + '*c* ⇒ '*b* ⇒ *bool* **where**
rel-throwE' L a c ⟷ (*case a of Inl a'* ⇒ *L a' c* | *Inr* - ⇒ *False*)

lemma *rel-throwE'-iff*: *rel-throwE' L a c* ⟷ (∃ *l*. *a* = *Inl l* ∧ *L l c*)
⟨*proof*⟩

lemma *rel-throwE'-Inl*:
L x y ⟹ *rel-throwE' L* (*Inl x*) *y*
⟨*proof*⟩

lemma *rel-throwE'-Inl-iff*[*simp*]:
rel-throwE' L (*Inl x*) *y* ⟷ *L x y*
⟨*proof*⟩

lemma *not-rel-throwE'-Inr*[*simp*]: ¬*rel-throwE' R* (*Inr d*) *a*
⟨*proof*⟩

definition *rel-throwE* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ ('*a*, '*c*) *xval* ⇒ '*b* ⇒ *bool* **where**
rel-throwE L a c ⟷ (*case a of Exn a'* ⇒ *L a' c* | *Result* - ⇒ *False*)

lemma *rel-throwE-iff*: *rel-throwE L a c* ⟷ (∃ *e*. *a* = *Exn e* ∧ *L e c*)
⟨*proof*⟩

lemma *rel-throwE-Exn*:
L x y ⟹ *rel-throwE L* (*Exn x*) *y*
⟨*proof*⟩

lemma *rel-throwE-Exn-iff*[*simp*]:
rel-throwE L (*Exn x*) *y* ⟷ *L x y*
⟨*proof*⟩

lemma *not-rel-throwE-Result*[*simp*]: ¬*rel-throwE R* (*Result d*) *a*
⟨*proof*⟩

lemma *case-right-eq*: (*case v of Inl l* ⇒ *False* | *Inr r* ⇒ *P r*) = (∃ *r*. *P r* ∧ *v* = *Inr r*)

⟨*proof*⟩
lemma *case-left-eq*: (*case v of Inr r ⇒ False | Inl l ⇒ P l*) = (∃ *l*. *P l* ∧ *v* = *Inl l*)
⟨*proof*⟩

lemma *rel-sum-right*: *rel-sum L R* (*Inr r*) *v* = (∃ *r'*. *v* = *Inr r'* ∧ *R r r'*)
⟨*proof*⟩
lemma *rel-sum-left*: *rel-sum L R* (*Inl l*) *v* = (∃ *l'*. *v* = *Inl l'* ∧ *L l l'*)
⟨*proof*⟩

lemma *rel-sum-eq-apply*: (*rel-sum* (=) (=) *x y*) = (*x* = *y*)
⟨*proof*⟩

lemma *gen-unit-eq*: *x* = (*y*::*unit*)
⟨*proof*⟩

lemma *liftE-L2-while*: *L2-while c* (λ*r*. *liftE* (*B r*)) *i n* = *liftE* (*whileLoop c B i*)
⟨*proof*⟩

lemma *liftE-L2-while-VARS*: *L2-while c* (λ*r*. *liftE* (*B r*)) *i n* = *liftE* (*L2-VARS* (*whileLoop c B i*) *n*)
⟨*proof*⟩

lemma *rel-XF-True-def*: (*rel-XF st ret-xf ex-xf* (λ- -. *True*)) =
(λ(*v*, *t*) (*r*, *s*). *s* = *st t* ∧
(*case v of Exn x ⇒ r* = *Exn* (*ex-xf x t*) | *Result x ⇒ r* = *Result* (*ret-xf x t*)))
⟨*proof*⟩

lemma *refines-corresXF*: (∀ *t*. *P t* ⟶ *refines C A t* (*st t*) (λ(*v*, *t*) (*r*, *s*). *s* = *st t* ∧
(*case v of Exn x ⇒ r* = *Exn* (*ex-xf x t*) | *Result x ⇒ r* = *Result* (*ret-xf x t*))))
⟹ *corresXF st ret-xf ex-xf P A C*
⟨*proof*⟩

lemma *corresXF-refines*: *corresXF st ret-xf ex-xf P A C* ⟹
(∀ *t*. *P t* ⟶ *refines C A t* (*st t*) (λ(*v*, *t*) (*r*, *s*). *s* = *st t* ∧
(*case v of Exn x ⇒ r* = *Exn* (*ex-xf x t*) | *Result x ⇒ r* = *Result* (*ret-xf x t*))))
⟨*proof*⟩

theorem *corresXF-refines-conv*:
‹*corresXF st ret-xf ex-xf P A C* ⟷
(∀ *t*. *P t* ⟶ *refines C A t* (*st t*) (λ(*v*, *t*) (*r*, *s*). *s* = *st t* ∧
(*case v of Exn x ⇒ r* = *Exn* (*ex-xf x t*) | *Result x ⇒ r* = *Result* (*ret-xf x t*))))›
⟨*proof*⟩

lemma *sim-nondet-to-corresXF*: (⋀*s*. *refines f f' s s* (=)) ⟹
corresXF (λ*s*. *s*) (λ*r* -. *r*) (λ*r* - . *r*) (λ-. *True*) *f' f*

⟨*proof*⟩

named-theorems *rel-spec-monad-rewrite-simps*

locale *sim-stack-heap-state* =
  abstract: *stack-heap-state* htd_a *htd-upd*$_a$ $hmem_a$ *hmem-upd*$_a$ $\mathcal{S}$ +
  concrete: *stack-heap-state* htd_c *htd-upd*$_c$ $hmem_c$ *hmem-upd*$_c$ $\mathcal{S}$
  for
    htd_a:: $'s_a \Rightarrow$ *heap-typ-desc* **and**
    htd-upd$_a$:: (*heap-typ-desc* $\Rightarrow$ *heap-typ-desc*) $\Rightarrow$ $'s_a \Rightarrow$ $'s_a$ **and**
    $hmem_a$:: $'s_a \Rightarrow$ *heap-mem* **and**
    hmem-upd$_a$:: (*heap-mem* $\Rightarrow$ *heap-mem*) $\Rightarrow$ $'s_a \Rightarrow$ $'s_a$ **and**

    htd_c:: $'s_c \Rightarrow$ *heap-typ-desc* **and**
    htd-upd$_c$:: (*heap-typ-desc* $\Rightarrow$ *heap-typ-desc*) $\Rightarrow$ $'s_c \Rightarrow$ $'s_c$ **and**
    $hmem_c$:: $'s_c \Rightarrow$ *heap-mem* **and**
    hmem-upd$_c$:: (*heap-mem* $\Rightarrow$ *heap-mem*) $\Rightarrow$ $'s_c \Rightarrow$ $'s_c$ **and**
    $\mathcal{S}$::*addr set*
begin

definition *rel-stack-free* **where**
  rel-stack-free s_c s_a $\equiv$ *stack-free* (htd_c s_c) $\subseteq$ *stack-free* (htd_a s_a)

lemma *refines-rel-stack-free-with-fresh-stack-ptr*:
  assumes *s*: *rel-stack-free* s_c s_a
  assumes *init-eq*: $\bigwedge s_c\ s_a.\ I_c\ s_c = I_a\ s_a$
  assumes *f*: $\bigwedge s_c\ s_a\ p.$ *rel-stack-free* s_c $s_a \Longrightarrow$
    refines (f_c *p*) (f_a *p*) s_c s_a
      (*rel-prod* (*rel-xval L R*) *rel-stack-free*)
  shows *refines*
    (*concrete.with-fresh-stack-ptr n* I_c f_c)
    (*abstract.with-fresh-stack-ptr n* I_a f_a) s_c s_a
      (*rel-prod* (*rel-xval L R*) *rel-stack-free*)
  ⟨*proof*⟩

definition *rel-stack-free-eq* **where**
  rel-stack-free-eq s_c s_a $\equiv$ *stack-free* (htd_c s_c) = *stack-free* (htd_a s_a)

lemma *refines-rel-stack-free-eq-with-fresh-stack-ptr*:
  assumes *s*: *rel-stack-free-eq* s_c s_a
  assumes *init-eq*: $\bigwedge s_c\ s_a.\ I_c\ s_c = I_a\ s_a$
  assumes *f*: $\bigwedge s_c\ s_a\ p.$ *rel-stack-free-eq* s_c $s_a \Longrightarrow$
    refines (f_c *p*) (f_a *p*) s_c s_a
      (*rel-prod* (*rel-xval L R*) *rel-stack-free-eq*)
  shows *refines*
    (*concrete.with-fresh-stack-ptr n* I_c f_c)
    (*abstract.with-fresh-stack-ptr n* I_a f_a) s_c s_a

(*rel-prod* (*rel-xval L R*) *rel-stack-free-eq*)
⟨*proof*⟩

end

locale *sim-stack-heap-raw-state* =
abstract: *stack-heap-raw-state* hrs_a *hrs-upd*$_a$ $\mathcal{S}$ +
concrete: *stack-heap-raw-state* hrs_c *hrs-upd*$_c$ $\mathcal{S}$
for
hrs_a:: $'s_a \Rightarrow$ *heap-raw-state* **and**
hrs-upd$_a$:: (*heap-raw-state* $\Rightarrow$ *heap-raw-state*) $\Rightarrow$ $'s_a \Rightarrow$ $'s_a$ **and**

hrs_c:: $'s_c \Rightarrow$ *heap-raw-state* **and**
hrs-upd$_c$:: (*heap-raw-state* $\Rightarrow$ *heap-raw-state*) $\Rightarrow$ $'s_c \Rightarrow$ $'s_c$ **and**
$\mathcal{S}$::*addr set*
begin
sublocale *sim-stack-heap-state*
$\lambda s.$ *hrs-htd* (hrs_a s) $\lambda upd.$ *hrs-upd*$_a$ (*hrs-htd-update upd*)
$\lambda s.$ *hrs-mem* (hrs_a s) $\lambda upd.$ *hrs-upd*$_a$ (*hrs-mem-update upd*)
$\lambda s.$ *hrs-htd* (hrs_c s) $\lambda upd.$ *hrs-upd*$_c$ (*hrs-htd-update upd*)
$\lambda s.$ *hrs-mem* (hrs_c s) $\lambda upd.$ *hrs-upd*$_c$ (*hrs-mem-update upd*)
$\mathcal{S}$
⟨*proof*⟩
end

named-theorems *L2corres-with-fresh-stack-ptr*

locale *L2* = *sim-stack-heap-state* htd_a *htd-upd*$_a$ $hmem_a$ *hmem-upd*$_a$ htd_c *htd-upd*$_c$ $hmem_c$ *hmem-upd*$_c$ $\mathcal{S}$
for
htd_a:: $'s_a \Rightarrow$ *heap-typ-desc* **and**
htd-upd$_a$:: (*heap-typ-desc* $\Rightarrow$ *heap-typ-desc*) $\Rightarrow$ $'s_a \Rightarrow$ $'s_a$ **and**
$hmem_a$:: $'s_a \Rightarrow$ *heap-mem* **and**
hmem-upd$_a$:: (*heap-mem* $\Rightarrow$ *heap-mem*) $\Rightarrow$ $'s_a \Rightarrow$ $'s_a$ **and**

htd_c:: $'s_c \Rightarrow$ *heap-typ-desc* **and**
htd-upd$_c$:: (*heap-typ-desc* $\Rightarrow$ *heap-typ-desc*) $\Rightarrow$ $'s_c \Rightarrow$ $'s_c$ **and**
$hmem_c$:: $'s_c \Rightarrow$ *heap-mem* **and**
hmem-upd$_c$:: (*heap-mem* $\Rightarrow$ *heap-mem*) $\Rightarrow$ $'s_c \Rightarrow$ $'s_c$ **and**
$\mathcal{S}$::*addr set* +
fixes *st*:: $'s_c \Rightarrow 's_a$
assumes *htd-st*: $\bigwedge s.$ htd_a (*st s*) = htd_c s
assumes *htd-upd-st*: $\bigwedge s\ f.$ (*htd-upd*$_a$ f (*st s*)) = *st* (*htd-upd*$_c$ f s)
assumes *hmem-upd-st*: $\bigwedge s\ f.$ (*hmem-upd*$_a$ f (*st s*)) = *st* (*hmem-upd*$_c$ f s)

begin
lemma *rel-stack-free-eq-st*: *rel-stack-free-eq* s (*st s*)
⟨*proof*⟩

definition *rel-L2* **where**
rel-L2 ex-xf ret-xf ≡ (λ (r_c, s_c) (r_a, s_a).
rel-xval (λ- *a*. *a* = *ex-xf* s_c) (λ- *a*. *a* = *ret-xf* s_c) r_c r_a ∧
(s_a = *st* s_c))

lemma *rel-L2-def'*:
(*rel-L2 ex-xf ret-xf*) = (λ(*v*, *t*) (*r*, *s*).
s = *st t* ∧ (*case v of Exn x* ⇒ *r* = *Exn* (*ex-xf t*) | *Result x* ⇒ *r* = *Result* (*ret-xf t*)))
⟨*proof*⟩

lemma *refines-rel-L2-on-exit'*:
assumes *f*: *refines* (f_c *p*) (f_a *p*) s_c (*st* s_c) (*rel-L2 ex-xf ret-xf*)
assumes *ex-xf-htd-indep*: ⋀*f s*. *ex-xf* (*htd-upd*$_c$ *f s*) = *ex-xf s*
assumes *ret-xf-htd-indep*: ⋀*f s*. *ret-xf* (*htd-upd*$_c$ *f s*) = *ret-xf s*
shows *refines*
(*on-exit* (f_c *p*) {(*s*,*t*). *t* = *htd-upd*$_c$ (*release p*) *s*})
(*on-exit* (f_a *p*) {(*s*,*t*). *t* = *htd-upd*$_a$ (*release p*) *s*}) s_c (*st* s_c)
(*rel-L2 ex-xf ret-xf*)
⟨*proof*⟩

lemma *refines-rel-L2-on-exit*:
assumes *f*: *refines* (f_c *p*) (f_a *p*) s_c (*st* s_c) (*rel-L2 ex-xf ret-xf*)
assumes *ex-xf-htd-indep*: ⋀*g f s*. *ex-xf* (*hmem-upd*$_c$ *g* (*htd-upd*$_c$ *f s*)) = *ex-xf s*
assumes *ret-xf-htd-indep*: ⋀*g f s*. *ret-xf* (*hmem-upd*$_c$ *g* (*htd-upd*$_c$ *f s*)) = *ret-xf s*
assumes *nonempty-cleanup*$_a$: *cleanup*$_a$ ≠ {}
assumes *cleanup-htd*: ⋀*s s'*. (*s*, *s'*) ∈ *cleanup*$_c$ ⟹ ∃ *g f*. *s'* = (*hmem-upd*$_c$ *g* (*htd-upd*$_c$ *f s*))
assumes *stuck-sim*: ⋀*s*. ∄ *s'*. (*s*, *s'*) ∈ *cleanup*$_c$ ⟹ ∄ *t'*. (*st s*, *t'*) ∈ *cleanup*$_a$
assumes *cleanup-sim*: ⋀*s t*. (*s*, *t*) ∈ *cleanup*$_c$ ⟹ (*st s*, *st t*) ∈ *cleanup*$_a$
shows *refines*
(*on-exit* (f_c *p*) *cleanup*$_c$)
(*on-exit* (f_a *p*) *cleanup*$_a$) s_c (*st* s_c)
(*rel-L2 ex-xf ret-xf*)
⟨*proof*⟩

lemma *refines-rel-L2-with-fresh-stack-ptr*:
assumes *init-eq*: I_c s_c = I_a (*st* s_c)
assumes *ex-xf-htd-indep*: ⋀*g f s*. *ex-xf* (*hmem-upd*$_c$ *g* (*htd-upd*$_c$ *f s*)) = *ex-xf s*
assumes *ret-xf-htd-indep*: ⋀*g f s*. *ret-xf* (*hmem-upd*$_c$ *g* (*htd-upd*$_c$ *f s*)) = *ret-xf s*
assumes *f*: ⋀(*p*::'*a*::*mem-type ptr*) *d vs bs*. (*p*, *d*) ∈ *stack-allocs n* $\mathcal{S}$ *TYPE*('*a*) (*htd*$_c$ s_c) ⟹ *vs* ∈ I_c s_c ⟹ *length vs* = *n* ⟹
length bs = *n* ∗ *size-of TYPE*('*a*) ⟹
refines
(f_c *p*)
(f_a *p*)

(*hmem-upd*$_c$ (*fold* (λi. *heap-update-padding* (*p* +$_p$ *int i*) (*vs* ! *i*) (*take* (*size-of TYPE*('*a*)) (*drop* (*i* ∗ *size-of TYPE*('*a*)) *bs*))) [*0*..<*length vs*]) ((*htd-upd*$_c$ (λ-. *d*)) *s*$_c$))
(*st* (*hmem-upd*$_c$ (*fold* (λi. *heap-update-padding* (*p* +$_p$ *int i*) (*vs* ! *i*) (*take* (*size-of TYPE*('*a*)) (*drop* (*i* ∗ *size-of TYPE*('*a*)) *bs*))) [*0*..<*length vs*]) ((*htd-upd*$_c$ (λ-. *d*)) *s*$_c$)))
(*rel-L2 ex-xf ret-xf*)

shows *refines*
(*concrete.with-fresh-stack-ptr n I*$_c$ *f*$_c$)
(*abstract.with-fresh-stack-ptr n I*$_a$ *f*$_a$) *s*$_c$ (*st s*$_c$)
(*rel-L2 ex-xf ret-xf*)
⟨*proof*⟩

lemma *L2corres-refines-rel-L2-conv*:
‹*L2corres st ret-xf ex-xf P A C* ⟷
(∀ *t*. *P t* ⟶ *refines C A t* (*st t*) (*rel-L2 ex-xf ret-xf*))›
⟨*proof*⟩

lemma *L2corres-refines-rel-L2*:
assumes *L2*: *L2corres st ret-xf ex-xf P A C*
assumes *P*: *P s*$_c$
shows ‹*refines C A s*$_c$ (*st s*$_c$) (*rel-L2 ex-xf ret-xf*)›
⟨*proof*⟩

lemma *L2corres-with-fresh-stack-ptr*[*L2corres-with-fresh-stack-ptr*]:
assumes *l2*: ⋀*p*. *L2corres st ret-xf ex-xf P* (*l2 p*) (*l1 p*)
assumes *I*: ⋀*s*. *R s* ⟹ *I-l1 s* = *I-l2* (*st s*)
assumes *P*: ⋀*s*. *R s* ⟹ *P s*
assumes *ex-xf-htd-indep*: ⋀*g f s*. *ex-xf* (*hmem-upd*$_c$ *g* (*htd-upd*$_c$ *f s*)) = *ex-xf s*
assumes *ret-xf-htd-indep*: ⋀*g f s*. *ret-xf* (*hmem-upd*$_c$ *g* (*htd-upd*$_c$ *f s*)) = *ret-xf s*
assumes *P-indep*: ⋀*f g s*. *P* (*hmem-upd*$_c$ *g* (*htd-upd*$_c$ *f s*)) = *P s*
shows *L2corres st ret-xf ex-xf R*
(*abstract.with-fresh-stack-ptr n* (*I-l2*) (*L2-VARS l2 nm*))
(*concrete.with-fresh-stack-ptr n I-l1 l1*)
⟨*proof*⟩

end

hide-const (**open**) *L2*

lemma *refines-rel-prod-guard-on-exit*:
assumes *f*: *refines f*$_c$ *f*$_a$ *s*$_c$ *s*$_a$ (*rel-prod R S*′)
assumes *cleanup*: ⋀*s*$_c$ *s*$_a$ *t*$_c$. *S*′ *s*$_c$ *s*$_a$ ⟹ *grd s*$_a$ ⟹ (*s*$_c$, *t*$_c$) ∈ *cleanup*$_c$ ⟹ ∃ *t*$_a$. (*s*$_a$, *t*$_a$) ∈ *cleanup*$_a$ ∧ *S t*$_c$ *t*$_a$
assumes *emp*: ⋀*s*$_c$ *s*$_a$. *S*′ *s*$_c$ *s*$_a$ ⟹ ∄ *t*$_c$. (*s*$_c$, *t*$_c$) ∈ *cleanup*$_c$ ⟹ ∄ *t*$_a$. (*s*$_a$, *t*$_a$) ∈ *cleanup*$_a$
shows *refines* (*on-exit f*$_c$ *cleanup*$_c$) (*guard-on-exit f*$_a$ *grd cleanup*$_a$) *s*$_c$ *s*$_a$ (*rel-prod R S*)

⟨*proof*⟩

lemma *refines-bind-no-throw-strong*:
assumes *no-throw* (λ-. *True*) *f*
assumes *no-throw* (λ-. *True*) *f'*
assumes *f*: *refines f f' s s' Q*
assumes *g*: ⋀*r t r' t'*. *reaches f s* (*Result r*) *t* ⟹ *reaches f' s'* (*Result r'*) *t'* ⟹
Q (*Result r*, *t*) (*Result r'*, *t'*) ⟹ *refines* (*g r*) (*g' r'*) *t t' R*
shows *refines* (*f* >>= *g*) (*f'* >>= *g'*) *s s' R*
⟨*proof*⟩

lemma *refines-bind-no-throw*:
assumes *no-throw* (λ-. *True*) *f*
assumes *no-throw* (λ-. *True*) *f'*
assumes *f*: *refines f f' s s' Q*
assumes *g*: ⋀*r t r' t'*.
Q (*Result r*, *t*) (*Result r'*, *t'*) ⟹ *refines* (*g r*) (*g' r'*) *t t' R*
shows *refines* (*f* >>= *g*) (*f'* >>= *g'*) *s s' R*
⟨*proof*⟩

lemma *no-throw-guard*[*simp*]: *no-throw P* (*guard G*)
⟨*proof*⟩

lemma *no-throw-assert-result-and-state*[*simp*]: *no-throw P* (*assert-result-and-state f*)
⟨*proof*⟩

lemma *no-throw-assume-result-and-state*[*simp*]: *no-throw P* (*assume-result-and-state f*)
⟨*proof*⟩

lemma *refines-guard-same*: *refines* (*guard P*) (*guard P*) *s s* (λ(*r*, *t*) (*r'*, *t'*). *t*=*s* ∧ *t'*=*s*)
⟨*proof*⟩

lemma *refines-state-select-same*:
refines (*state-select R*) (*state-select R*) *s s* (λ(*r*, *t*) (*r'*, *t'*). *t'* = *t* ∧ (*s*, *t*) ∈ *R*)
⟨*proof*⟩

lemma *refines-assert-result-and-state-same*:
refines (*assert-result-and-state R*) (*assert-result-and-state R*) *s s*
(λ(*r*, *t*) (*r'*, *t'*). ∃ *v*. (*v*, *t*) ∈ *R s* ∧ *r* = *Result v* ∧ *r'* = *Result v* ∧ *t* = *t'*)
⟨*proof*⟩

lemma *refines-assume-result-and-state-same*:
refines (*assume-result-and-state R*) (*assume-result-and-state R*) *s s*
(λ(*r*, *t*) (*r'*, *t'*). ∃ *v*. (*v*, *t*) ∈ *R s* ∧ *r* = *Result v* ∧ *r'* = *Result v* ∧ *t* = *t'*)
⟨*proof*⟩

lemma *refines-assume-result-and-state-same'*:
assumes $R\ s = R'\ s$
shows *refines* (*assume-result-and-state* R) (*assume-result-and-state* R') $s\ s$
$(\lambda(r,\ t)\ (r',\ t').\ \exists\, v.\ (v,\ t) \in R\ s \wedge r = \mathit{Result}\ v \wedge r' = \mathit{Result}\ v \wedge t = t')$
⟨*proof*⟩

lemma *refines-rel-xval-guard-on-exit*:
assumes *f*: *refines* $f_c\ f_a\ s\ s$ (*rel-prod* (*rel-xval* $L\ R$) (=))
shows *refines*
(*guard-on-exit* $f_c\ P$ *cleanup*)
(*guard-on-exit* $f_a\ P$ *cleanup*) $s\ s$
(*rel-prod* (*rel-xval* $L\ R$) (=))
⟨*proof*⟩

lemma *refines-rel-xval-on-exit*:
assumes *f*: *refines* $f_c\ f_a\ s\ s$ (*rel-prod* (*rel-xval* $L\ R$) (=))
shows *refines*
(*on-exit* f_c *cleanup*)
(*on-exit* f_a *cleanup*) $s\ s$
(*rel-prod* (*rel-xval* $L\ R$) (=))
⟨*proof*⟩

lemma *refines-rel-xval-assume-on-exit*:
assumes *f*: *refines* $f_c\ f_a\ s\ s$ (*rel-prod* (*rel-xval* $L\ R$) (=))
shows *refines*
(*assume-on-exit* $f_c\ P$ *cleanup*)
(*assume-on-exit* $f_a\ P$ *cleanup*) $s\ s$
(*rel-prod* (*rel-xval* $L\ R$) (=))
⟨*proof*⟩

lemma *refines-state-assume-pred*:
assumes *P*: $P\ t$
assumes *ref*: *refines* $f\ g\ s\ t\ R$
shows *refines* f (*do* {*assume-result-and-state* $(\lambda s.\ \{(x,\ y).\ (\lambda()\ s'.\ s' = s \wedge P\ s)$ $x\ y\}); g\ \}) \ s\ t\ R$
⟨*proof*⟩

lemma *refines-rel-prod-assume-on-exit*:
assumes *f*: *refines* $f_c\ f_a\ s_c\ s_a$ (*rel-prod* $R\ S'$)
assumes *cleanup*: $\bigwedge s_c\ s_a\ t_c.\ S'\ s_c\ s_a \Longrightarrow (s_c,\ t_c) \in \mathit{cleanup}_c \Longrightarrow \exists\, t_a.\ (s_a,\ t_a)$ $\in \mathit{cleanup}_a \wedge S\ t_c\ t_a$
assumes *emp*: $\bigwedge s_c\ s_a.\ S'\ s_c\ s_a \Longrightarrow \nexists\, t_c.\ (s_c,\ t_c) \in \mathit{cleanup}_c \Longrightarrow \nexists\, t_a.\ (s_a,\ t_a) \in$ $\mathit{cleanup}_a$
assumes *conseq*: $\bigwedge s_c\ s_a.\ S'\ s_c\ s_a \Longrightarrow P\ s_a$
shows *refines* (*on-exit* f_c $\mathit{cleanup}_c$) (*assume-on-exit* $f_a\ P\ \mathit{cleanup}_a$) $s_c\ s_a$ (*rel-prod* $R\ S$)
⟨*proof*⟩

lemma *refines-runs-to-partial-rel-prod-assume-on-exit*:
assumes *f*: *refines* f_c f_a s_c s_a (*rel-prod R S'*)
assumes *runs-to*: $f_c \cdot s_c$ $?\{\!|\lambda r\ t.\ P\ t|\!\}$
assumes *cleanup*: $\bigwedge s_c\ s_a\ t_c.\ S'\ s_c\ s_a \Longrightarrow (s_c,\ t_c) \in cleanup_c \Longrightarrow P\ s_c \Longrightarrow \exists\, t_a.\ (s_a,\ t_a) \in cleanup_a \wedge S\ t_c\ t_a$
assumes *emp*: $\bigwedge s_c\ s_a.\ S'\ s_c\ s_a \Longrightarrow P\ s_c \Longrightarrow \nexists\, t_c.\ (s_c,\ t_c) \in cleanup_c \Longrightarrow \nexists\, t_a.\ (s_a,\ t_a) \in cleanup_a$
assumes *conseq*: $\bigwedge s_c\ s_a.\ \ S'\ s_c\ s_a \ \Longrightarrow P\ s_c \Longrightarrow P'\ s_a$
shows *refines* (*on-exit* f_c $cleanup_c$) (*assume-on-exit* f_a P' $cleanup_a$) s_c s_a (*rel-prod R S*)
⟨*proof*⟩

locale *L2-heap-raw-state* = *sim-stack-heap-raw-state* hrs_a $hrs\text{-}upd_a$ hrs_c $hrs\text{-}upd_c$ $\mathcal{S}$
for
hrs_a:: $'s_a \Rightarrow$ *heap-raw-state* **and**
$hrs\text{-}upd_a$:: (*heap-raw-state* ⇒ *heap-raw-state*) ⇒ $'s_a \Rightarrow\ 's_a$ **and**

hrs_c:: $'s_c \Rightarrow$ *heap-raw-state* **and**
$hrs\text{-}upd_c$:: (*heap-raw-state* ⇒ *heap-raw-state*) ⇒ $'s_c \Rightarrow\ 's_c$ **and**
$\mathcal{S}$::*addr set* +
fixes *st*:: $'s_c \Rightarrow\ 's_a$
assumes *hrs-htd-st*: $\bigwedge s.$ *hrs-htd* (hrs_a (*st s*)) = *hrs-htd* (hrs_c *s*)
assumes *hrs-upd-st*: $\bigwedge s\ f.$ ($hrs\text{-}upd_a$ *f* (*st s*)) = *st* ($hrs\text{-}upd_c$ *f s*)

begin
sublocale *L2*
$\lambda s.$ *hrs-htd* (hrs_a *s*) $\lambda upd.$ $hrs\text{-}upd_a$ (*hrs-htd-update upd*)
$\lambda s.$ *hrs-mem* (hrs_a *s*) $\lambda upd.$ $hrs\text{-}upd_a$ (*hrs-mem-update upd*)
$\lambda s.$ *hrs-htd* (hrs_c *s*) $\lambda upd.$ $hrs\text{-}upd_c$ (*hrs-htd-update upd*)
$\lambda s.$ *hrs-mem* (hrs_c *s*) $\lambda upd.$ $hrs\text{-}upd_c$ (*hrs-mem-update upd*)
$\mathcal{S}$
⟨*proof*⟩
end

locale *typ-heap-typing* = *pointer-lense r w* + *heap-typing-state heap-typing heap-typing-upd*

for
r:: $'t \Rightarrow$ ($'a$::*xmem-type*) *ptr* ⇒ $'a$ **and**
w:: $'a$ *ptr* ⇒ ($'a \Rightarrow\ 'a$) ⇒ $'t \Rightarrow\ 't$ **and**
heap-typing :: $'t \Rightarrow$ *heap-typ-desc* **and**
heap-typing-upd :: (*heap-typ-desc* ⇒ *heap-typ-desc*) ⇒ $'t \Rightarrow\ 't$ **and**
$\mathcal{S}$:: *addr set*
begin

definition *stack-ptr-acquire* :: *nat* ⇒ $'a$ *list* ⇒ $'a$ *ptr* ⇒ *heap-typ-desc* ⇒ $'t \Rightarrow\ 't$
where *stack-ptr-acquire n vs p d s* =
(*fold* ($\lambda i.$ *w* (*p* +$_p$ *int i*) (λ-. (*vs* ! *i*))) [*0*..<*n*]) (*heap-typing-upd* (λ-. *d*) *s*)

definition *stack-ptr-release* :: *nat* ⇒ *'a ptr* ⇒ *'t* ⇒ *'t*
where *stack-ptr-release n p s* =
(*heap-typing-upd* (*stack-releases n p*) *o* (*fold* (λ*i*. *w* (*p* +$_p$ *int i*) (λ-. *c-type-class.zero*)) [*0*..<*n*])) *s*

definition *assume-stack-alloc*:: *nat* ⇒ (*'t* ⇒ (*'a*::*xmem-type*) *list set*) ⇒ (*'e*::*default*, *'a ptr*, *'t*) *spec-monad* **where**
assume-stack-alloc n init ≡ *assume-result-and-state* (λ*s*. {(*p*, *t*).
∃ *d vs*. (*p*, *d*) ∈ *stack-allocs n* $\mathcal{S}$ *TYPE*(*'a*::*xmem-type*) (*heap-typing s*) ∧
vs ∈ *init s* ∧
length vs = *n* ∧
(∀ *i* ∈ {*0*..<*n*}. *r s* (*p* +$_p$ *int i*) = *ZERO*(*'a*::*xmem-type*)) ∧
t = *stack-ptr-acquire n vs p d s*})

definition *guard-with-fresh-stack-ptr* :: *nat* ⇒ (*'t* ⇒ *'a list set*) ⇒ (*'a*::*xmem-type ptr* ⇒ (*'e*::*default*, *'v*, *'t*) *spec-monad*) ⇒ (*'e*, *'v*, *'t*) *spec-monad* **where**
guard-with-fresh-stack-ptr n init c ≡
do {
p ← *assume-stack-alloc n init*;
guard-on-exit (*c p*)
(λ*s*. ∀ *i* < *n*. *root-ptr-valid* (*heap-typing s*) (*p* +$_p$ *int i*))
{(*s*, *t*). *t* = *stack-ptr-release n p s*}
}

definition *assume-with-fresh-stack-ptr* :: *nat* ⇒ (*'t* ⇒ *'a list set*) ⇒ (*'a*::*xmem-type ptr* ⇒ (*'e*::*default*,*'v*, *'t*) *spec-monad*) ⇒ (*'e*, *'v*, *'t*) *spec-monad* **where**
assume-with-fresh-stack-ptr n init c ≡
do {
p ← *assume-stack-alloc n init*;
assume-on-exit (*c p*)
(λ*s*. ∀ *i* < *n*. *root-ptr-valid* (*heap-typing s*) (*p* +$_p$ *int i*))
{(*s*, *t*). *t* = *stack-ptr-release n p s*}
}

definition *with-fresh-stack-ptr* :: *nat* ⇒ (*'t* ⇒ *'a list set*) ⇒ (*'a*::*xmem-type ptr* ⇒ (*'e*::*default*,*'v*, *'t*) *spec-monad*) ⇒ (*'e*, *'v*, *'t*) *spec-monad* **where**
with-fresh-stack-ptr n init c ≡
do {
p ← *assume-stack-alloc n init*;
on-exit (*c p*)
{(*s*, *t*). *t* = *stack-ptr-release n p s*}
}

lemma *monotone-guard-with-fresh-stack-ptr-le*[*partial-function-mono*]:
assumes [*partial-function-mono*]: ⋀*p*. *monotone R* (≤) (λ*f*. *c f p*)
shows *monotone R* (≤) (λ*f*. *guard-with-fresh-stack-ptr n I* (*c f*))
⟨*proof*⟩

lemma *monotone-guard-with-fresh-stack-ptr-ge*[*partial-function-mono*]:

assumes [*partial-function-mono*]: $\bigwedge p.$ *monotone* R $(\geq)$ $(\lambda f.$ c f $p)$
shows *monotone* R $(\geq)$ $(\lambda f.$ *guard-with-fresh-stack-ptr* n I $(c$ $f))$
⟨*proof*⟩

lemma *monotone-assume-with-fresh-stack-ptr-le*[*partial-function-mono*]:
assumes [*partial-function-mono*]: $\bigwedge p.$ *monotone* R $(\leq)$ $(\lambda f.$ c f $p)$
shows *monotone* R $(\leq)$ $(\lambda f.$ *assume-with-fresh-stack-ptr* n I $(c$ $f))$
⟨*proof*⟩

lemma *monotone-assume-with-fresh-stack-ptr-ge*[*partial-function-mono*]:
assumes [*partial-function-mono*]: $\bigwedge p.$ *monotone* R $(\geq)$ $(\lambda f.$ c f $p)$
shows *monotone* R $(\geq)$ $(\lambda f.$ *assume-with-fresh-stack-ptr* n I $(c$ $f))$
⟨*proof*⟩

lemma *monotone-with-fresh-stack-ptr-le*[*partial-function-mono*]:
assumes [*partial-function-mono*]: $\bigwedge p.$ *monotone* R $(\leq)$ $(\lambda f.$ c f $p)$
shows *monotone* R $(\leq)$ $(\lambda f.$ *with-fresh-stack-ptr* n I $(c$ $f))$
⟨*proof*⟩

lemma *monotone-with-fresh-stack-ptr-ge*[*partial-function-mono*]:
assumes [*partial-function-mono*]: $\bigwedge p.$ *monotone* R $(\geq)$ $(\lambda f.$ c f $p)$
shows *monotone* R $(\geq)$ $(\lambda f.$ *with-fresh-stack-ptr* n I $(c$ $f))$
⟨*proof*⟩

lemma *refines-rel-xval-guard-with-fresh-stack-ptr*:
assumes *init-eq*: $init_c$ s = $init_a$ s
assumes f: $\bigwedge s$ $p.$ *refines* $(f_c$ $p)$ $(f_a$ $p)$ s s (*rel-prod* (*rel-xval* L R) $(=)$)
shows
refines
(*guard-with-fresh-stack-ptr* n $init_c$ (*L2-VARS* f_c nm))
(*guard-with-fresh-stack-ptr* n $init_a$ (*L2-VARS* f_a nm)) s s
(*rel-prod* (*rel-xval* L R) $(=)$)
⟨*proof*⟩

lemma *refines-rel-xval-assume-with-fresh-stack-ptr*:
assumes *init-eq*: $init_c$ s = $init_a$ s
assumes f: $\bigwedge s$ $p.$ *refines* $(f_c$ $p)$ $(f_a$ $p)$ s s (*rel-prod* (*rel-xval* L R) $(=)$)
shows
refines
(*assume-with-fresh-stack-ptr* n $init_c$ (*L2-VARS* f_c nm))
(*assume-with-fresh-stack-ptr* n $init_a$ (*L2-VARS* f_a nm)) s s
(*rel-prod* (*rel-xval* L R) $(=)$)
⟨*proof*⟩

lemma *refines-rel-xval-with-fresh-stack-ptr*:
assumes *init-eq*: $init_c$ s = $init_a$ s
assumes f: $\bigwedge s$ $p.$ *refines* $(f_c$ $p)$ $(f_a$ $p)$ s s (*rel-prod* (*rel-xval* L R) $(=)$)
shows

refines
(*with-fresh-stack-ptr n init$_c$* (*L2-VARS f$_c$ nm*))
(*with-fresh-stack-ptr n init$_a$* (*L2-VARS f$_a$ nm*)) *s s*
(*rel-prod* (*rel-xval L R*) (=))
⟨*proof*⟩

lemma *write-same-ZERO*:
r s p = *ZERO*('*a*) ⟹ *w p* (λ*y*. *ZERO*('*a*)) *s* = *s*
⟨*proof*⟩

end

lemma *refines-rel-prod-eq-on-exit*:
assumes *f*: *refines f$_c$ f$_a$ s s* (*rel-prod Q* (=))
shows *refines*
(*on-exit f$_c$ cleanup*)
(*on-exit f$_a$ cleanup*) *s s*
(*rel-prod Q* (=))
⟨*proof*⟩

context *stack-heap-state*
begin
lemma *refines-rel-prod-with-fresh-stack-ptr*:
assumes *init-eq*: *init$_c$ s* = *init$_a$ s*
assumes *f*: ⋀*s p*. *refines* (*f$_c$ p*) (*f$_a$ p*) *s s* (*rel-prod Q* (=))
shows
refines
(*with-fresh-stack-ptr n init$_c$* (*L2-VARS f$_c$ nm*))
(*with-fresh-stack-ptr n init$_a$* (*L2-VARS f$_a$ nm*)) *s s*
(*rel-prod Q* (=))
⟨*proof*⟩
end

lemma *h-val-coerce-ptr-coerce-packed*:
fixes *p*::'*c*::*packed-type ptr*
assumes *sz-eq*: *size-of TYPE*('*c*) = *size-of TYPE*('*a*::*packed-type*)
shows *h-val h* (*PTR-COERCE* ('*c* → '*a*) *p*) = *COERCE* ('*c* → '*a*) (*h-val h p*)
⟨*proof*⟩

lemma *h-val-field-ptr-coerce-from-bytes-packed*:
fixes *p*::'*c*::*packed-type ptr*
assumes *field-ti TYPE*('*a*) *f* = *Some t*
assumes *export-uinfo t* = *export-uinfo* (*typ-info-t TYPE*('*b*::*mem-type*))
assumes [*simp*]: *size-of TYPE*('*c*) = *size-of TYPE*('*a*)
shows *h-val h* (*PTR*('*b*) &((*PTR-COERCE*('*c*::*packed-type* → '*a*::*packed-type*) *p*)→*f*)) =
from-bytes (*access-ti$_0$ t* (*COERCE* ('*c* → '*a*) (*h-val h p*)))

⟨*proof*⟩

lemma *packed-type-value-update-ti-explode*:
(*v*::'*a*::*packed-type*) = *update-ti* (*typ-info-t TYPE*('*a*)) (*to-bytes-p v*) *v*'
⟨*proof*⟩

lemma *packed-type-to-bytes-to-bytes-p-conv*:
length bs = *size-of TYPE*('*a*::*packed-type*) ⟹ *to-bytes v bs* = *to-bytes-p* (*v*::'*a*)
⟨*proof*⟩

lemma *packed-type-to-byte-p-coerce*:
assumes *sz*:*size-of TYPE*('*c*::*packed-type*) = *size-of TYPE*('*a*::*packed-type*)
shows *to-bytes-p* (*COERCE*('*a* → '*c*) *v*) = *to-bytes-p v*
⟨*proof*⟩

lemma *to-bytes-coerce-packed*:
size-of TYPE('*a*) = *size-of TYPE*('*b*) ⟹ *length bs* = *size-of TYPE*('*a*) ⟹
to-bytes (*COERCE*('*a*::*packed-type*→'*b*::*packed-type*) *v*) *bs* = *to-bytes v bs*
⟨*proof*⟩

lemma *heap-update-ptr-coerse-packed*:
fixes *p*::'*c*::*packed-type ptr*
assumes *cgrd*: *c-guard p*
assumes [*simp*]: *size-of TYPE*('*c*) = *size-of TYPE*('*a*::*packed-type*)
shows *heap-update* (*PTR-COERCE*('*c* → '*a*) *p*) *v* = *heap-update p* (*COERCE*('*a* →'*c*) *v*)
⟨*proof*⟩

lemma *c-guard-ptr-coerceI*:
size-td (*typ-info-t TYPE*('*c*)) = *size-td* (*typ-info-t TYPE*('*a*)) ⟹
align-td (*typ-info-t TYPE*('*a*)) ≤ *align-td* (*typ-info-t TYPE*('*c*)) ⟹
c-guard p ⟹
c-guard (*PTR-COERCE*('*c*::*c-type*→'*a*::*c-type*) *p*)
⟨*proof*⟩

lemma *heap-update-field-ptr-coerce-from-bytes-packed*:
fixes *p*::'*c*::{*xmem-type*, *packed-type*} *ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*::{*xmem-type*, *packed-type*})) *f* 0 = *Some* (*t*, *n*)
assumes *match*: *export-uinfo t* = *typ-uinfo-t* (*TYPE*('*b*::{*xmem-type*}))
assumes *cgrd*: *c-guard* (*PTR-COERCE*('*c*→ '*a*) *p*)
assumes *sz*[*simp*]: *size-of TYPE*('*c*) = *size-of TYPE*('*a*)
shows *heap-update* (*PTR*('*b*) &((*PTR-COERCE*('*c*→ '*a*) *p*)→*f*)) *v h* =
heap-update p (*coerce-map* (*update-ti t* (*to-bytes-p v*)) (*h-val h p*)) *h*
⟨*proof*⟩

named-theorems *L2-modify-heap-update-field-root-conv*

context *heap-state*
begin
lemma *heap-update-field-root-conv*:
fixes *p*::*'a*::*xmem-type ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*adjust-ti* (*typ-info-t TYPE*(*'b*::*xmem-type*)) *fld* (*fld-update* ∘ (λ*x* -. *x*)), *n*)
assumes *fg-cons*: *fg-cons fld* (*fld-update* ∘ (λ*x* -. *x*))
assumes *cgrd*: *c-guard p*
shows (*hmem-upd* (*heap-update* (*PTR*(*'b*) &(*p*→*f*)) *v*)) =
(λ*s*. (*hmem-upd* (*heap-update p* (*fld-update* (λ-. *v*) (*h-val* (*hmem s*) *p*)))) *s*)
⟨*proof*⟩

lemma *heap-update-field-root-conv'*:
fixes *p*::*'a*::*xmem-type ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*t*, *n*)
assumes *match*: *export-uinfo t* = *typ-uinfo-t TYPE*(*'b*::*xmem-type*)
assumes *cgrd*: *c-guard p*
shows (*hmem-upd* (*heap-update* (*PTR*(*'b*) &(*p*→*f*)) *v*)) =
(λ*s*. (*hmem-upd* (*heap-update p* (*update-ti t* (*to-bytes-p v*) (*h-val* (*hmem s*) *p*)))) *s*)
⟨*proof*⟩

lemma *L2-modify-heap-update-field-root-conv*:
fixes *p*::*'a*::*xmem-type ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*adjust-ti* (*typ-info-t TYPE*(*'b*::*xmem-type*)) *fld* (*fld-update* ∘ (λ*x* -. *x*)), *n*)
assumes *fg-cons*: *fg-cons fld* (*fld-update* ∘ (λ*x* -. *x*))
assumes *cgrd*: *c-guard p*
shows *L2-modify* (λ*s*. *hmem-upd* (*heap-update* (*PTR*(*'b*) &(*p*→*f*)) (*v s*)) *s*) =
L2-modify (λ*s*. (*hmem-upd* (*heap-update p* (*fld-update* (λ-. *v s*) (*h-val* (*hmem s*) *p*)))) *s*)
⟨*proof*⟩

lemma *L2-modify-heap-update-field-root-conv'* [*L2-modify-heap-update-field-root-conv*]:
fixes *p*::*'a*::*xmem-type ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*t*, *n*)
assumes *match*: *export-uinfo t* = *typ-uinfo-t TYPE*(*'b*::*xmem-type*)
assumes *cgrd*: *c-guard p*
shows *L2-modify* (λ*s*. *hmem-upd* (*heap-update* (*PTR*(*'b*) &(*p*→*f*)) (*v s*)) *s*) =
L2-modify (λ*s*. (*hmem-upd* (*heap-update p* (*update-ti t* (*to-bytes-p* (*v s*)) (*h-val* (*hmem s*) *p*)))) *s*)
⟨*proof*⟩

lemma *L2-modify-heap-update-ptr-coerce-packed-conv'*:
fixes *p*::*'c*::*packed-type ptr*
assumes *cgrd*: *c-guard p*

assumes *sz*: *size-td* (*typ-info-t TYPE*('*c*)) = *size-td* (*typ-info-t TYPE*('*a*::*packed-type*))

shows *L2-modify* (λ*s*. *hmem-upd* (*heap-update* (*PTR-COERCE*('*c* → '*a*) *p*) (*COERCE*('*c* → '*a*) (*v s*))) *s*) =
L2-modify (λ*s*. (*hmem-upd* (*heap-update p* (*v s*))) *s*)
⟨*proof*⟩

lemma *L2-modify-heap-update-ptr-coerce-packed-conv*[*L2-modify-heap-update-field-root-conv*]:
fixes *p*::'*c*::*packed-type ptr*
assumes *cgrd*: *c-guard p*
assumes *sz*: *size-td* (*typ-info-t TYPE*('*c*)) = *size-td* (*typ-info-t TYPE*('*a*::*packed-type*))

shows *L2-modify* (λ*s*. *hmem-upd* (*heap-update* (*PTR-COERCE*('*c* → '*a*) *p*) (*v s*)) *s*) =
L2-modify (λ*s*. (*hmem-upd* (*heap-update p* (*COERCE*('*a* → '*c*) (*v s*)))) *s*)
⟨*proof*⟩

lemma *L2-modify-heap-update-ptr-coerce-packed-field-root-conv* [*L2-modify-heap-update-field-root-conv*]:
fixes *p*::'*c*::{*xmem-type*, *packed-type*} *ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*::{*xmem-type*, *packed-type*})) *f* 0 = *Some* (*t*, *n*)
assumes *match*: *export-uinfo t* = *typ-uinfo-t TYPE*('*b*::*xmem-type*)
assumes *cgrd*: *c-guard* (*PTR-COERCE*('*c*→ '*a*) *p*)
assumes *sz*: *size-td* (*typ-info-t TYPE*('*c*)) = *size-td* (*typ-info-t TYPE*('*a*))
shows *L2-modify* (λ*s*. *hmem-upd* (*heap-update* (*PTR*('*b*) &((*PTR-COERCE*('*c*→ '*a*) *p*)→*f*)) (*v s*)) *s*) =
L2-modify (λ*s*. (*hmem-upd* (*heap-update p* (*coerce-map* (*update-ti t* (*to-bytes-p* (*v s*))) (*h-val* (*hmem s*) *p*)))) *s*)
⟨*proof*⟩
end

lemma *L2-try-state-asssumeE-bindE*:
L2-try (*assume-result-and-state P* >>= *f*) = *assume-result-and-state P* >>= (λ*x*. *L2-try* (*f x*))
⟨*proof*⟩

lemma *refines-L2-try-L2-seq-nondet*:
assumes *g* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (⋀*v t*. *refines* (*L2-try* (*g v*)) (*g*′ *v*) *t t* (*rel-prod rel-liftE* (=)))
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines f f*′ *s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-try* (*L2-seq f g*)) (*f*′ >>= *g*′) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-try-pure*:
assumes *f*: *refines f* (*return f*′) *s s* (*rel-prod rel-liftE* (=))

shows *refines* (*L2-try f*) (*return f'*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-liftE-conv*: *refines f f' s t* (*rel-prod rel-liftE* (=)) ⟷
refines f (*liftE f'*) *s t* (=)
⟨*proof*⟩

lemma *refines-rel-liftE-L2-try-on-exit*:
assumes *f*: *refines* f_c f_a *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-try* (*on-exit* f_c *cleanup*)) (*on-exit* f_a *cleanup*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *map-value-on-exit*:
map-value g (*on-exit f cleanup*) = *on-exit* (*map-value g f*) *cleanup*
⟨*proof*⟩

lemma *L2-try-on-exit*: *L2-try* (*on-exit f cleanup*) = *on-exit* (*L2-try f*) *cleanup*
⟨*proof*⟩

context *stack-heap-state*
begin
lemma *L2-try-with-fresh-stack-ptr*:
L2-try (*with-fresh-stack-ptr n init f*) = *with-fresh-stack-ptr n init* (λ*p*. *L2-try* (*f p*))
⟨*proof*⟩
end

lemma *map-value-guard-on-exit*:
map-value g (*guard-on-exit f P cleanup*) = *guard-on-exit* (*map-value g f*) *P cleanup*
⟨*proof*⟩

lemma *L2-try-guard-on-exit*:
L2-try (*guard-on-exit f P cleanup*) = *guard-on-exit* (*L2-try f*) *P cleanup*
⟨*proof*⟩

lemma *map-value-assume-on-exit*:
map-value g (*assume-on-exit f P cleanup*) = *assume-on-exit* (*map-value g f*) *P cleanup*
⟨*proof*⟩

lemma *L2-try-assume-on-exit*:
L2-try (*assume-on-exit f P cleanup*) = *assume-on-exit* (*L2-try f*) *P cleanup*
⟨*proof*⟩

context *typ-heap-typing*
begin
lemma *L2-try-guard-with-fresh-stack-ptr*:
L2-try (*guard-with-fresh-stack-ptr n init f*) = *guard-with-fresh-stack-ptr n init* (λ*p*. *L2-try* (*f p*))
⟨*proof*⟩

lemma *L2-try-assume-with-fresh-stack-ptr*:
L2-try (*assume-with-fresh-stack-ptr n init f*) = *assume-with-fresh-stack-ptr n init* (λ*p*. *L2-try* (*f p*))
⟨*proof*⟩

lemma *L2-try-with-fresh-stack-ptr*:
L2-try (*with-fresh-stack-ptr n init f*) = *with-fresh-stack-ptr n init* (λ*p*. *L2-try* (*f p*))
⟨*proof*⟩
end

lemma *refines-L2-seq*:
assumes *f*: *refines f f' s s' Q*
assumes *ll*: ⋀*e e' t t'*. *Q* (*Exn e*, *t*) (*Exn e'*, *t'*) ⟹ *R* (*Exn e*, *t*) (*Exn e'*, *t'*)
assumes *lr*: ⋀*e v' t t'*. *Q* (*Exn e*, *t*) (*Result v'*, *t'*) ⟹ *refines* (*throw e*) (*g' v'*) *t t' R*
assumes *rl*: ⋀*v e' t t'*. *Q* (*Result v*, *t*) (*Exn e'*, *t'*) ⟹ *refines* (*g v*) (*throw e'*) *t t' R*
assumes *rr*: ⋀*v v' t t'*. *Q* (*Result v*, *t*) (*Result v'*, *t'*) ⟹ *refines* (*g v*) (*g' v'*) *t t' R*
shows *refines* (*L2-seq f g*) (*L2-seq f' g'*) *s s' R*
⟨*proof*⟩

lemma *rel-Nonlocal-Nonlocal*: *rel-Nonlocal* (*Nonlocal x*) *x*
⟨*proof*⟩

lemmas *rel-sum-Inl* = *rel-sum.intros*(*1*)
lemmas *rel-sum-Inr* = *rel-sum.intros*(*2*)

lemma *rel-sum-eq*: *rel-sum* (=) (=) *x x*
⟨*proof*⟩

definition (**in** *heap-state*)
IO-modify-heap-padding::'*a*::*mem-type ptr* ⇒ ('*s* ⇒ '*a*) ⇒ ('*b*::*default*, *unit*, '*s*) *spec-monad* **where**
IO-modify-heap-padding p v =
state-select {(*s*, *t*). ∃ *bs*. *length bs* = *size-of* (*TYPE*('*a*)) ∧ *t* = *hmem-upd* (*heap-update-padding p* (*v s*) *bs*) *s*}

lemma (**in** *heap-state*) *liftE-IO-modify-heap-padding*: *liftE* (*IO-modify-heap-padding p v*) = (*IO-modify-heap-padding p v*)
⟨*proof*⟩

abbreviation (**in** *heap-state*) *IO-modify-heap-paddingE*::
'*a*::*mem-type ptr* ⇒ ('*s* ⇒ '*a*) ⇒ ('*b*, *unit*, '*s*) *exn-monad* **where**
IO-modify-heap-paddingE p v ≡ *liftE* (*IO-modify-heap-padding p v*)

lemma (**in** *heap-state*) *no-fail-IO-modify-padding*[*simp*]: *succeeds* (*IO-modify-heap-padding p v*) *s*
⟨*proof*⟩

lemma (**in** *heap-state*) *no-fail-IO-modify-paddingE*[*simp*]: *succeeds* (*IO-modify-heap-paddingE p v*) *s*
⟨*proof*⟩

end

theory *LocalVarExtract*
imports *SimplConv L2Defs*
begin

lemma *Collect-prod-inter*:
{(*s*, *t*). *P s t*} ∩ {(*s*, *t*). *Q s t*} = {(*s*, *t*). *P s t* ∧ *Q s t*}
⟨*proof*⟩

lemma *Collect-prod-union*:
{(*s*, *t*). *P s t*} ∪ {(*s*, *t*). *Q s t*} = {(*s*, *t*). *P s t* ∨ *Q s t*}
⟨*proof*⟩

end

theory *AutoCorresSimpset*
imports *SimplBucket*
begin

lemma *globals-surj*: *surj globals*
⟨*proof*⟩

lemma *triv-ex-apply*: ∃ *s1 s0*. *f s0* = *s1*
⟨*proof*⟩

lemmas *ptr-val-ptr-add-simps* =
ptr-add-word32

ptr-add-word32-signed
ptr-add-word64
ptr-add-word64-signed

⟨*ML*⟩

end

theory *ExceptionRewrite*
imports *L1Defs L1Peephole*
begin

definition *always-fail P A* ≡ ∀ *s*. *P s* ⟶ *run A s* = *Failure*

lemma *always-fail-fail* [*simp*]: *always-fail P fail*
⟨*proof*⟩

lemma *bindE-alwaysfail-lhs*: ⟦ *always-fail* (λ-. *True*) *L* ⟧ ⟹ *always-fail* (λ-. *True*) (*L* >>= *R*)
⟨*proof*⟩

lemma *bindE-alwaysfail-rhs*: ⟦ *always-progress L*; *no-throw* (λ-. *True*) *L*; ⋀*x*. *always-fail* (λ-. *True*) (*R x*) ⟧ ⟹ *always-fail* (λ-. *True*) (*L* >>= *R*)
⟨*proof*⟩

lemma *handleE′-alwaysfail-lhs*: ⟦ *always-fail* (λ-. *True*) *L* ⟧ ⟹ *always-fail* (λ-. *True*) (*L* <*catch*> *R*)
⟨*proof*⟩

lemma *handleE′-alwaysfail-rhs*: ⟦ *always-progress L*; *no-return* (λ-. *True*) *L*; ⋀*r*. *always-fail* (λ-. *True*) (*R r*) ⟧ ⟹ *always-fail* (λ-. *True*) (*L* <*catch*> *R*)
⟨*proof*⟩

lemma *alwaysfail-noreturn*: *always-fail P A* ⟹ *no-return P A*
⟨*proof*⟩

lemma *alwaysfail-nothrow*: *always-fail P A* ⟹ *no-throw P A*
⟨*proof*⟩

lemma *no-return-Success-distrib-aux1*:
assumes ∀ (*a* ,*b*) ∈ *X*. ∃ *e*. *a* = *Exception e* ∧ *e* ≠ *default*
shows *X* = *apfst* (*Exception o the-Exception*) ‘ *X* ∧ (∀ *a* ∈ *fst* ‘ *X*. *the-Exception*

$a \neq default$)
⟨*proof*⟩

lemma *no-return-Success-distrib-aux2*:
assumes $\forall (a\ ,b) \in X.\ \exists e.\ a = Exception\ e \wedge e \neq default$
shows $P\ (\bigsqcup x{\in}X.$
case x of $(Exception\ e,\ t) \Rightarrow f\ e\ t$
$|\ (Result\ v,\ t) \Rightarrow g\ v\ t) \longleftrightarrow$
$(P\ (\bigsqcup (e,\ t) \in apfst\ (the\text{-}Exception)\ `\ X.$
$f\ e\ t))$
⟨*proof*⟩

lemma *no-return-Success-distrib-aux3*:
assumes $\forall (a\ ,b) \in X.\ \exists e.\ a = Exception\ e \wedge e \neq default$
shows $P\ (\bigsqcup x{\in}X.$
case x of $(Exception\ e,\ t) \Rightarrow f\ e\ t$
$|\ (Result\ v,\ t) \Rightarrow g\ v\ t) \longleftrightarrow$
$(X = apfst\ (Exception\ o\ the\text{-}Exception)\ `\ X \wedge P\ (\bigsqcup (e,\ t) \in apfst$
$(the\text{-}Exception)\ `\ X.$
$f\ e\ t))$
⟨*proof*⟩

lemma *no-return-bindE*:
no-return $(\lambda\text{-}.\ True)\ A \Longrightarrow (A >>= B) = A$
⟨*proof*⟩

lemma *L1-skip-always-progress* [*simp*, *L1except*, *always-progress-intros*]:
always-progress L1-skip
⟨*proof*⟩

lemma *L1-modify-always-progress* [*simp*, *L1except*, *always-progress-intros*]:
always-progress (*L1-modify m*)
⟨*proof*⟩

lemma *L1-guard-always-progress* [*simp*,*L1except*, *always-progress-intros*]:
always-progress (*L1-guard g*)
⟨*proof*⟩

lemma *L1-init-always-progress* [*simp*,*L1except*, *always-progress-intros*]:
always-progress (*L1-init v*)
⟨*proof*⟩

lemma *L1-spec-always-progress* [*simp*,*L1except*, *always-progress-intros*]: *always-progress*

(*L1-spec s*)
⟨*proof*⟩

lemma *L1-throw-always-progress* [*simp*,*L1except*,*always-progress-intros*]: *always-progress L1-throw*
⟨*proof*⟩

lemma *L1-fail-always-progress* [*simp*,*L1except*, *always-progress-intros*]: *always-progress L1-fail*
⟨*proof*⟩

lemma *L1-condition-always-progress*[*always-progress-intros*]:
⟦ *always-progress L*; *always-progress R* ⟧ ⟹ *always-progress* (*L1-condition C L R*)
⟨*proof*⟩

lemma *L1-seq-always-progress*[*always-progress-intros*]:
⟦ *always-progress L*; *always-progress R* ⟧ ⟹ *always-progress* (*L1-seq L R*)
⟨*proof*⟩

lemma *L1-catch-always-progress*[*always-progress-intros*]:
⟦ *always-progress L*; *always-progress R* ⟧ ⟹ *always-progress* (*L1-catch L R*)
⟨*proof*⟩

lemma *L1-while-always-progress*[*always-progress-intros*]:
always-progress B ⟹ *always-progress* (*L1-while C B*)
⟨*proof*⟩

lemma *L1-call-always-progress*[*always-progress-intros*]: ⟦ *always-progress b* ⟧ ⟹ *always-progress* (*L1-call a b c d e*)
⟨*proof*⟩

lemma *L1-skip-nothrow* [*simp*,*L1except*]: *no-throw* (λ-. *True*) *L1-skip*
⟨*proof*⟩

lemma *L1-modify-nothrow* [*simp*,*L1except*]: *no-throw* (λ-. *True*) (*L1-modify m*)
⟨*proof*⟩

lemma *L1-guard-nothrow* [*simp*,*L1except*]: *no-throw* (λ-. *True*) (*L1-guard g*)
⟨*proof*⟩

lemma *L1-init-nothrow* [*simp*,*L1except*]: *no-throw* (λ-. *True*) (*L1-init a*)
⟨*proof*⟩

lemma *L1-spec-nothrow* [*simp*,*L1except*]: *no-throw* (λ-. *True*) (*L1-spec a*)

⟨*proof*⟩

lemma *L1-assume-nothrow* [*simp*,*L1except*]: *no-throw* (λ-. *True*) (*L1-assume a*)
⟨*proof*⟩

lemma *L1-fail-nothrow* [*simp*,*L1except*]: *no-throw* (λ-. *True*) *L1-fail*
⟨*proof*⟩

lemma *L1-while-nothrow* [*L1except*]: *no-throw* (λ-. *True*) *B* ⟹ *no-throw* (λ-. *True*) (*L1-while C B*)
⟨*proof*⟩

lemma *L1-catch-nothrow-lhs*: ⟦ *no-throw* (λ-. *True*) *L* ⟧ ⟹ *no-throw* (λ-. *True*) (*L1-catch L R*)
⟨*proof*⟩

lemma *L1-catch-nothrow-rhs*: ⟦ *no-throw* (λ-. *True*) *R* ⟧ ⟹ *no-throw* (λ-. *True*) (*L1-catch L R*)
⟨*proof*⟩

lemma *L1-catch-nothrow-both* [*L1except*]: *no-throw* (λ-. *True*) *L* ∨ *no-throw* (λ-. *True*) *R* ⟹ *no-throw* (λ-. *True*) (*L1-catch L R*)
⟨*proof*⟩

lemma *bind-nothrow-simple*: ⟦ *no-throw* (λ-. *True*) *L*; (⋀*x*. *no-throw* (λ-. *True*) (*R x*)) ⟧ ⟹ *no-throw* (λ-. *True*) (*bind L R*)
⟨*proof*⟩

lemma *L1-seq-nothrow* [*L1except*]: ⟦ *no-throw* (λ-. *True*) *L*; *no-throw* (λ-. *True*) *R* ⟧ ⟹ *no-throw* (λ-. *True*) (*L1-seq L R*)
⟨*proof*⟩

lemma *L1-condition-nothrow* [*L1except*]:
⟦ *no-throw* (λ-. *True*) *L*; *no-throw* (λ-. *True*) *R* ⟧ ⟹ *no-throw* (λ-. *True*) (*L1-condition C L R*)
⟨*proof*⟩

lemma *L1-call-nothrow*[*L1except*]: *no-throw* (λ-. *True*) *y* ⟹ *no-throw* (λ-. *True*) (*L1-call x y z q r*)
⟨*proof*⟩

lemma *no-throw-state-assumeE*: *no-throw* (λ-. *True*) (*assume-result-and-state p*)
⟨*proof*⟩

lemma *no-throw-on-exit*:
assumes *c*: *no-throw* (λ-. *True*) *c*
shows *no-throw* (λ-. *True*) (*on-exit c cleanup*)

⟨*proof*⟩

named-theorems *no-throw-with-fresh-stack-ptr*
lemma (**in** *stack-heap-raw-state*) *no-throw-with-fresh-stack-ptr*[*no-throw-with-fresh-stack-ptr*]:
assumes *c*: ⋀*p*. *no-throw* (λ-. *True*) (*c p*)
shows *no-throw* (λ-. *True*) (*with-fresh-stack-ptr n init c*)
⟨*proof*⟩

lemmas *L1-nothrows* =
L1-seq-nothrow
L1-skip-nothrow
L1-modify-nothrow
L1-condition-nothrow
L1-catch-nothrow-both
L1-while-nothrow
L1-spec-nothrow
L1-assume-nothrow
L1-guard-nothrow
L1-init-nothrow
L1-call-nothrow
L1-fail-nothrow

lemma *L1-throw-noreturn* [*simp*,*L1except*]: *no-return* (λ-. *True*) *L1-throw*
⟨*proof*⟩

lemma *L1-fail-noreturn* [*simp*,*L1except*]: *no-return* (λ-. *True*) *L1-fail*
⟨*proof*⟩

lemma *L1-seq-noreturn-lhs*: *no-return* (λ-. *True*) *L* ⟹ *no-return* (λ-. *True*) (*L1-seq L R*)
⟨*proof*⟩

lemma *L1-seq-noreturn-rhs*: ⟦ *no-return* (λ-. *True*) *R* ⟧ ⟹ *no-return* (λ-. *True*) (*L1-seq L R*)
⟨*proof*⟩

lemma *L1-catch-noreturn*: ⟦ *no-return* (λ-. *True*) *L*; *no-return* (λ-. *True*) *R* ⟧ ⟹ *no-return* (λ-. *True*) (*L1-catch L R*)
⟨*proof*⟩

lemma *L1-condition-noreturn*: ⟦ *no-return* (λ-. *True*) *L*; *no-return* (λ-. *True*) *R* ⟧ ⟹ *no-return* (λ-. *True*) (*L1-condition C L R*)
⟨*proof*⟩

lemma *bindE-noreturn-lhs*: ⟦*no-return* (λ-. *True*) *L* ⟧ ⟹ *no-return* (λ-. *True*) (*L*

$>>= R$)
⟨*proof*⟩

lemma *bindE-noreturn-rhs*: ⟦⋀*x*. *no-return* (λ-. *True*) (*R x*) ⟧ ⟹ *no-return* (λ-. *True*) (*L >>= R*)
⟨*proof*⟩

lemma *on-exit-noreturn*:
assumes *c*: *no-return* (λ-. *True*) *c*
shows *no-return* (λ-. *True*) (*on-exit c cleanup*)
⟨*proof*⟩

named-theorems *no-return-with-fresh-stack-ptr*
lemma (**in** *stack-heap-raw-state*) *no-return-with-fresh-stack-ptr*[*no-return-with-fresh-stack-ptr*]:
assumes *c*: ⋀*p*. *no-return* (λ-. *True*) (*c p*)
shows *no-return* (λ-. *True*) (*with-fresh-stack-ptr n init c*)
⟨*proof*⟩

lemma *L1-fail-alwaysfail* [*simp*,*L1except*]: *always-fail* (λ-. *True*) *L1-fail*
⟨*proof*⟩

lemma *L1-seq-alwaysfail-lhs*: ⟦ *always-fail* (λ-. *True*) *L* ⟧ ⟹ *always-fail* (λ-. *True*) (*L1-seq L R*)
⟨*proof*⟩

lemma *L1-seq-alwaysfail-rhs*: ⟦ *always-progress L*; *no-throw* (λ-. *True*) *L*; *always-fail* (λ-. *True*) *R* ⟧ ⟹ *always-fail* (λ-. *True*) (*L1-seq L R*)
⟨*proof*⟩

lemma *L1-catch-alwaysfail-lhs*: ⟦ *always-fail* (λ-. *True*) *L* ⟧ ⟹ *always-fail* (λ-. *True*) (*L1-catch L R*)
⟨*proof*⟩

lemma *L1-catch-alwaysfail-rhs*: ⟦ *always-progress L*; *no-return* (λ-. *True*) *L*; *always-fail* (λ-. *True*) *R* ⟧
⟹ *always-fail* (λ-. *True*) (*L1-catch L R*)
⟨*proof*⟩

lemma *L1-condition-alwaysfail*: ⟦ *always-fail* (λ-. *True*) *L*; *always-fail* (λ-. *True*) *R* ⟧ ⟹ *always-fail* (λ-. *True*) (*L1-condition C L R*)
⟨*proof*⟩

lemma *L1-catch-nothrow* []:

no-throw (λ-. *True*) *A* ⟹ *L1-catch A E* = *A*
⟨*proof*⟩

lemma *L1-seq-noreturn* [*L1except*]:
no-return (λ-. *True*) *A* ⟹ *L1-seq A B* = *A*
⟨*proof*⟩

lemma *L1-catch-throw* [*L1except*]:
L1-catch L1-throw E = *E*
⟨*proof*⟩

lemma *anything-to-L1-fail* [*L1except*]:
always-fail (λ-. *True*) *A* ⟹ *A* = *L1-fail*
⟨*proof*⟩

lemmas *L1-is-local-simps* [*L1except*] = *is-local-simps*

lemma *L1-catch-return* [*L1except*]: (⋀*s*. *is-local* (*get-exn* (*upd-exn s*))) ⟹
(*L1-seq*
(*L1-modify* (*upd-exn*))
(*L1-condition* (λ*s*. *is-local* (*get-exn s*))
L1-skip L1-throw))
= *L1-modify* (*upd-exn*)
⟨*proof*⟩

lemma *L1-catch-L1-seq-nothrow* [*L1except*]:
⟦*no-throw* (λ-. *True*) *A* ⟧ ⟹ *L1-catch* (*L1-seq A B*) *C* = *L1-seq A* (*L1-catch B C*)
⟨*proof*⟩

lemma *L1-catch-simple-seq* [*L1except*]:
L1-catch (*L1-seq L1-skip B*) *E* = (*L1-seq L1-skip* (*L1-catch B E*))
L1-catch (*L1-seq L1-fail B*) *E* = (*L1-seq L1-fail* (*L1-catch B E*))
L1-catch (*L1-seq* (*L1-modify m*) *B*) *E* = (*L1-seq* (*L1-modify m*) (*L1-catch B E*))
L1-catch (*L1-seq* (*L1-spec s*) *B*) *E* = (*L1-seq* (*L1-spec s*) (*L1-catch B E*))
L1-catch (*L1-seq* (*L1-assume f*) *B*) *E* = (*L1-seq* (*L1-assume f*) (*L1-catch B E*))
L1-catch (*L1-seq* (*L1-guard g*) *B*) *E* = (*L1-seq* (*L1-guard g*) (*L1-catch B E*))
L1-catch (*L1-seq* (*L1-init i*) *B*) *E* = (*L1-seq* (*L1-init i*) (*L1-catch B E*))
⟨*proof*⟩

declare *L1-catch-simple-seq*(*6*) [*L1opt*]
lemma *L1-catch-L1-init-seq'*[*L1opt*, *L1except*]: *L1-catch* (*L1-seq* (*L1-seq* (*L1-init*

i) *A*) *B*) *E* = (*L1-seq* (*L1-init i*) (*L1-catch* (*L1-seq A B*) *E*))
⟨*proof*⟩

lemma *L1-catch-call-simple-seq* [*L1except*]:
no-throw (λ-. *True*) *b* ⟹ *L1-catch* (*L1-seq* (*L1-call a b c d e*) *B*) *E* = (*L1-seq* (*L1-call a b c d e*) (*L1-catch B E*))
⟨*proof*⟩

lemma *L1-catch-single* [*L1except*]:
L1-catch (*L1-skip*) *E* = *L1-skip*
L1-catch (*L1-fail*) *E* = *L1-fail*
L1-catch (*L1-modify m*) *E* = *L1-modify m*
L1-catch (*L1-spec s*) *E* = *L1-spec s*
L1-catch (*L1-assume f*) *E* = *L1-assume f*
L1-catch (*L1-guard g*) *E* = *L1-guard g*
L1-catch (*L1-init i*) *E* = *L1-init i*
⟨*proof*⟩

lemma *L1-catch-call-single* [*L1except*]: *no-throw* (λ-. *True*) *b* ⟹ *L1-catch* (*L1-call a b c d e*) *E* = *L1-call a b c d e*
⟨*proof*⟩

lemma *L1-catch-single-while* [*L1except*]:
⟦ *no-throw* (λ-. *True*) *B* ⟧ ⟹ *L1-catch* (*L1-while C B*) *E* = *L1-while C B*
⟨*proof*⟩

lemma *L1-catch-seq-while* [*L1except*]:
⟦ *no-throw* (λ-. *True*) *B* ⟧ ⟹ *L1-catch* (*L1-seq* (*L1-while C B*) *X*) *E* = *L1-seq* (*L1-while C B*) (*L1-catch X E*)
⟨*proof*⟩

lemma *L1-catch-single-cond* [*L1except*]:
L1-catch (*L1-condition C L R*) *E* = *L1-condition C* (*L1-catch L E*) (*L1-catch R E*)
⟨*proof*⟩

lemma *L1-catch-cond-seq*:
L1-catch (*L1-seq* (*L1-condition C L R*) *B*) *E* = *L1-condition C* (*L1-catch* (*L1-seq L B*) *E*) (*L1-catch* (*L1-seq R B*) *E*)
⟨*proof*⟩

lemma *L1-catch-seq-cond-noreturn-ex*:
⟦ *no-return* (λ-. *True*) *E* ⟧ ⟹ (*L1-catch* (*L1-seq* (*L1-condition c A B*) *C*) *E*) = (*L1-seq* (*L1-catch* (*L1-condition c A B*) *E*) (*L1-catch C E*))
⟨*proof*⟩

lemmas *L1-catch-seq-cond-nothrow* = *L1-catch-L1-seq-nothrow* [*OF L1-condition-nothrow*]

end

19.2 Nested Exceptions

theory *L2ExceptionRewrite*
imports
L2Defs
ExceptionRewrite
begin

synthesize-rules *L2-rel-spec-monad*

19.3 Transformations from single level exceptions to nested exceptions

lemma *catch-yield-map-value-conv*: (*f* <*catch*> (λ*e*. *yield* (*g e*))) =
(*map-value* (λ*x*. *case x of Exn e* ⇒ *g e* | *Result v* ⇒ *Result v*) *f*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-xval-try-catch*:
assumes *bdy*: *rel-spec-monad Q* (*rel-xval* (λ*e e'*. (*rel-sum L R*) (*from-xval* (*f e*)) *e'*) *R*) *B B'*
shows *rel-spec-monad Q* (*rel-xval L R*) (*L2-catch B* (λ*e*. *yield* (*f e*))) (*L2-try B'*)
⟨*proof*⟩

lemma *rel-spec-monad-L2-seq-rel-xval-result-eq*:
assumes *f-f'*: *rel-spec-monad S* (*rel-xval L* (=)) *f f'*
assumes *Res-Res*: ⋀*v*. *rel-spec-monad S* (*rel-xval L* (=)) (*g v*) (*g' v*)
shows *rel-spec-monad S* (*rel-xval L* (=)) (*L2-seq f g*) (*L2-seq f' g'*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-xval-L2-unknown*:
rel-spec-monad (=) (*rel-xval L* (=)) (*L2-unknown ns*) (*L2-unknown ns*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-xval-L2-modify*:
rel-spec-monad (=) (*rel-xval L* (=)) (*L2-modify f*) (*L2-modify f*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-xval-L2-gets*:
rel-spec-monad (=) (*rel-xval L* (=)) (*L2-gets f ns*) (*L2-gets f ns*)
⟨*proof*⟩

lemma *rel-spec-monad-eq-rel-xval-L2-condition*:

assumes *f*: *rel-spec-monad* (=) (*rel-xval L* (=)) *f f′*
and *g*: *rel-spec-monad* (=) (*rel-xval L* (=)) *g g′*
shows *rel-spec-monad* (=) (*rel-xval L* (=)) (*L2-condition P f g*) (*L2-condition P f′ g′*)
⟨*proof*⟩

lemma *rel-spec-monad-L2-throw-sanitize-names*:
assumes *xy*: *R* (*Exn x*) (*Exn y*)
assumes *ns′*: *SANITIZE-NAMES y ns ns′*
shows *rel-spec-monad* (=) *R* (*L2-throw x ns*) (*L2-throw y ns′*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-xval-L2-spec*:
rel-spec-monad (=) (*rel-xval L* (=)) (*L2-spec r*) (*L2-spec r*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-xval-L2-assume*:
rel-spec-monad (=) (*rel-xval L* (=)) (*L2-assume r*) (*L2-assume r*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-xval-L2-guard*:
rel-spec-monad (=) (*rel-xval L* (=)) (*L2-guard c*) (*L2-guard c*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-xval-L2-guarded*:
assumes *c-c′*: *rel-spec-monad* (=) (*rel-xval L* (=)) *c c′*
shows *rel-spec-monad* (=) (*rel-xval L* (=)) (*L2-guarded g c*) (*L2-guarded g c′*)
⟨*proof*⟩

lemma *rel-spec-monad-L2-fail*:
rel-spec-monad Q R (*L2-fail*) (*L2-fail*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-xval-L2-call*:
assumes *emb*: ⋀*e*. *L* (*emb e*) (*emb′ e*)
assumes *ns′*: *SANITIZE-NAMES emb′ ns ns′*
shows *rel-spec-monad* (=) (*rel-xval L* (=)) (*L2-call f emb ns*) (*L2-call f emb′ ns′*)
⟨*proof*⟩

lemma *rel-fun-eq-refl*: *rel-fun* (=) (=) *f f*
⟨*proof*⟩

lemma *rel-spec-monad-result-exec-concrete*:
assumes *m-m′*: *rel-spec-monad* (=) *R m m′*
shows *rel-spec-monad* (=) *R* (*exec-concrete st m*) (*exec-concrete st m′*)
⟨*proof*⟩

lemma *rel-spec-monad-result-exec-abstract*:

assumes *m-m'*: *rel-spec-monad* (=) *R m m'*
shows *rel-spec-monad* (=) *R* (*exec-abstract st m*) (*exec-abstract st m'*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-project-L2-unknown'*:
assumes *surj-prj*: *surj prj*
assumes *names*: *SANITIZE-NAMES prj ns ns'*
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*L2-unknown ns*) (*L2-unknown ns'*)
⟨*proof*⟩

Note that *prj* is the identity function on unit here

lemma *rel-spec-monad-rel-project-L2-modify*:
rel-spec-monad (=) (*rel-xval L* (*rel-project prj*)) (*L2-modify f*) (*L2-modify f*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-project-L2-gets'*:
assumes *g*: ⋀*x*. *g x* = *prj* (*f x*)
assumes *names*: *SANITIZE-NAMES prj ns ns'*
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*L2-gets f ns*) (*L2-gets g ns'*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-project-L2-condition*:
assumes *f*: *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) *f f'*
assumes *g*: *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) *g g'*
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*L2-condition c f g*) (*L2-condition c f' g'*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-project-L2-throw*:
assumes *x-xs*: *L x y*
assumes *names*: *SANITIZE-NAMES* (*L*, *x*) *ns ns'*
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*L2-throw x ns*) (*L2-throw y ns'*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-project-L2-spec*:
assumes *prj-surj*: *surj prj*
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*L2-spec r*) (*L2-spec r*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-project-L2-assume*:
shows *rel-spec-monad* (=) (*rel-xval L* (=)) (*L2-assume r*) (*L2-assume r*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-project-L2-guard*:
rel-spec-monad (=) (*rel-xval L* (*rel-project prj*)) (*L2-guard c*) (*L2-guard c*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-project-L2-seq*:
assumes *m-n*: *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj'*)) *m n*
assumes *f-g*: ⋀*x*. *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*f x*) (*g* (*prj' x*))
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*L2-seq m f*) (*L2-seq n g*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-project-L2-guarded*:
assumes *c-c'*: *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) *c c'*
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*L2-guarded g c*) (*L2-guarded g c'*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-project-L2-try*:
assumes *fg*: *rel-spec-monad* (=) (*rel-xval* (*rel-sum L* (*rel-project prj*)) (*rel-project prj*)) *f g*
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*L2-try f*) (*L2-try g*)
⟨*proof*⟩

Tailored projection for local handler functions that may emerge in IO phase to handle exit case. These handlers are composed from liftE functions followed by rethrowing the exception. We do not attempt to optimise projections for the handlers.

lemma *rel-spec-monad-rel-project-L2-catch*:
assumes *f*: *rel-spec-monad* (=) (*rel-xval* (=) (*rel-project prj*)) *f f'*
assumes *g*: ⋀*v*. (*rel-spec-monad* (=) (*rel-xval L* (λ- -. *False*))) (*g v*) (*g v*)
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*L2-catch f g*) (*L2-catch f' g*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-project-liftE*:
assumes *f*: *rel-spec-monad* (=) (*rel-map the-Res OO rel-project prj OO rel-map Result*) *f f'*
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*liftE f*) (*liftE f'*)
⟨*proof*⟩

lemma *rel-project-Res-conv*:
(*rel-map the-Res OO rel-project prj OO rel-map Result*) = (*rel-project* (*Result o prj o the-Res*))
⟨*proof*⟩

lemma *rel-spec-monad-rel-project-id*:
shows *rel-spec-monad* (=) (*rel-project* (λ*v*. *v*)) *f f*
⟨*proof*⟩

lemma *rel-spec-monad-rel-project-unit*:
fixes *f*:: (*unit*, *'s*) *res-monad*
shows *rel-spec-monad* (=) (*rel-project* (λ*v*. *Result* ())) *f f*
⟨*proof*⟩

lemma *rel-spec-monad-rel-project-liftE-unit-id*:
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project* (*prj*::*unit* ⇒ *unit*))) (*liftE f*) (*liftE f*)
⟨*proof*⟩

lemma *rel-project-unit-eq*: (*rel-project* (*prj*::*unit* ⇒ *unit*)) = (=)
⟨*proof*⟩

lemma *rel-spec-monad-L2-seq-rel-xval-same-split*:
assumes *mn*: *rel-spec-monad R* (*rel-xval L* (=)) *m n*
and *fg*: ⋀*x*. (*rel-spec-monad R* (*rel-xval L* (=))) (*f x*) (*g x*)
shows *rel-spec-monad R* (*rel-xval L* (=)) (*L2-seq m f*) (*L2-seq n g*)
⟨*proof*⟩

lemma *rel-spec-monad-L2-while-rel-xval-same-split*:
assumes *B*: ⋀*x*. (*rel-spec-monad* (=) (*rel-xval L* (=))) (*B x*) (*B' x*)
shows *rel-spec-monad* (=) (*rel-xval L* (=)) (*L2-while C' B I' ns*) (*L2-while C' B' I' ns*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-project-L2-call-adapt-emb*:
assumes *L*: ⋀*x*. *L* (*emb x*) (*emb' x*)
assumes *prj*: ⋀*v*. *prj v* = *v*
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*L2-call x emb ns*) (*L2-call x emb' ns*)
⟨*proof*⟩

lemma *rel-spec-monad-liftE-id*: *rel-spec-monad* (=) (*rel-xval* (λ- -. *False*) (=)) (*liftE f*) (*liftE f*)
⟨*proof*⟩

lemma *rel-spec-monad-L2-seq-exit-handler*:
assumes ⋀*v*. *rel-spec-monad* (=) (*rel-xval L* (λ- -. *False*)) (*g v*) (*g' v*)
shows *rel-spec-monad* (=) (*rel-xval L* (λ- -. *False*)) (*L2-seq* (*liftE f*) *g*) (*L2-seq* (*liftE f*) *g'*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-project-L2-while'*:
assumes *C-C'*: ⋀*v s*. *C v s* = *C'* (*prj v*) *s*
assumes *I'*: *I'* = *prj I*
assumes *names*: *SANITIZE-NAMES prj ns ns'*

assumes *B-B'*: $\bigwedge v$. *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*B v*) (*B'* (*prj v*))
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*L2-while C B I ns*) (*L2-while C' B' I' ns'*)
⟨*proof*⟩

lemma *rel-spec-monad-rel-xval-on-exit*:
assumes *c-c'*: *rel-spec-monad* (=) (*rel-xval L* (=)) *c c'*
shows *rel-spec-monad* (=) (*rel-xval L* (=)) (*on-exit c cleanup*) (*on-exit c' cleanup*)
⟨*proof*⟩

lemma (**in** *stack-heap-raw-state*) *rel-spec-monad-rel-xval-with-fresh-stack-ptr*:
assumes *c*: $\bigwedge p$. *rel-spec-monad* (=) (*rel-xval L* (=)) (*c p*) (*c' p*)
shows *rel-spec-monad* (=) (*rel-xval L* (=)) (*with-fresh-stack-ptr n init* (*L2-VARS c nm*)) (*with-fresh-stack-ptr n init* (*L2-VARS c' nm*))
⟨*proof*⟩

lemma *rel-spec-monad-rel-project-on-exit*:
assumes *c-c'*: *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) *c c'*
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*on-exit c cleanup*) (*on-exit c' cleanup*)
⟨*proof*⟩

lemma (**in** *stack-heap-raw-state*) *rel-spec-monad-rel-project-with-fresh-stack-ptr*:
assumes *c*: $\bigwedge p$. *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*c p*) (*c' p*)
shows *rel-spec-monad* (=) (*rel-xval L* (*rel-project prj*)) (*with-fresh-stack-ptr n init* (*L2-VARS c nm*)) (*with-fresh-stack-ptr n init* (*L2-VARS c' nm*))
⟨*proof*⟩

lemma *rel-spec-monad-L2-VARS*:
assumes *f-f'*: *rel-spec-monad P Q f f'*
shows *rel-spec-monad P Q* (*L2-VARS f ns*) (*L2-VARS f' ns*)
⟨*proof*⟩

lemma *cond-return1*: (λ*a*.
L2-condition (λ*s*. *P a*) (*L2-gets* (λ*s*. *f a*) *ns*)
(*L2-throw* (*g a*) *xs*)) =
(λ*a*. *yield* (*if P a then Result* (*f a*) *else* (*Exn* (*g a*))))
⟨*proof*⟩

lemma *cond-return2*: (λ(*a*, *b*).
L2-condition (λ*s*. *P a b*) (*L2-gets* (λ*s*. *f a b*) *ns*)
(*L2-throw* (*g a b*) *xs*)) =
(λ(*a*, *b*). *yield* (*if P a b then Result* (*f a b*) *else* (*Exn* (*g a b*))))
⟨*proof*⟩

lemma *rel-spec-monad-rel-xvalI*:

rel-spec-monad R (rel-xval (=) (=)) f g ⟹ rel-spec-monad R (=) f g
⟨*proof*⟩

lemma *is-local-split*: ((*is-local a ⟶ e = Result b*) ∧ (¬ *is-local a ⟶ e = Exn* (*the-Nonlocal a*)))
=
(*case a of Nonlocal x ⇒ e = Exn x* | - ⇒ *e = Result b*)
⟨*proof*⟩

lemma *is-local-splitI* :
(*case a of Nonlocal x ⇒ e = Exn x* | - ⇒ *e = Result b*) ⟹
((*is-local a ⟶ e = Result b*) ∧ (¬ *is-local a ⟶ e = Exn* (*the-Nonlocal a*)))
⟨*proof*⟩

lemma *if-is-local-cases*: (*if is-local e then f else g*) = (*case e of Nonlocal x ⇒ g* | - ⇒ *f*)
⟨*proof*⟩

lemma *if-Break-cases*: (*if e = Break then f else g*) = (*case e of Break ⇒ f* | - ⇒ *g*)
⟨*proof*⟩

lemma *if-Continue-cases*: (*if e = Continue then f else g*) = (*case e of Continue ⇒ f* | - ⇒ *g*)
⟨*proof*⟩

lemma *if-Return-cases*: (*if e = Return then f else g*) = (*case e of Return ⇒ f* | - ⇒ *g*)
⟨*proof*⟩

lemmas *if-c-exntype-cases* =
if-is-local-cases if-Break-cases if-Continue-cases if-Return-cases

lemmas *case-sum-c-exntype-swap* = *c-exntype.case-distrib*

lemma *ex-c-enxtype-cases-distrib*: (∃ *a*. *P a* ∧
(*case e of Break ⇒ brk a* | *Continue ⇒ cnt a* | *Return ⇒ ret a* | *Goto l ⇒ goto l a*
| *Nonlocal gx ⇒ nonlocal gx a*)) =
(*case e of Break* ⇒ ∃ *a*. *P a* ∧ *brk a* | *Continue* ⇒ ∃ *a*. *P a* ∧ *cnt a* | *Return* ⇒ ∃ *a*. *P a* ∧ *ret a*
| *Goto l* ⇒ ∃ *a*. *P a* ∧ *goto l a*

| *Nonlocal gx* ⇒ ∃ *a*. *P a* ∧ *nonlocal gx a*)

⟨*proof*⟩

19.4 Transformations for procedure local exceptions

Introduce constant for relations to avoid higher order patterns in term net for rules

19.4.1 Transformations for *try* and *finally*

19.5 Transformations on global exceptions

definition
lift-exit-status :: (′*e*, ′*a*, ′*s*) *exn-monad* ⇒ (′*e c-exntype*, ′*a*, ′*s*) *exn-monad*
where
lift-exit-status f ≡ *map-value* (*map-exn Nonlocal*) *f*

19.5.1 Removing unused tuple components

lemma *rel-project-eqI*: *rel-spec-monad* (=) (*rel-xval* (=) (*rel-project* (*ETA-TUPLED* (λ*v*. *v*)))) *f g* ⟹ *f* = *g*
⟨*proof*⟩

19.5.2 Setup basic rules

lemma *rel-xval-case-Nonlocal-sameI*:
assumes *L*: ⋀*v*. *e* = *Nonlocal v* ⟹ *L* (*Nonlocal v*) (*Nonlocal v*)
assumes *R*: *is-local e* ⟹ *R v v*′
shows *rel-xval L R* (*case e of Nonlocal x* ⇒ *Exn* (*Nonlocal x*) | - ⇒ *Result v*)
(*case e of Nonlocal x* ⇒ *Exn* (*Nonlocal x*) | - ⇒ *Result v*′)
⟨*proof*⟩

lemma *rel-sum-map-sum-InlI*: *L* (*l x*) *v* ⟹ *rel-sum L R* (*map-sum l r* (*Inl x*)) (*Inl v*)
⟨*proof*⟩

lemma *rel-sum-map-sum-InrI*: *R* (*r x*) *v* ⟹ *rel-sum L R* (*map-sum l r* (*Inr x*)) (*Inr v*)
⟨*proof*⟩

lemma *rel-map-xval-xval-ExnI*: *L* (*l x*) *v* ⟹ *rel-xval L R* (*map-xval l r* (*Exn x*)) (*Exn v*)
⟨*proof*⟩

lemma *rel-map-xval-xval-ResultI*: *R* (*r x*) *v* ⟹ *rel-xval L R* (*map-xval l r* (*Result x*)) (*Result v*)

⟨*proof*⟩

⟨*ML*⟩

declare [[*verbose=3*]]
⟨*ML*⟩

context *stack-heap-raw-state*
begin
⟨*ML*⟩
end
declare [[*verbose=0*]]

lemma *map-sum-right*: (*map-sum l r v* = *Inr x*) = (∃ *v'*. *v* = *Inr v'* ∧ *x* = *r v'*)
⟨*proof*⟩

lemma *map-sum-left*: (*map-sum l r v* = *Inl x*) = (∃ *v'*. *v* = *Inl v'* ∧ *x* = *l v'*)
⟨*proof*⟩

lemma *case-Nonlocal-Inr*: ((*case e of Nonlocal x* ⇒ *Inl x* | - ⇒ *Inr v*) = *Inr y*) = (*is-local e* ∧ (*v* = *y*))
⟨*proof*⟩

lemma *case-Nonlocal-Inl*: ((*case e of Nonlocal x* ⇒ *Inl x* | - ⇒ *Inr v*) = *Inl y*) = (*e* = *Nonlocal y*)
⟨*proof*⟩

⟨*ML*⟩

method *rel-spec-monad-L2-step* **uses** *more-intro-thms* =
((*rule-tac more-intro-thms*, *verbose-msg applied*: *rule-tac*) |
trace resolve-split applied: *resolve-split* |
trace assumption applied: *assumption* |
trace sanitize-names applied: *sanitize-names* |
trace clarsimp-solve applied: *clarsimp-solve*
)

method *rel-spec-monad-L2-rewrite* =
(*use* **in** ‹*rel-spec-monad-L2-step more-intro-thms*: *method-facts*›)+

⟨*ML*⟩

end

19.6 Peep-hole optimisations applied to L2

theory *L2Peephole*
imports *L2Defs Tuple-Tools*
begin

lemmas *id-def* [*L2opt*]

lemma *L2-seq-skip* [*L2opt*]:
L2-seq (*L2-gets* (λ-. ()) *n*) *X* = (*X* ())
L2-seq Y (λ-. (*L2-gets* (λ-. ()) *n*)) = *Y*
⟨*proof*⟩

lemma *L2-seq-L2-gets* [*L2opt*]: *L2-seq X* (λ*x*. *L2-gets* (λ-. *x*) *n*) = *X*
⟨*proof*⟩

lemma *L2-seq-L2-gets-unit*[*L2opt*]: *L2-seq* (*L2-gets g ns*) (λ*x*:: *unit*. *f x*) = *f* ()
⟨*proof*⟩

lemma *L2-seq-L2-gets-const*: *L2-seq* (*L2-gets* (λ-. *c*) *n*) *X* = *X c*
⟨*proof*⟩

lemma *const-propagation-cong*: *X c* = *X′ c* ⟹ (*L2-seq* (*L2-gets* (λ-. *c*) *n*) *X*) = (*L2-seq* (*L2-gets* (λ-. *c*) *n*) *X′*)
⟨*proof*⟩

lemma *L2-seq-const′*:
assumes *bdy-eq*: *f c* ≡ *g c*
shows *L2-seq* (*L2-gets* (λ-. *c*) *n*) *f* ≡ *L2-seq* (*L2-gets* (λ-. *c*) *n*) *g*
⟨*proof*⟩

lemma *L2-seq-const*:
assumes *bdy-eq*: *f′* ≡ *g′*
assumes *f-app*: *f c* ≡ *f′*
assumes *g-app*: *g c* ≡ *g′*
shows *L2-seq* (*L2-gets* (λ-. *c*) *n*) *f* ≡ *L2-seq* (*L2-gets* (λ-. *c*) *n*) *g*
⟨*proof*⟩

lemma *L2-seq-const-stop*:
assumes *bdy-eq*: *f′* ≡ *g′*
assumes *f-app*: *f c* ≡ *f′*
assumes *g-app*: *g c* ≡ *g′*
shows *L2-seq* (*L2-gets* (λ-. *c*) *n*) *f* ≡ *STOP* (*L2-seq* (*L2-gets* (λ-. *c*) *n*) *g*)
⟨*proof*⟩

lemma *L2-seq-const-stop′*:

assumes *bdy-eq*: $f\ c \equiv g'$
assumes *g-app*: $g\ c \equiv g'$
shows *L2-seq* (*L2-gets* (λ-. c) n) $f \equiv$ *STOP* (*L2-seq* (*L2-gets* (λ-. c) n) g)
⟨*proof*⟩

lemma *L2-seq-const-stop''*:
assumes *bdy-eq*: $f\ c \equiv g\ c$
shows *L2-seq* (*L2-gets* (λ-. c) n) $f \equiv$ *STOP* (*L2-seq* (*L2-gets* (λ-. c) n) g)
⟨*proof*⟩

lemma *L2-seq-const-stop'''*:
assumes *bdy-eq*: $f\ c \equiv g$
shows *L2-seq* (*L2-gets* (λ-. c) n) $f \equiv$ *STOP* (*L2-seq* (*L2-gets* (λ-. c) n) (λ-. g))
⟨*proof*⟩

lemma *L2-marked-seq-gets-cong*:
$c{=}c' \Longrightarrow$ *L2-seq-gets* $c\ n\ A \equiv$ *L2-seq-gets* $c'\ n\ A$
⟨*proof*⟩

lemma *L2-marked-seq-gets-stop*:
assumes *bdy-eq*: $f\ c \equiv g$
shows *L2-seq-gets* $c\ n\ f \equiv$ *STOP* (*L2-seq-gets* $c\ n$ (λ-. g))
⟨*proof*⟩

lemma *L2-marked-seq-gets-stop'*:
assumes *bdy-eq*: $f\ c \equiv g\ c$
shows *L2-seq-gets* $c\ n\ f \equiv$ *STOP* (*L2-seq-gets* $c\ n\ g$)
⟨*proof*⟩

lemma *L2-marked-seq-gets-stop''*:
assumes *bdy-eta*: $f \equiv f'$
assumes *bdy-eq*: $\bigwedge v.\ v = c \Longrightarrow f'\ v \equiv g\ v$
shows *L2-seq-gets* $c\ n\ f \equiv$ *STOP* (*L2-seq-gets* $c\ n\ g$)
⟨*proof*⟩

lemma *L2-guarded-block-cong*: *L2-guarded* $g\ c =$ *L2-guarded* $g\ c$
⟨*proof*⟩

lemma *L2-guarded-cong-stop'*:
assumes *guard-eq*: $\bigwedge s.\ g\ s \equiv g'\ s$
assumes *bdy-eq*: $\bigwedge s.\ g'\ s \Longrightarrow$ *run* $c\ s \equiv$ *run* $c'\ s$
shows *L2-guarded* $g\ c \equiv$ *STOP* (*L2-guarded* $g'\ c'$)
⟨*proof*⟩

lemma *L2-seq-guard-cong-stop0*:
assumes *guard-eq*: $\bigwedge s.\ g\ s \equiv g'\ s$
assumes *bdy-eq*: $\bigwedge s.\ g'\ s \Longrightarrow$ *run* (c ()) $s \equiv$ *run* $c'\ s$
shows *L2-seq-guard* $g\ c \equiv$ *STOP* (*L2-seq-guard* g' (λ-. c'))

⟨*proof*⟩

lemma *L2-guarded-cong-stop*:
assumes *guard-eq*: $g \equiv g'$
assumes *bdy-eq*: $\bigwedge s.\ g'\ s \Longrightarrow run\ c\ s \equiv run\ c'\ s$
shows *L2-guarded g c* ≡ *STOP* (*L2-guarded g' c'*)
⟨*proof*⟩

lemma *L2-seq-assoc* :
L2-seq (*L2-seq A* (λ*x*. *B x*)) *C* = *L2-seq A* (λ*x*. *L2-seq* (*B x*) *C*)
⟨*proof*⟩

lemma *L2-seq-assoc'* [*L2opt*]:
L2-seq (*L2-seq A B*) *C* = *L2-seq A* (λ*x*. *L2-seq* (*B x*) *C*)
⟨*proof*⟩

lemma *L2-seq-rev-assoc'*:
L2-seq A (λ*x*. (*L2-seq* (*B x*) (*C x*))) =
L2-seq (*L2-seq A* (λ*x*. *L2-seq* (*B x*) (λ*y*. *L2-gets* (λ-. (*x*, *y*)) *ns*))) (λ(*x*, *y*). *C x y*)
⟨*proof*⟩

lemma *L2-seq-rev-assoc-accumulated'*:
L2-seq A (λ(*a*, *x*). (*L2-seq* (*B a x*) (*C a x*))) =
L2-seq (*L2-seq A* (λ(*a*, *x*). *L2-seq* (*B a x*) (λ*y*. *L2-gets* (λ-. (*a*, *x*, *y*)) *ns*))) (λ(*a*, *x*, *y*). *C a x y*)
⟨*proof*⟩

lemma *L2-seq-rev-assoc*:
L2-seq A (λ*a*. (*L2-seq* (*B a*) (*C a*))) =
L2-seq (*L2-seq A* (λ*a*. *L2-seq* (*B a*) (λ*x*. *L2-gets* (λ-. (*x*, *a*)) *ns*))) (λ(*x*, *a*). *C a x*)
⟨*proof*⟩

lemma *L2-seq-rev-assoc-accumulated*:
L2-seq A (λ(*x*, *a*). (*L2-seq* (*B a x*) (*C a x*))) =
L2-seq (*L2-seq A* (λ(*x*, *a*). *L2-seq* (*B a x*) (λ*y*. *L2-gets* (λ-. (*y*, *x*, *a*)) *ns*))) (λ(*y*, *x*, *a*). *C a x y*)
⟨*proof*⟩

lemma *L2-trim-after-throw* [*L2opt*]:
L2-seq (*L2-throw x n*) *Z* = (*L2-throw x n*)
⟨*proof*⟩

lemma *L2-guard-true* [*L2opt*]: *L2-guard* (λ-. *True*) = *L2-gets* (λ-. ()) []
⟨*proof*⟩

This rule can be too expensive in simplification as it might invoke the

arithmetic solver

lemma *L2-guard-solve-true*: $[\![\bigwedge s.\ P\ s\ ]\!] \Longrightarrow$ *L2-guard* P = *L2-gets* (λ-. ()) []
⟨*proof*⟩

lemma *L2-guard-false* [*L2opt*]: *L2-guard* (λ-. *False*) = *L2-fail*
⟨*proof*⟩

This rule can be too expensive in simplification as it might invoke the arithmetic solver

lemma *L2-guard-solve-false*: $[\![\bigwedge s.\ \neg\ P\ s\ ]\!] \Longrightarrow$ *L2-guard* P = *L2-fail*
⟨*proof*⟩

lemma *L2-spec-empty* [*L2opt*]:

L2-spec {} = *L2-fail*
$[\![\bigwedge s\ t.\ \neg\ C\ s\ t\ ]\!] \Longrightarrow$ *L2-spec* {(*s*, *t*). *C s t*} = *L2-fail*
⟨*proof*⟩

lemma *L2-unknown-bind-unbound*[*L2opt*]:
L2-seq (*L2-unknown ns*) ($\lambda x.$ *f*) = *f*
⟨*proof*⟩

lemma *L2-unknown-bind-unbound′*:
f = (λ-. *g*) $\Longrightarrow$ *L2-seq* (*L2-unknown ns*) *f* = *g*
⟨*proof*⟩

lemma *L2-seq-L2-unknown-unit*[*L2opt*]: *L2-seq* (*L2-unknown ns*) (λx:: *unit*. *f x*) = *f* ()
⟨*proof*⟩

The following rule can cause some exponential blowup, especially when rewriting is not successful. Note that *f* is duplicated in the premise. The more restricted *L2-seq* (*L2-unknown ?ns*) ($\lambda x.$ *?f*) = *?f* should also do the job, in case of properly initialised variables. Maybe we should remove it entirely from "L2opt".

lemma *L2-unknown-bind* []:
($\bigwedge a\ b.\ f\ a = f\ b$) $\Longrightarrow$ (*L2-seq* (*L2-unknown ns*) *f*) = *f undefined*
⟨*proof*⟩

lemma *L2-seq-guard-cong*:
$[\![P = P'; \bigwedge s.\ P'\ s \Longrightarrow run\ X\ s = run\ X'\ s\ ]\!] \Longrightarrow$ *L2-seq* (*L2-guard P*) (λ-. *X*) = *L2-seq* (*L2-guard P′*) (λ-. *X′*)
⟨*proof*⟩

lemma *L2-seq-guard-cong′*:

⟦ $P \equiv P'$; $\bigwedge s.\ P'\ s \Longrightarrow run\ X\ s \equiv run\ X'\ s$ ⟧ $\Longrightarrow$ *L2-seq* (*L2-guard P*) (λ-. *X*) ≡ *L2-seq* (*L2-guard P′*) (λ-. *X′*)
⟨*proof*⟩

lemma *L2-seq-guard-cong-stop*:
⟦ $P = P'$; $\bigwedge s.\ P'\ s \Longrightarrow run\ X\ s = run\ X'\ s$ ⟧ $\Longrightarrow$ *L2-seq* (*L2-guard P*) (λ-. *X*) = *STOP* (*L2-seq* (*L2-guard P′*) (λ-. *X′*))
⟨*proof*⟩

lemma *L2-seq-guard-cong-stop′*:
⟦ $P \equiv P'$; $\bigwedge s.\ P'\ s \Longrightarrow run\ X\ s \equiv run\ X'\ s$ ⟧ $\Longrightarrow$ *L2-seq* (*L2-guard P*) (λ-. *X*) ≡ *STOP* (*L2-seq* (*L2-guard P′*) (λ-. *X′*))
⟨*proof*⟩

lemma *L2-seq-guard-cong-stop″*:
⟦$\bigwedge s.\ P\ s \Longrightarrow run\ X\ s \equiv run\ X'\ s$ ⟧ $\Longrightarrow$ *L2-seq* (*L2-guard P*) (λ-. *X*) ≡ *STOP* (*L2-seq* (*L2-guard P*) (λ-. *X′*))
⟨*proof*⟩

lemma *L2-seq-guard-cong-stop‴*:
⟦$\bigwedge s.\ P\ s \Longrightarrow run\ (X\ ())\ s \equiv run\ X'\ s$ ⟧ $\Longrightarrow$ *L2-seq* (*L2-guard P*) *X* ≡ *STOP* (*L2-seq* (*L2-guard P*) (λ-. *X′*))
⟨*proof*⟩

lemma *L2-marked-seq-guard-block-cong*:
L2-seq-guard P X = *L2-seq-guard P X*
⟨*proof*⟩

lemma *L2-marked-seq-guard-cong*:
⟦$P = P'$; $\bigwedge s.\ P'\ s \Longrightarrow run\ (X\ ())\ s = run\ X'\ s$⟧ $\Longrightarrow$ *L2-seq-guard P X* = *L2-seq-guard P′* (λ-. *X′*)
⟨*proof*⟩

lemma *L2-gaurded-keep-guard-cong*:
($\bigwedge s.\ g\ s \Longrightarrow run\ c\ s = run\ c'\ s$) $\Longrightarrow$ *L2-guarded g c* = *L2-guarded g c′*
⟨*proof*⟩

lemma *gets-guard-move-before* [*L2opt*]:
L2-seq (*L2-gets f ns*) (λ*r*. *L2-seq* (*L2-guard P*) (λ-. *X r*)) =
L2-seq (*L2-guard P*) (λ-. *L2-seq* (*L2-gets f ns*) *X*)
⟨*proof*⟩

lemma *L2-seq-guard-cong″*:
⟦ $\bigwedge s.\ P\ s = P'\ s$; $\bigwedge s.\ P'\ s \Longrightarrow run\ X\ s = run\ X'\ s$ ⟧ $\Longrightarrow$ *L2-seq* (*L2-guard P*) (λ-. *X*) = *L2-seq* (*L2-guard P′*) (λ-. *X′*)
⟨*proof*⟩

lemma *guard-triv* [*L2opt*]: *L2-seq* (*L2-guard* (λ*s*. *True*)) (λ-. *X*) = *X*
⟨*proof*⟩

lemma *L2-condition-cong*:
⟦ $C = C'$; $\bigwedge s.\ C'\ s \Longrightarrow run\ A\ s = run\ A'\ s$;$\bigwedge s.\ \neg\ C'\ s \Longrightarrow run\ B\ s = run\ B'\ s$ ⟧ $\Longrightarrow$ *L2-condition* $C\ A\ B$ = *L2-condition* $C'\ A'\ B'$
⟨*proof*⟩

lemma *L2-condition-cong'*:
⟦ $\bigwedge s.\ C\ s = C'\ s$; $\bigwedge s.\ C'\ s \Longrightarrow run\ A\ s = run\ A'\ s$;$\bigwedge s.\ \neg\ C'\ s \Longrightarrow run\ B\ s = run\ B'\ s$ ⟧ $\Longrightarrow$ *L2-condition* $C\ A\ B$ = *L2-condition* $C'\ A'\ B'$
⟨*proof*⟩

lemma *L2-condition-true* [*L2opt*]: ⟦ $\bigwedge s.\ C\ s$ ⟧ $\Longrightarrow$ *L2-condition* $C\ A\ B = A$
⟨*proof*⟩

lemma *L2-condition-false* [*L2opt*]: ⟦ $\bigwedge s.\ \neg\ C\ s$ ⟧ $\Longrightarrow$ *L2-condition* $C\ A\ B = B$
⟨*proof*⟩

lemma *L2-condition-true'* [*simp*]: *L2-condition* $(\lambda s.\ True)\ A\ B = A$
⟨*proof*⟩

lemma *L2-condition-false'* [*simp*]: *L2-condition* $(\lambda s.\ False)\ A\ B = B$
⟨*proof*⟩

lemma *L2-condition-same* [*L2opt*]: *L2-condition* $C\ A\ A = A$
⟨*proof*⟩

lemma *L2-fail-seq* [*L2opt*]: *L2-seq L2-fail* X = *L2-fail*
⟨*proof*⟩

lemma *bind-fail-propagates*: ⟦ *no-throw* $(\lambda\text{-}.\ True)\ A$; *always-progress* A ⟧ $\Longrightarrow$ $A >>= (\lambda\text{-}.\ fail) = fail$
⟨*proof*⟩

lemma *state-select-select-fail*:
state-select $S \gg\!= (\lambda\text{-}.\ select\ UNIV) \gg\!= (\lambda\text{-}.\ fail) = fail$
⟨*proof*⟩

lemma *L2-fail-propagates* [*L2opt*]:
L2-seq $(L2\text{-}gets\ V\ n)\ (\lambda\text{-}.\ L2\text{-}fail) = L2\text{-}fail$
L2-seq $(L2\text{-}modify\ M)\ (\lambda\text{-}.\ L2\text{-}fail) = L2\text{-}fail$
L2-seq $(L2\text{-}spec\ S)\ (\lambda\text{-}.\ L2\text{-}fail) = L2\text{-}fail$
L2-seq $(L2\text{-}guard\ G)\ (\lambda\text{-}.\ L2\text{-}fail) = L2\text{-}fail$
L2-seq $(L2\text{-}unknown\ ns)\ (\lambda\text{-}.\ L2\text{-}fail) = L2\text{-}fail$
L2-seq L2-fail $(\lambda\text{-}.\ L2\text{-}fail) = L2\text{-}fail$
⟨*proof*⟩

lemma *L2-condition-distrib*:
L2-seq (*L2-condition C L R*) *X* = *L2-condition C* (*L2-seq L X*) (*L2-seq R X*)
⟨*proof*⟩

lemmas *L2-fail-propagate-condition* [*L2opt*] = *L2-condition-distrib* [**where** *X*=λ-. *L2-fail*]

lemma *L2-seq-condition-skip-throw* [*L2opt*]: *L2-seq*
(*L2-condition c* (*L2-gets* (λ-. ()) *ns*)
(*L2-throw x ms*))
(λ*r*. *y*) =
(*L2-condition c y*
(*L2-throw x ms*))
⟨*proof*⟩

lemma *L2-fail-propagate-catch* [*L2opt*]:
(*L2-seq* (*L2-catch L R*) (λ-. *L2-fail*)) = (*L2-catch* (*L2-seq L* (λ-. *L2-fail*)) (λ*e*. *L2-seq* (*R e*) (λ-. *L2-fail*)))
⟨*proof*⟩

lemma *L2-condition-fail-lhs* [*L2opt*]:
L2-condition C L2-fail A = *L2-seq* (*L2-guard* (λ*s*. ¬ *C s*)) (λ-. *A*)
⟨*proof*⟩

lemma *L2-condition-fail-rhs* [*L2opt*]:
L2-condition C A L2-fail = *L2-seq* (*L2-guard* (λ*s*. *C s*)) (λ-. *A*)
⟨*proof*⟩

lemma *L2-catch-fail* [*L2opt*]: *L2-catch L2-fail A* = *L2-fail*
⟨*proof*⟩

lemma *L2-try-fail* [*L2opt*]: *L2-try L2-fail* = *L2-fail*
⟨*proof*⟩

lemma *L2-while-fail* [*L2opt*]: *L2-while C* (λ-. *L2-fail*) *i n* = (*L2-seq* (*L2-guard* (λ*s*. ¬ *C i s*)) (λ-. *L2-gets* (λ*s*. *i*) *n*))
⟨*proof*⟩

lemma *unit-bind*: (λ*x*. *f* (*x*::*unit*)) = (λ-. *f* ())
⟨*proof*⟩

lemma *unit-bind'*: *f* ≡ (λ-. *f* ())
⟨*proof*⟩

lemma *L2-while-cong*:
assumes *c-eq*: ⋀*r s*. *c r s* = *c'* *r s*
assumes *bdy-eq*: ⋀*r s*. *c'* *r s* ⟹ *run* (*A r*) *s* = *run* (*A'* *r*) *s*
shows *L2-while c A* = *L2-while c' A'*

⟨*proof*⟩

lemma *L2-while-simp-cong*:
assumes *c-eq*: $\bigwedge r\ s.\ c\ r\ s = c'\ r\ s$
assumes *bdy-eq*[*simplified simp-implies-def*]: $\bigwedge r\ s.\ c'\ r\ s$ =*simp*=> *run* $(A\ r)\ s$ = *run* $(A'\ r)\ s$
shows *L2-while* $c\ A$ = *L2-while* $c'\ A'$
⟨*proof*⟩

lemma *L2-while-cong'*:
assumes *c-eq*: $c = c'$
assumes *bdy-eq*: $\bigwedge r\ s.\ c'\ r\ s \Longrightarrow$ *run* $(A\ r)\ s$ = *run* $(A'\ r)\ s$
shows *L2-while* $c\ A$ = *L2-while* $c'\ A'$
⟨*proof*⟩

lemma *L2-while-simp-cong'*:
assumes *c-eq*: $c = c'$
assumes *bdy-eq*[*simplified simp-implies-def*]: $\bigwedge r\ s.\ c'\ r\ s$ =*simp*=> *run* $(A\ r)\ s$ = *run* $(A'\ r)\ s$
shows *L2-while* $c\ A$ = *L2-while* $c'\ A'$
⟨*proof*⟩

lemma *L2-while-cong-split*:
assumes *c-eq*: $c = c'$
assumes *bdy-eq*: *PROP SPLIT* $(\bigwedge r\ s.\ c'\ r\ s \Longrightarrow$ *run* $(A\ r)\ s$ = *run* $(A'\ r)\ s)$
shows *L2-while* $c\ A$ = *L2-while* $c'\ A'$
⟨*proof*⟩

lemma *L2-while-cong-simp-split*:
assumes *c-eq*: $c = c'$
assumes *bdy-eq*: *PROP SPLIT* $(\bigwedge r\ s.\ c'\ r\ s$ =*simp*=> *run* $(A\ r)\ s$ = *run* $(A'\ r)\ s)$
shows *L2-while* $c\ A$ = *L2-while* c' (*ETA-TUPLED* A')
⟨*proof*⟩

lemma *L2-while-cong-split-stop*:
assumes *c-eq*: $c = c'$
assumes *bdy-eq*: *PROP SPLIT* $(\bigwedge r\ s.\ c'\ r\ s \Longrightarrow$ *run* $(A\ r)\ s$ = *run* $(A'\ r)\ s)$
shows *L2-while* $c\ A$ = *STOP* (*L2-while* $c'\ A'$)
⟨*proof*⟩

lemma *L2-while-cong-simp-split-stop*:
assumes *c-eq*: $c = c'$
assumes *bdy-eq*: *PROP SPLIT* $(\bigwedge r\ s.\ c'\ r\ s$ =*simp*=> *run* $(A\ r)\ s$ = *run* $(A'\ r)\ s)$
shows *L2-while* $c\ A$ = *STOP* (*L2-while* $c'\ A'$)

⟨*proof*⟩

lemma *L2-while-cong''*:
assumes *c-eq*: $c = c'$
assumes *bdy-eq*: $\bigwedge r\ s.\ c'\ r\ s \Longrightarrow run\ (A\ r)\ s = run\ (A'\ r)\ s$
assumes *i-eq*: $i = i'$
shows *L2-while c A i ns* = *L2-while c' A' i' ns*
⟨*proof*⟩

lemma *L2-while-cong-block*: *L2-while c b* = *L2-while c b*
⟨*proof*⟩
lemma *L2-while-unbind-STOP*: $c \equiv c' \Longrightarrow b \equiv b' \Longrightarrow$ *L2-while c b i ns* $\equiv$ *STOP-UNBIND* (*L2-while c' b' i ns*)
⟨*proof*⟩

lemma *L2-returncall-trivial* [*L2opt*]:
$[\![\bigwedge s\ v.\ f\ v\ s = v\ ]\!] \Longrightarrow$ *L2-returncall x f* = *L2-call x*
⟨*proof*⟩

lemma *L2-gets-unused*:
$[\![\bigwedge x\ y\ s.\ run\ (B\ x)\ s = run\ (B\ y)\ s\ ]\!] \Longrightarrow$ *L2-seq* (*L2-gets A n*) *B* = (*B undefined*)
⟨*proof*⟩

lemma *L2-gets-unbound*[*L2opt*]:
L2-seq (*L2-gets A n*) ($\lambda x.\ f$) = *f*
⟨*proof*⟩

lemma *L2-gets-bind*:
L2-seq (*L2-gets* (λ-. *x* :: *'var-type*) *n*) *f* = *f x*
⟨*proof*⟩

lemma *L2-gets-bind-stop-cong*:
L2-seq (*L2-gets* (λ-. *x*) *n*) *f* = *L2-seq* (*L2-gets* (λ-. *x*) *n*) *f*
⟨*proof*⟩

lemma *L2-seq-stop-cong*:
L2-seq x y = *L2-seq x y*
⟨*proof*⟩

lemma *L2-marked-seq-gets-apply*:
L2-seq-gets c n A $\equiv$ *A c*
⟨*proof*⟩

lemma *split-seq-assoc* [*L2opt*]:

$(\lambda x.\ L2\text{-}seq\ (case\ x\ of\ (a,\ b) \Rightarrow B\ a\ b)\ (G\ x)) = (\lambda x.\ case\ x\ of\ (a,\ b) \Rightarrow (L2\text{-}seq\ (B\ a\ b)\ (G\ x)))$
⟨*proof*⟩

lemma *whileLoop-succeeds-terminates-infinite'*:
assumes *run* (*whileLoop* (λ-. *C*) (λ*x*. *gets* (λ*s*. *B s x*)) *i*) *s* ≠ ⊤
shows *C s* ⟹ *False*
⟨*proof*⟩

lemma *run-whileLoop-infinite'*: *run* (*whileLoop* (λ*i*. *C*)
(λ*x*. *gets* (λ*s*. *B s x*))
i)
s =
run (*guard* (λ*s*. ¬ *C s*) ≫= (λ-. *gets* (λ-. *i*))) *s*
⟨*proof*⟩

lemma *whileLoop-infinite'*:
whileLoop (λ*i*. *C*)
(λ*x*. *gets* (λ*s*. *B s x*))
i
=
guard (λ*s*. ¬ *C s*) ≫= (λ-. *gets* (λ-. *i*))
⟨*proof*⟩

lemma *L2-while-infinite* [*L2opt*]:
L2-while (λ*i s*. *C s*) (λ*x*. *L2-gets* (λ*s*. *B s x*) *n'*) *i n*
= (*L2-seq* (*L2-guard* (λ*s*. ¬ *C s*)) (λ-. *L2-gets* (λ-. *i*) *n*))
⟨*proof*⟩

lemmas *L2-gets-bind-c-exntype* [*L2opt*] = *L2-gets-bind* [**where** *'var-type='gx c-exntype*]

lemmas *L2-gets-bind-unit* [*L2opt*] = *L2-gets-bind* [**where** *'var-type=unit*]

declare *L2-voidcall-def* [*L2opt*]
declare *L2-modifycall-def* [*L2opt*]
declare *ucast-id* [*L2opt*]
declare *scast-id* [*L2opt*]

declare *singleton-iff* [*L2opt*]

lemma *L2-gets-L2-seq-if-P-1-0* [*L2opt*]:
L2-seq (*L2-gets* (λ*s*. *if P s then 1 else 0*) *n*) (λ*x*. *Q x*)
= (*L2-seq* (*L2-gets P n*) (λ*x*. *Q* (*if x then 1 else 0*)))
⟨*proof*⟩

lemma *L2-guard-join-simple* [*L2opt*]:
L2-seq (*L2-guard A*) (λ-. *L2-guard B*) = *L2-guard* (λ*s*. *A s* ∧ *B s*)
⟨*proof*⟩

lemma *L2-guard-join-nested* [*L2opt*]:
L2-seq (*L2-guard A*) (λ-. *L2-seq* (*L2-guard B*) (λ-. *C*))
= *L2-seq* (*L2-guard* (λ*s*. *A s* ∧ *B s*)) (λ-. *C*)
⟨*proof*⟩

lemma *L2-guarded-L2-gets*[*L2opt*]:
L2-guarded (λ-. *P*) (*L2-seq* (*L2-gets f ns*) *g*) = *L2-seq* (*L2-gets f ns*) (λ*x*. *L2-guarded* (λ-. *P*) (*g x*))
⟨*proof*⟩
lemma *L2-guarded-L2-guard*[*L2opt*]:
L2-guarded (λ-. *P*) (*L2-seq* (*L2-guard Q*) *g*) = *L2-seq* (*L2-guard Q*) (λ*x*. *L2-guarded* (λ-. *P*) (*g x*))
⟨*proof*⟩

lemma *unbind*:
assumes *eq*: ⋀*x*. *f x* = *g*
shows *f* = (λ-. *f ZERO*('*a*::*c-type*))
⟨*proof*⟩

lemma *L2-seq-unknown-block-cong*: *L2-seq-unknown ns X* = *L2-seq-unknown ns X*
⟨*proof*⟩

lemma *L2-seq-unknown-STOP*: *f* ≡ *f'* ⟹ *L2-seq-unknown ns f* ≡ *STOP* (*L2-seq* (*L2-unknown ns*) *f'*)
⟨*proof*⟩

lemma *L2-seq-unknown-unfold-STOP*: *f* ≡ (λ-. *g*) ⟹ *L2-seq-unknown ns f* ≡ *STOP g*
⟨*proof*⟩

lemma *L2-seq-gets-unfold*: *L2-seq-gets c ns f* = *f c*
⟨*proof*⟩

lemma *L2-condition-L2-seq-distrib*:
L2-seq (*L2-condition C L R*) (λ*x*. *L2-seq* (*X x*) *Y*) ≡
L2-seq
(*L2-condition C*
(*L2-seq L X*)
(*L2-seq R X*)) *Y*

⟨*proof*⟩

lemma *L2-condition-L2-seq-gets-distrib*:
L2-seq (*L2-condition C L R*) (λ*x*. *STOP* (*L2-seq-gets* (*X x*) *n Y*)) ≡
L2-seq
(*L2-condition C*
(*L2-seq L* (λ*x*. *L2-gets* (λ-. *X x*) *n*))
(*L2-seq R* (λ*x*. *L2-gets* (λ-. *X x*) *n*))) *Y*
⟨*proof*⟩

lemma *L2-condition-L2-seq-gets-distrib′*:
assumes *asm*: ⋀*x*. *STOP* (*L2-seq-gets* (*X x*) *n* (*Y x*)) ≡ *L2-seq* (*A x*) *B*
shows *L2-seq* (*L2-condition C L R*) (λ*x*. *STOP* (*L2-seq-gets* (*X x*) *n* (*Y x*))) ≡

L2-seq
(*L2-condition C*
(*FUSE* (*L2-seq L A*))
(*FUSE* (*L2-seq R A*))) *B*
⟨*proof*⟩

lemma *L2-seq-condition-block-cong*: *L2-seq-condition c L R X* = *L2-seq-condition c L R X*
⟨*proof*⟩

lemma *L2-seq-condition-unfold-STOP*:
L2-seq L X ≡ *L′* ⟹ *L2-seq R X* ≡ *R′* ⟹
L2-seq-condition c L R X ≡ *STOP* (*L2-condition c L′ R′*)
⟨*proof*⟩

thm *L2-split-fixups*

thm *L2-condition-L2-seq-distrib* [*split-tuple X arity* : *3*, *simplified L2-split-fixups*]
thm *L2-condition-L2-seq-gets-distrib* [*split-tuple X arity* : *3*, *simplified L2-split-fixups*]
thm *L2-condition-L2-seq-gets-distrib′* [*split-tuple X* **and** *A arity* : *2*, *simplified L2-split-fixups*]
thm *L2-seq-assoc*
end

Chapter 20

IO phase: In/Out Parameters

20.1 Heap Typing for Split Heap

theory *TypHeapSimple*
imports
AutoCorres-Base
begin

20.2 Valid root footprint

lemma *c-null-guard-size-of-limit*:
c-null-guard (*p*::'*a*::*c-type ptr*) ⟹ *size-td* (*typ-uinfo-t* (*TYPE*('*a*))) < 2 ^ *len-of* *TYPE*(*addr-bitsize*)
⟨*proof*⟩

lemma *root-ptr-valid-legacy-def*: *root-ptr-valid d p* ⟷
valid-root-footprint d (*ptr-val* (*p*::'*a ptr*)) (*typ-uinfo-t TYPE*('*a*)) ∧
valid-simple-footprint d (*ptr-val* (*p*::'*a*::*c-type ptr*)) (*typ-uinfo-t TYPE*('*a*)) ∧
d, *c-guard* ⊨$_t$ *p*
⟨*proof*⟩

definition
simple-lift :: *heap-raw-state* ⇒ ('*a*::*c-type*) *ptr* ⇒ '*a option*
where
simple-lift s p = (
if (*root-ptr-valid* (*hrs-htd s*) *p*) *then*
(*Some* (*h-val* (*hrs-mem s*) *p*))
else
None)

lemma *simple-lift-root-ptr-valid*:
simple-lift s p = *Some x* ⟹ *root-ptr-valid* (*hrs-htd s*) *p*

⟨*proof*⟩

lemma *simple-lift-h-t-valid*:
simple-lift s p = *Some x* ⟹ (*hrs-htd s*), *c-guard* ⊨$_t$ *p*
⟨*proof*⟩

lemma *simple-lift-valid-root-footprint*:
simple-lift s (*p*::*'a*::*c-type ptr*) = *Some x* ⟹ *valid-root-footprint* (*hrs-htd s*) (*ptr-val p*) (*typ-uinfo-t* (*TYPE*(*'a*)))
⟨*proof*⟩

lemma *simple-lift-c-guard*:
assumes *lift*: *simple-lift s p* = *Some x*
shows *c-guard p*
⟨*proof*⟩

lemma *root-ptr-valid-heap-update-other*:
assumes *val-p*: *root-ptr-valid d* (*p*::*'a*::*mem-type ptr*)
and *val-q*: *root-ptr-valid d* (*q*::*'b*::*c-type ptr*)
and *neq*: *ptr-val p* ≠ *ptr-val q*
shows *h-val* (*heap-update p v h*) *q* = *h-val h q*
⟨*proof*⟩

lemma *root-ptr-valid-heap-update-other-typ*:
assumes *val-p*: *root-ptr-valid d* (*p*::*'a*::*mem-type ptr*)
and *val-q*: *root-ptr-valid d* (*q*::*'b*::*c-type ptr*)
and *neq*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*'b*)
shows *h-val* (*heap-update p v h*) *q* = *h-val h q*
⟨*proof*⟩

lemma *simple-lift-heap-update*:
⟦ *root-ptr-valid* (*hrs-htd h*) *p* ⟧ ⟹
simple-lift (*hrs-mem-update* (*heap-update p v*) *h*)
= (*simple-lift h*)(*p* := *Some* (*v*::*'a*::*mem-type*))
⟨*proof*⟩

lemma *simple-lift-heap-update-other*:
⟦ *root-ptr-valid* (*hrs-htd d*) (*p*::*'b*::*mem-type ptr*);
typ-uinfo-t TYPE(*'a*) ≠ *typ-uinfo-t TYPE*(*'b*) ⟧ ⟹
simple-lift (*hrs-mem-update* (*heap-update p v*) *d*) = ((*simple-lift d*)::*'a*::*c-type typ-heap*)
⟨*proof*⟩

lemma *h-val-simple-lift*:
simple-lift h p = Some v ⟹ h-val (hrs-mem h) p = v
⟨*proof*⟩

lemma *the-simple-lift-h-val-conv*:
root-ptr-valid (hrs-htd h) p ⟹ the (simple-lift h p) = h-val (hrs-mem h) p
⟨*proof*⟩

lemma *slift-clift-Some-same*:
assumes *slift*: *simple-lift s p = Some x*
assumes *clift*: *clift s p = Some y*
shows *x = y*
⟨*proof*⟩

lemma *simple-lift-Some-clift-Some*:
assumes *slift*: *simple-lift s p = Some x*
shows *clift s p = Some x*
⟨*proof*⟩

lemma *h-val-field-simple-lift*:
⟦ *simple-lift h (pa :: 'a ptr) = Some (v::'a::mem-type)*;
field-ti TYPE('a) f = Some t;
export-uinfo (the (field-ti TYPE('a) f)) = export-uinfo (typ-info-t TYPE('b :: mem-type)) ⟧ ⟹
h-val (hrs-mem h) (Ptr &(pa→f) :: 'b :: mem-type ptr) = from-bytes (access-ti$_0$ (the (field-ti TYPE('a) f)) v)
⟨*proof*⟩

lemma *simple-lift-heap-update'*:
simple-lift h p = Some v' ⟹
simple-lift (hrs-mem-update (heap-update (p::('a::{mem-type}) ptr) v) h)
= (simple-lift h)(p := Some v)
⟨*proof*⟩

lemma *simple-lift-hrs-mem-update-None* [*simp*]:
(simple-lift (hrs-mem-update a hp) x = None) = (simple-lift hp x = None)
⟨*proof*⟩

lemma *simple-lift-hrs-mem-update-Some*: *(∃ z. simple-lift (hrs-mem-update upd h) x = Some z)*
⟷ (∃ z. simple-lift h x = Some z)

⟨*proof*⟩

lemma *clift-hrs-mem-update-None* [*simp*]:

(*clift* (*hrs-mem-update a hp*) *x* = *None*) = (*clift hp x* = *None*)
⟨*proof*⟩

lemma *clift-hrs-mem-update-Some*:
(∃ *z*. *clift* (*hrs-mem-update a hp*) *x* = *Some z*) = (∃ *z*. *clift hp x* = *Some z*)
⟨*proof*⟩

lemma *simple-lift-data-eq*:
⟦ *h-val* (*hrs-mem h*) *p* = *h-val* (*hrs-mem h'*) *p'*;
root-ptr-valid (*hrs-htd h*) *p* = *root-ptr-valid* (*hrs-htd h'*) *p'* ⟧ ⟹
simple-lift h p = *simple-lift h' p'*
⟨*proof*⟩

lemma *h-val-heap-update-disjoint*:
⟦ {*ptr-val p* ..+ *size-of TYPE*('*a*::*c-type*)}
∩ {*ptr-val q* ..+ *size-of TYPE*('*b*::*mem-type*)} = {} ⟧ ⟹
h-val (*heap-update* (*q* :: '*b ptr*) *r h*) (*p* :: '*a ptr*) = *h-val h p*
⟨*proof*⟩

lemma *update-ti-t-valid-size*:
size-of TYPE('*b*) = *size-td t* ⟹
update-ti-t t (*to-bytes-p* (*val*::'*b*::*mem-type*)) *obj* = *update-ti t* (*to-bytes-p val*) *obj*
⟨*proof*⟩

lemma *h-val-field-from-bytes'*:
⟦ *field-ti TYPE*('*a*::{*mem-type*}) *f* = *Some t*;
export-uinfo t = *export-uinfo* (*typ-info-t TYPE*('*b*::{*mem-type*})) ⟧ ⟹
h-val h (*Ptr* &(*pa*→*f*) :: '*b ptr*) = *from-bytes* (*access-ti*$_0$ *t* (*h-val h pa*))
⟨*proof*⟩

lemma *h-val-field-from-root*:
fixes *p*::'*a*:: *mem-type ptr*
assumes *fl*:
field-lookup (*typ-info-t TYPE*('*a*::{*mem-type*})) *f 0* =
Some (*adjust-ti* (*typ-info-t TYPE*('*b*::*mem-type*)) *fld fld-update*, *n*)
assumes *fg-cons*: *fg-cons fld* (*fld-update*)
shows *h-val h* (*PTR*('*b*) &(*p*→*f*)) = *fld* (*h-val h p*)
⟨*proof*⟩

lemma *simple-lift-super-field-update-lookup*:
fixes *dummy* :: '*b* :: *mem-type*
assumes *field-lookup* (*typ-info-t TYPE*('*b*::*mem-type*)) *f 0* = *Some* (*s*,*n*)
and *typ-uinfo-t TYPE*('*a*) = *export-uinfo s*
and *simple-lift h p* = *Some v'*
shows (*super-field-update-t* (*Ptr* (&(*p*→*f*))) (*v*::'*a*::*mem-type*) ((*simple-lift h*)::'*b ptr* ⇒ '*b option*)) =
((*simple-lift h*)(*p* ↦ *field-update* (*field-desc s*) (*to-bytes-p v*) *v'*))

⟨*proof*⟩

lemma *field-offset-addr-card*:
∃ *x*. *field-lookup* (*typ-info-t TYPE*('*a*::*mem-type*)) *f* 0 = *Some x*
⟹ *field-offset TYPE*('*a*) *f* < *addr-card*
⟨*proof*⟩

lemma *unat-of-nat-field-offset*:
∃ *x*. *field-lookup* (*typ-info-t TYPE*('*a*::*mem-type*)) *f* 0 = *Some x* ⟹
unat (*of-nat* (*field-offset TYPE*('*a*) *f*) :: *addr*) = *field-offset TYPE*('*a*) *f*
⟨*proof*⟩

lemma *field-of-t-field-lookup*:
assumes *a*: *field-lookup* (*typ-info-t TYPE*('*a*::*mem-type*)) *f* 0 = *Some* (*s*, *n*)
assumes *b*: *export-uinfo s* = *typ-uinfo-t TYPE*('*b*::*mem-type*)
assumes *n*: *n* = *field-offset TYPE*('*a*) *f*
shows *field-of-t* (*Ptr* &(*ptr*→*f*) :: ('*b ptr*)) (*ptr* :: '*a ptr*)
⟨*proof*⟩

lemma *simple-lift-field-update'*:
fixes *val* :: '*b* :: *mem-type* **and** *ptr* :: '*a* :: *mem-type ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f* 0 =
Some (*adjust-ti* (*typ-info-t TYPE*('*b*)) *xf xfu*, *n*)
and *xf-xfu*: *fg-cons xf xfu*
and *cl*: *simple-lift hp ptr* = *Some z*
shows (*simple-lift* (*hrs-mem-update* (*heap-update* (*Ptr* &(*ptr*→*f*)) *val*) *hp*)) =
(*simple-lift hp*)(*ptr* ↦ *xfu val z*)
(**is** *?LHS* = *?RHS*)
⟨*proof*⟩

lemma *simple-lift-field-update*:
fixes *val* :: '*b* :: *mem-type* **and** *ptr* :: '*a* :: *mem-type ptr*
assumes *fl*: *field-ti TYPE*('*a*) *f* =
Some (*adjust-ti* (*typ-info-t TYPE*('*b*)) *xf* (*xfu o* (λ*x* -. *x*)))
and *xf-xfu*: *fg-cons xf* (*xfu o* (λ*x* -. *x*))
and *cl*: *simple-lift hp ptr* = *Some z*
shows (*simple-lift* (*hrs-mem-update* (*heap-update* (*Ptr* &(*ptr*→*f*)) *val*) *hp*)) =
(*simple-lift hp*)(*ptr* ↦ *xfu* (λ-. *val*) *z*)
(**is** *?LHS* = *?RHS*)
⟨*proof*⟩

lemma *simple-heap-diff-types-impl-diff-ptrs*:
⟦ *root-ptr-valid h* (*p*::('*a*::*c-type*) *ptr*);
root-ptr-valid h (*q*::('*b*::*c-type*) *ptr*);
typ-uinfo-t TYPE('*a*) ≠ *typ-uinfo-t TYPE*('*b*) ⟧ ⟹
ptr-val p ≠ *ptr-val q*
⟨*proof*⟩

lemma *h-val-update-regions-disjoint*:

⟦ { ptr-val p ..+ size-of TYPE('a) } ∩ { ptr-val x ..+ size-of TYPE('b)} = {} ⟧
⟹
h-val (heap-update p (v::('a::mem-type)) h) x = h-val h (x::('b::c-type) ptr)
⟨proof⟩

lemma *h-val-heap-update-padding-disjoint*:
fixes p::'a::mem-type ptr
fixes q::'b::c-type ptr
shows ⟦ ptr-span p ∩ ptr-span q = {}; length bs = size-of TYPE('a) ⟧ ⟹
h-val (heap-update-padding p v bs h) q = h-val h q
⟨proof⟩

lemma *simple-lift-field-update-t*:
fixes val :: 'b :: mem-type **and** ptr :: 'a :: mem-type ptr
assumes fl: field-ti TYPE('a) f = Some t
and diff: typ-uinfo-t TYPE('a) ≠ typ-uinfo-t TYPE('c :: c-type)
and eu: export-uinfo t = export-uinfo (typ-info-t TYPE('b))
and cl: simple-lift hp ptr = Some z
shows ((simple-lift (hrs-mem-update (heap-update (Ptr &(ptr→f)) val) hp)) :: 'c ptr ⇒ 'c option) =
simple-lift hp
⟨proof⟩

lemma *simple-lift-heap-update-other'*:
⟦ simple-lift h (p::'b::mem-type ptr) = Some v';
typ-uinfo-t TYPE('a) ≠ typ-uinfo-t TYPE('b) ⟧ ⟹
simple-lift (hrs-mem-update (heap-update p v) h) = ((simple-lift h)::'a::c-type typ-heap)
⟨proof⟩

lemma *simple-lift-heap-update-bytes-in-other*:
⟦ simple-lift h (p::'b::mem-type ptr) = Some v';
typ-uinfo-t TYPE('b) ≠ typ-uinfo-t TYPE('c);
{ ptr-val q ..+ size-of TYPE('a)} ⊆ {ptr-val p ..+ size-of TYPE('b) } ⟧ ⟹
simple-lift (hrs-mem-update (heap-update (q::'a::mem-type ptr) v) h) = ((simple-lift h)::'c::mem-type typ-heap)
⟨proof⟩

lemma *typ-name-neq*:
typ-name (export-uinfo (typ-info-t TYPE('a::c-type)))
≠ typ-name (export-uinfo (typ-info-t TYPE('b::c-type)))
⟹ typ-uinfo-t TYPE('a) ≠ typ-uinfo-t TYPE('b)

⟨*proof*⟩

lemma *of-nat-mod-div-decomp*:
of-nat k
= *of-nat* (*k div size-of TYPE*('*b*)) ∗ *of-nat* (*size-of TYPE*('*b*::*mem-type*)) +
of-nat (*k mod size-of TYPE*('*b*))
⟨*proof*⟩

lemma *c-guard-array-c-guard*:
⟦ ⋀*x*. *x* < *CARD*('*a*) ⟹ *c-guard* (*ptr-coerce p* +$_p$ *int x* :: '*b ptr*) ⟧ ⟹ *c-guard*
(*p* :: ('*b* :: *mem-type*, '*a* :: *finite*) *array ptr*)
⟨*proof*⟩

lemma *heap-list-update-list'*:
⟦ *n* + *x* ≤ *length v*; *length v* < *addr-card*; *q* = (*p* + *of-nat x*) ⟧ ⟹
heap-list (*heap-update-list p v h*) *n q* = *take n* (*drop x v*)
⟨*proof*⟩

lemma *outside-intvl-range*:
p ∉ {*a* ..+ *b*} ⟹ *p* < *a* ∨ *p* ≥ *a* + *of-nat b*
⟨*proof*⟩

lemma *root-ptr-valid-intersect-array*:
⟦ ∀ *j* < *n*. *root-ptr-valid htd* (*p* +$_p$ *int j*);
root-ptr-valid htd (*q* :: ('*a* :: *c-type*) *ptr*) ⟧
⟹ (∃ *m* < *n*. *q* = (*p* +$_p$ *int m*))
∨ ({*ptr-val p* ..+ *size-of TYPE* ('*a*) ∗ *n*} ∩ {*ptr-val q* ..+ *size-of TYPE* ('*a* ::
c-type)} = {})
⟨*proof*⟩

lemmas *simple-lift-simps* =
typ-name-neq
simple-lift-c-guard
h-val-simple-lift
simple-lift-heap-update
simple-lift-heap-update-other
c-guard-field
h-val-field-simple-lift
simple-lift-field-update
simple-lift-field-update-t
c-guard-array-field

lemmas *typ-simple-heap-simps* = *simple-lift-simps*

lemma *valid-footprint-overlap-cases*:
assumes *valid-p*: *valid-footprint d* (*ptr-val* (*p*::'*a*::*mem-type ptr*)) (*typ-uinfo-t TYPE*('*a*))
assumes *valid-q*: *valid-footprint d* (*ptr-val* (*q*::'*b*::*mem-type ptr*)) (*typ-uinfo-t TYPE*('*b*))
assumes *overlap*: *ptr-span p* ∩ *ptr-span q* ≠ {}
shows *TYPE*('*a*) ≤$_\tau$ *TYPE*('*b*) ∨ *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)
⟨*proof*⟩

lemma *valid-root-footprint-valid-footprint-overlap-case*:
assumes *valid-p*: *valid-root-footprint d* (*ptr-val* (*p*::'*a*:: *mem-type ptr*)) (*typ-uinfo-t* (*TYPE*('*a*)))
assumes *valid-q*: *valid-footprint d* (*ptr-val* (*q*::'*b*::*mem-type ptr*)) (*typ-uinfo-t* (*TYPE*('*b*)))
assumes *overlap*: *ptr-span p* ∩ *ptr-span q* ≠ {}
shows *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)
⟨*proof*⟩

lemma *valid-root-footprint-overlap-contained*:
assumes *valid-root-x*: *valid-root-footprint d x t*
assumes *valid-y*: *valid-footprint d y s*
assumes *overlap*: {*x* ..+ *size-td t*} ∩ {*y* ..+ *size-td s*} ≠ {}
shows *y* ∈ {*x* ..+ *size-td t*}
⟨*proof*⟩

lemma *valid-footprint-field-of-contained*:
assumes *valid-x*: *valid-footprint d x t*
assumes *field*: *field-of off s t*
shows {*x* + *off* ..+ *size-td s*} ⊆ {*x* ..+ *size-td t*}
⟨*proof*⟩

lemma *valid-root-footprint-overlap-field-of*:
assumes *valid-root-x*: *valid-root-footprint d x t*
assumes *valid-y*: *valid-footprint d y s*
assumes *y*: *y* ∈ {*x* ..+ *size-td t*}
shows *field-of* (*y* − *x*) *s t*
⟨*proof*⟩

lemma *valid-root-footprint-overlap-contained'*:
assumes *valid-root-x*: *valid-root-footprint d x t*
assumes *valid-y*: *valid-footprint d y s*
assumes *overlap*: {*x* ..+ *size-td t*} ∩ {*y* ..+ *size-td s*} ≠ {}
shows {*y* ..+ *size-td s*} ⊆ {*x* ..+ *size-td t*}
⟨*proof*⟩

lemma *valid-footprint-sub-cases*:

assumes *valid-p*: *valid-footprint d p s*
assumes *valid-q*: *valid-footprint d q t*
assumes *sub*: $\neg\ t < s$
shows $\{p..+size\text{-}td\ s\} \cap \{q..+size\text{-}td\ t\} = \{\} \vee field\text{-}of\ (p - q)\ (s)\ (t)$
⟨*proof*⟩

lemma *valid-root-footprint-dist-type-cases*:
assumes *valid-p*: *valid-root-footprint d* (*ptr-val* (*p*::'*a*:: *mem-type ptr*)) (*typ-uinfo-t* (*TYPE*('*a*)))
assumes *valid-q*: *valid-footprint d* (*ptr-val* (*q*::'*b*::*mem-type ptr*)) (*typ-uinfo-t* (*TYPE*('*b*)))
assumes *dist-type*: *typ-uinfo-t* (*TYPE*('*a*)) ≠ *typ-uinfo-t* (*TYPE*('*b*))
assumes *nested-case*: $\bigwedge f.\ f \in set\ (field\text{-}names\text{-}u\ (typ\text{-}uinfo\text{-}t\ (TYPE('a)))\ (typ\text{-}uinfo\text{-}t\ (TYPE('b)))) \Longrightarrow$
field-lookup (*typ-uinfo-t TYPE*('*a*)) *f 0* = *Some* (*typ-uinfo-t TYPE*('*b*), *unat* (*ptr-val q* − *ptr-val p*)) ⟹
field-of-t (*PTR*('*b*) &(*p*→*f*)) *p* ⟹
ptr-val q = &(*p*→*f*) ⟹
ptr-span q ⊆ *ptr-span p* ⟹ *P*
assumes *disj-case*: *ptr-span p* ∩ *ptr-span q* = {} ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *cvalid-field-pres*:
assumes *lookup*: *field-lookup* (*typ-uinfo-t TYPE*('*a*::*mem-type*)) *f 0* = *Some* (*typ-uinfo-t TYPE*('*b*::*mem-type*), *n*)
assumes *valid*: *d*,*c-guard* ⊨$_t$ (*p*::'*a*:: *mem-type ptr*)
shows *d*, *c-guard* ⊨$_t$ *PTR*('*b*) &(*p*→*f*)
⟨*proof*⟩

lemma *cvalid-field-pres′*:
assumes *ptr-val-eq*: *ptr-val q* = &(*p*→*f*)
assumes *lookup*: *field-lookup* (*typ-uinfo-t TYPE*('*a*::*mem-type*)) *f 0* = *Some* (*typ-uinfo-t TYPE*('*b*::*mem-type*), *n*)
assumes *valid*: *d*,*c-guard* ⊨$_t$ (*p*::'*a*:: *mem-type ptr*)
shows *d*, *c-guard* ⊨$_t$ (*q*::'*b ptr*)
⟨*proof*⟩

lemma *cvalid-field-pres″*:
assumes *ptr-val-eq*: *ptr-val q* = &(*p*→*f*)
assumes *lookup*: *field-lookup* (*typ-info-t TYPE*('*a*::*mem-type*)) *f 0* = *Some* (*s*, *n*)
assumes *match*: *export-uinfo s* = *typ-uinfo-t TYPE*('*b*::*mem-type*)
assumes *valid*: *d*,*c-guard* ⊨$_t$ (*p*::'*a*:: *mem-type ptr*)
shows *d*, *c-guard* ⊨$_t$ (*q*::'*b ptr*)
⟨*proof*⟩

lemma *cvalid-field-pres'''*:
assumes *lookup*: *field-lookup* (*typ-info-t TYPE*('*a*::*mem-type*)) *f 0* = *Some* (*t*, *n*)
assumes *match*: *export-uinfo t* = *typ-uinfo-t TYPE*('*b*::*mem-type*)
assumes *valid*: *d*,*c-guard* $\models_t$ (*p*::'*a*:: *mem-type ptr*)
shows *d*, *c-guard* $\models_t$ *PTR*('*b*) &(*p*→*f*)
⟨*proof*⟩

lemma *the-clift-eq-h-val-eq*:
assumes *h-val-eq*: *hrs-htd hp* $\models_t$ *p* ⟹ *h-val* (*hrs-mem* (*hrs-mem-update a hp*)) *p* = *h-val* (*hrs-mem hp*) *p*
shows *the* (*clift* (*hrs-mem-update a hp*) *p*) = *the* (*clift hp p*)
⟨*proof*⟩

lemma *field-lvalue-same-conv*: &(*p*::'*a*:: *c-type ptr*→*f*) = &(*q*::'*a*:: *c-type ptr*→*f*) ⟹ *p* = *q*
⟨*proof*⟩

lemma *ptr-val-field-convD*: *ptr-val p* = &(*q*→*f*) ⟹ *p* = *Ptr* &(*q*→*f*)
⟨*proof*⟩

lemma *ptr-val-field-conv*: *ptr-val p* = &(*q*→*f*) ⟷ *p* = *Ptr* &(*q*→*f*)
⟨*proof*⟩

lemma *ptr-val-array-index-convD*:
ptr-val p = *ptr-val* (*array-ptr-index q False j*) ⟹ *p* = *array-ptr-index q False j*
⟨*proof*⟩

lemma *ptr-val-conv'*: *ptr-coerce p* = *Ptr q* ⟷ *ptr-val p* = *q*
⟨*proof*⟩

lemma *ptr-val-conv*: *p* = *q* ⟷ *ptr-val p* = *ptr-val q*
⟨*proof*⟩

lemma *ptr-val-neq-conv*: *p* ≠ *q* ⟷ *ptr-val p* ≠ *ptr-val q*
⟨*proof*⟩

lemma *the-simple-lift-strong-eqI*:
fixes *p*::'*a*::*mem-type ptr*
fixes *q*::'*b*::*mem-type ptr*

assumes *eq*: ⋀*x1 x2*. *root-ptr-valid* (*hrs-htd s*) *q* ⟹
simple-lift s q = *Some x1* ⟹
(*simple-lift* (*hrs-mem-update* (*heap-update p v*) *s*) *q*) = *Some x2* ⟹
x1 = *x2*
shows *the* (*simple-lift*
(*hrs-mem-update* (*heap-update p v*) *s*) *q*) =
the (*simple-lift s q*)
⟨*proof*⟩

lemma *the-simple-lift-hval-eqI*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*::*'b*::*mem-type ptr*

assumes *eq*: *root-ptr-valid* (*hrs-htd s*) *q* ⟹
(*h-val* ((*heap-update p v*) (*hrs-mem s*)) *q*) = *h-val* (*hrs-mem s*) *q*

shows *the* (*simple-lift*
(*hrs-mem-update* (*heap-update p v*) *s*) *q*) =
the (*simple-lift s q*)
⟨*proof*⟩

lemma *h-t-valid-hrs-mem-update-pres*: *hrs-htd s*,*g* ⊨$_t$ *q* ⟹
hrs-htd ((*hrs-mem-update* (*heap-update p v*) *s*)), *g*⊨$_t$ *q*
⟨*proof*⟩

lemma *field-the-clift-hval-eqI*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*::*'b*::*mem-type ptr*
assumes *eq*: *hrs-htd s*, *c-guard* ⊨$_t$ *q* ⟹
f (*h-val* ((*heap-update p v*) (*hrs-mem s*)) *q*) = *f* (*h-val* (*hrs-mem s*) *q*)
shows *f* (*the* (*clift*
(*hrs-mem-update* (*heap-update p v*) *s*) *q*)) =
f (*the* (*clift s q*))
⟨*proof*⟩

lemma *the-clift-hval-eqI*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*::*'b*::*mem-type ptr*
assumes *eq*: *hrs-htd s*, *c-guard* ⊨$_t$ *q* ⟹
(*h-val* ((*heap-update p v*) (*hrs-mem s*)) *q*) = (*h-val* (*hrs-mem s*) *q*)
shows *the* (*clift*
(*hrs-mem-update* (*heap-update p v*) *s*) *q*) =
(*the* (*clift s q*))
⟨*proof*⟩

lemma *valid-root-footprint-same-type-cases*:
assumes *valid-p*: *valid-root-footprint d* (*ptr-val* (*p*::*'a*:: *mem-type ptr*)) (*typ-uinfo-t* (*TYPE*(*'a*)))
assumes *valid-q*: *valid-footprint d* (*ptr-val* (*q*::*'a*::*mem-type ptr*)) (*typ-uinfo-t* (*TYPE*(*'a*)))

assumes *eq-case*: *p* = *q* ⟹ *P*
assumes *disj-case*: *ptr-span p* ∩ *ptr-span q* = {} ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *valid-footprint-overlap-contained*:

assumes *valid-root-x*: *valid-footprint d x t*
assumes *valid-y*: *valid-footprint d y s*
assumes *overlap*: $\{x\ ..+\ size\text{-}td\ t\} \cap \{y\ ..+\ size\text{-}td\ s\} \neq \{\}$
shows $y \in \{x\ ..+\ size\text{-}td\ t\} \vee x \in \{y\ ..+\ size\text{-}td\ s\}$
⟨*proof*⟩

lemma *valid-footprint-same-type-cases*:
assumes *valid-p*: *valid-footprint d* (*ptr-val* (*p*::'*a*:: *mem-type ptr*)) (*typ-uinfo-t* (*TYPE*('*a*)))
assumes *valid-q*: *valid-footprint d* (*ptr-val* (*q*::'*a*::*mem-type ptr*)) (*typ-uinfo-t* (*TYPE*('*a*)))

assumes *eq-case*: $p = q \Longrightarrow P$
assumes *disj-case*: *ptr-span p* $\cap$ *ptr-span q* $= \{\} \Longrightarrow P$
shows *P*
⟨*proof*⟩

theorem *subfield-deref-append*:
fixes *q*::'*a*::*mem-type ptr*
fixes *p*::'*b*::*mem-type ptr*
assumes *base*: *ptr-val p* = &(*q*→*g*)
assumes *g*: *g* ∈ *set* (*field-names-u* (*typ-uinfo-t* (*TYPE*('*a*))) (*typ-uinfo-t* (*TYPE*('*b*))))
assumes *f*: *f* ∈ *set* (*all-field-names* (*typ-uinfo-t* (*TYPE*('*b*))))
shows &(*p*→*f*) = &(*q*→(*g*@*f*))
⟨*proof*⟩

lemmas *root-ptr-valid-same-type-cases* = *valid-root-footprint-same-type-cases* [*OF root-ptr-valid-valid-root-footprint h-t-valid-valid-footprint*]
lemmas *ptr-valid-same-type-cases* = *valid-footprint-same-type-cases* [*OF h-t-valid-valid-footprint h-t-valid-valid-footprint*]

theorem *root-ptr-valid-heap-update-other'*:
assumes *val-p*: *root-ptr-valid d* (*p*::'*a*::*mem-type ptr*)
assumes *val-q*: *d* ⊨$_t$ (*q*::'*b*::*mem-type ptr*)
assumes *not-subtype*: ¬ *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)
shows *h-val* (*heap-update p v h*) *q* = *h-val h q*
⟨*proof*⟩

theorem *root-ptr-valid-heap-update-field-other*:
assumes *val-p*: *root-ptr-valid d* (*p*::'*a*::*mem-type ptr*)
assumes *val-q*: *d* ⊨$_t$ (*q*::'*b*::*mem-type ptr*)
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*, *n*)
assumes *match*: *export-uinfo t* = *typ-uinfo-t TYPE*('*c*::*mem-type*)
assumes *not-subtype-b-c*: ¬ *TYPE*('*b*) ≤$_\tau$ *TYPE*('*c*)
assumes *not-subtype-c-b*: ¬ *TYPE*('*c*) ≤$_\tau$ *TYPE*('*b*)

shows *h-val* (*heap-update* (*PTR*(*'c*) &(*p*→*f*)) *v h*) *q* = *h-val h q*
⟨*proof*⟩

theorem *root-ptr-valid-heap-update-field-other'*:
assumes *val-p*: *root-ptr-valid d* (*p*::*'a*::*mem-type ptr*)
assumes *val-q*: *d* ⊨$_t$ (*q*::*'b*::*mem-type ptr*)
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'b*)) *f 0* = *Some* (*t*, *n*)
assumes *match*: *export-uinfo t* = *typ-uinfo-t TYPE*(*'c*::*mem-type*)
assumes *not-subtype-b-a*: ¬ *TYPE*(*'b*) ≤$_\tau$ *TYPE*(*'a*)
shows *h-val* (*heap-update p v h*) (*PTR* (*'c*) &(*q*→*f*)) = *h-val h* (*PTR* (*'c*) &(*q*→*f*))
⟨*proof*⟩

lemmas *root-ptr-valid-dist-type-cases* = *valid-root-footprint-dist-type-cases* [*OF root-ptr-valid-valid-root-footpri*
h-t-valid-valid-footprint,
consumes 3, *case-names Field Disjoint-Spans*]

theorem *ptr-valid-heap-update-field-disj*:
assumes *val-p*: *d* ⊨$_t$ (*p*::*'a*::*mem-type ptr*)
assumes *val-q*: *d* ⊨$_t$ (*q*::*'b*::*mem-type ptr*)
assumes *subtype*: *TYPE*(*'b*) ≤$_\tau$ *TYPE*(*'a*)
assumes *disj* [*rule-format*] :
∀*f*. *f* ∈ *set* (*field-names-u* (*typ-uinfo-t* (*TYPE*(*'a*))) (*typ-uinfo-t* (*TYPE*(*'b*))))
⟶ *q* ≠ *Ptr* (&(*p*→*f*))
shows *h-val* (*heap-update p v h*) *q* = *h-val h q*
⟨*proof*⟩

lemma *is-padding-tag-access-ti-eq*: *is-padding-tag s* ⟹ *access-ti s x bs* = *access-ti s y bs*
⟨*proof*⟩

lemma *is-padding-tag-access-ti*$_0$: *is-padding-tag s* ⟹ *access-ti*$_0$ *s x* = *replicate* (*size-td s*) *0*
⟨*proof*⟩

lemma *is-padding-tag-from-bytes-eq*: *is-padding-tag s* ⟹ *from-bytes* (*access-ti*$_0$ *s x*) = *from-bytes* (*access-ti*$_0$ *s y*)
⟨*proof*⟩

theorem *ptr-valid-heap-update-field-disj'*:
assumes *val-p*: *d* ⊨$_t$ (*p*::*'a*::*xmem-type ptr*)
assumes *val-q*: *d* ⊨$_t$ (*q*::*'b*::*xmem-type ptr*)
assumes *subtype*: *TYPE*(*'b*) ≤$_\tau$ *TYPE*(*'a*)
assumes *disj* [*rule-format*] :
∀*f*. *f* ∈ *set* (*field-names-no-padding* (*typ-info-t* (*TYPE*(*'a*))) (*export-uinfo* (*typ-info-t* (*TYPE*(*'b*))))) ⟶ *q* ≠ *Ptr* (&(*p*→*f*))

shows *h-val* (*heap-update p v h*) *q* = *h-val h q*
⟨*proof*⟩

lemma *is-padding-tag-update-ti-id*: *is-padding-tag s* ⟹ *update-ti s bs v* = *v*
⟨*proof*⟩

theorem *ptr-valid-heap-update-field-disj''*:
assumes *val-p*: *d* ⊨$_t$ (*p*::'*a*::*xmem-type ptr*)
assumes *val-q*: *d* ⊨$_t$ (*q*::'*b*::*xmem-type ptr*)
assumes *subtype*: *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)
assumes *fl-g*: *field-lookup* (*typ-info-t TYPE*('*a*)) *g 0* = *Some* (*t*, *n*)
assumes *match*: *export-uinfo t* = *typ-uinfo-t TYPE*('*c*::*xmem-type*)
assumes *disj* [*rule-format*] :
∀*f*. *f* ∈ *set* (*field-names-no-padding* (*typ-info-t* (*TYPE*('*a*))) (*export-uinfo* (*typ-info-t* (*TYPE*('*b*)))))
⟶ *ptr-span* (*PTR*('*b*) &(*p*→*f*)) ∩ *ptr-span* (*PTR*('*c*) &(*p*→*g*)) = {}
shows *h-val* (*heap-update q v h*) (*PTR*('*c*) &(*p*→*g*)) = *h-val h* (*PTR*('*c*) &(*p*→*g*))
⟨*proof*⟩

theorem *ptr-valid-heap-update-field-access-ti-disj*:
assumes *val-p*: *d* ⊨$_t$ (*p*::'*a*::*xmem-type ptr*)
assumes *val-q*: *d* ⊨$_t$ (*q*::'*b*::*xmem-type ptr*)
assumes *subtype*: *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)
assumes *disj* [*rule-format*] :
∀*f*. *f* ∈ *set* (*field-names-u* (*typ-uinfo-t* (*TYPE*('*a*))) (*typ-uinfo-t* (*TYPE*('*b*))))
⟶
(*access-ti* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) *f 0*)))
v
(*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*))) =
(*access-ti* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) *f 0*)))
(*h-val h p*)
(*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*)))
shows *h-val* (*heap-update p v h*) *q* = *h-val h q*
⟨*proof*⟩

lemma *access-ti-field-names-no-padding-to-field-names-u*:
assumes *val-p*: *d* ⊨$_t$ (*p*::'*a*::*xmem-type ptr*)
assumes *val-q*: *d* ⊨$_t$ (*q*::'*b*::*xmem-type ptr*)
assumes *subtype*: *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)
assumes *disj* [*rule-format*]:
∀*f*. *f* ∈ *set* (*field-names-no-padding* (*typ-info-t* (*TYPE*('*a*))) (*export-uinfo* (*typ-info-t* (*TYPE*('*b*))))) ⟶
(*access-ti* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) *f 0*)))
v
(*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*))) =
(*access-ti* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) *f 0*)))
(*h-val h p*)
(*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*)))
assumes *f-in*: *f* ∈ *set* (*field-names-u* (*typ-uinfo-t* (*TYPE*('*a*))) (*typ-uinfo-t* (*TYPE*('*b*))))

shows (*access-ti* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) *f 0*)))
v
(*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*))) =
(*access-ti* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) *f 0*)))
(*h-val h p*)
(*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*)))
⟨*proof*⟩

theorem *ptr-valid-heap-update-field-access-ti-disj'*:
assumes *val-p*: *d* ⊨$_t$ (*p*::'*a*::*xmem-type ptr*)
assumes *val-q*: *d* ⊨$_t$ (*q*::'*b*::*xmem-type ptr*)
assumes *subtype*: *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)
assumes *disj* :
∀*f*. *f* ∈ *set* (*field-names-no-padding* (*typ-info-t* (*TYPE*('*a*))) (*export-uinfo* (*typ-info-t* (*TYPE*('*b*))))) ⟶
(*access-ti* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) *f 0*)))
v
(*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*))) =
(*access-ti* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) *f 0*)))
(*h-val h p*)
(*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*)))
shows *h-val* (*heap-update p v h*) *q* = *h-val h q*
⟨*proof*⟩

theorem *ptr-valid-heap-update-subtype-field-access-ti-disj*:
assumes *val-p*: *d* ⊨$_t$ (*p*::'*a*::*xmem-type ptr*)
assumes *val-q*: *d* ⊨$_t$ (*q*::'*b*::*xmem-type ptr*)
assumes *subtype-b-a*: *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)
assumes *fl-g*: *field-lookup* (*typ-uinfo-t TYPE*('*b*)) *g 0* = *Some* (*typ-uinfo-t TYPE*('*c*::*xmem-type*), *n*)
assumes *disj* [*rule-format*] :
∀*f*. *f* ∈ *set* (*field-names-u* (*typ-uinfo-t* (*TYPE*('*a*))) (*typ-uinfo-t* (*TYPE*('*b*))))
⟶
(*access-ti* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) *f 0*)))
v
(*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*))) =
(*access-ti* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) *f 0*)))
(*h-val h p*)
(*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*)))
shows *h-val* (*heap-update p v h*) (*PTR*('*c*) (&(*q*→*g*))) = *h-val h* (*PTR*('*c*) (&(*q*→*g*)))
⟨*proof*⟩

corollary *ptr-valid-heap-update-subtype-field-access-ti-disj'*:
assumes *val-p*: *d* ⊨$_t$ (*p*::'*a*::*xmem-type ptr*)
assumes *val-q*: *d* ⊨$_t$ (*q*::'*b*::*xmem-type ptr*)
assumes *subtype-b-a*: *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)

assumes *fl-g*: *field-lookup* (*typ-info-t TYPE*('*b*)) *g 0* = *Some* (*s*, *n*)
assumes *match*: *export-uinfo s* = *typ-uinfo-t TYPE*('*c*::*xmem-type*)
assumes *disj* :
∀ *f*. *f* ∈ *set* (*field-names-no-padding* (*typ-info-t* (*TYPE*('*a*))) (*export-uinfo* (*typ-info-t* (*TYPE*('*b*))))) ⟶
(*access-ti* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) *f 0*)))
v
(*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*))) =
(*access-ti* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) *f 0*)))
(*h-val h p*)
(*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*)))
shows *h-val* (*heap-update p v h*) (*PTR*('*c*) (&(*q*→*g*))) = *h-val h* (*PTR*('*c*) (&(*q*→*g*)))
⟨*proof*⟩

lemma *array-ptr-index-field-lvalue-conv*:
fixes *p*:: ('*a*::*c-type*['*b*::*finite*]) *ptr*
assumes *i-bound*: *i* < *CARD*('*b*)
shows (*array-ptr-index p False i*) = (*PTR*('*a*) &(*p*→[*replicate i CHR* ''*1*''])
⟨*proof*⟩

lemma *field-lvalue-array-index*:
fixes *p*:: ('*a*::*c-type*['*b*::*finite*]) *ptr*
shows *i* < *CARD*('*b*) ⟹ &(*p*→[*replicate i CHR* ''*1*'']) =
ptr-val (*PTR-COERCE*('*a*['*b*] → '*a*) *p* +$_p$ *int i*)
⟨*proof*⟩

lemma *field-lvalue-array-index'*:
fixes *p*:: ('*a*::*c-type*['*b*::*finite*]) *ptr*
shows *i* < *CARD*('*b*) ⟹
PTR('*a*) &(*p*→[*replicate i CHR* ''*1*'']) =
(*PTR-COERCE*('*a*['*b*] → '*a*) *p* +$_p$ *int i*)
⟨*proof*⟩

thm *field-lvalue-append*
corollary *ptr-valid-heap-update-subtype-array-access-ti-disj*:
assumes *val-p*: *d* ⊨$_t$ (*p*::'*a*::*xmem-type ptr*)
assumes *val-q*: *d* ⊨$_t$ (*q*::('*b*::*array-outer-max-size*['*c*::*array-max-count*]) *ptr*)
assumes *subtype-b-a*: *TYPE*('*b*['*c*]) ≤$_\tau$ *TYPE*('*a*)
assumes *i-bound*: *i* < *CARD*('*c*)
assumes *disj* :
∀ *f*. *f* ∈ *set* (*field-names-no-padding* (*typ-info-t* (*TYPE*('*a*))) (*export-uinfo* (*typ-info-t* (*TYPE*('*b*['*c*])))))) ⟶
(*access-ti* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) *f 0*)))
v
(*heap-list h* (*size-of TYPE*('*b*['*c*])) (*ptr-val q*))) =
(*access-ti* (*fst* (*the* (*field-lookup* (*typ-info-t TYPE*('*a*)) *f 0*)))
(*h-val h p*)
(*heap-list h* (*size-of TYPE*('*b*['*c*])) (*ptr-val q*)))

shows *h-val* (*heap-update p v h*) (*array-ptr-index q False i*) = *h-val h* (*array-ptr-index q False i*)
⟨*proof*⟩

theorem *ptr-valid-heap-update-subtype-field-access-ti-indep*:
assumes *val-p*: *d* ⊨$_t$ (*p*::'*a*::*xmem-type ptr*)
assumes *val-q*: *d* ⊨$_t$ (*q*::'*a*::*xmem-type ptr*)
assumes *fl-f*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*, *n*)
assumes *match*: *export-uinfo t* = *typ-uinfo-t* (*TYPE*('*b*::*xmem-type*))
assumes *disj*:
(*access-ti t v* (*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*))) =
(*access-ti t* (*h-val h p*) (*heap-list h* (*size-of TYPE*('*b*)) (*ptr-val q*)))
shows *h-val* (*heap-update p v h*) (*PTR*('*b*) (&(*q*→*f*))) = *h-val h* (*PTR*('*b*) (&(*q*→*f*)))
⟨*proof*⟩

lemmas *clift-field'* = *clift-field* [*simplified fold-typ-uinfo-t*]

lemma *field-lookup-cons-Some*:
fixes *t*::('*a*, '*b*) *typ-desc*
and *st*::('*a*, '*b*) *typ-struct*
and *ts*::('*a*, '*b*) *typ-tuple list*
and *x*::('*a*, '*b*) *typ-tuple*
shows
wf-desc t ⟹ *field-lookup t* (*f*#*fs*) *n* = *Some* (*s*, *m*) ⟹
∃ *w k*. *field-lookup t* [*f*] *n* = *Some* (*w*, *k*) ∧ *field-lookup w fs k* = *Some* (*s*, *m*)
wf-desc-struct st ⟹ *field-lookup-struct st* (*f* # *fs*) *n* = *Some* (*s*, *m*) ⟹
∃ *w k*. *field-lookup-struct st* [*f*] *n* = *Some* (*w*, *k*) ∧ *field-lookup w fs k* = *Some* (*s*, *m*)
wf-desc-list ts ⟹ *field-lookup-list ts* (*f*#*fs*) *n* = *Some* (*s*, *m*) ⟹
∃ *w k*. *field-lookup-list ts* [*f*] *n* = *Some* (*w*, *k*) ∧ *field-lookup w fs k* = *Some* (*s*, *m*)
wf-desc-tuple x ⟹ *field-lookup-tuple x* (*f* # *fs*) *n* = *Some* (*s*, *m*) ⟹
f = *dt-snd x* ∧
field-lookup (*dt-fst x*) *fs n* = *Some* (*s*, *m*)
⟨*proof*⟩

lemma *field-offset-append*:
assumes *f*: *field-lookup* (*typ-uinfo-t TYPE*('*a*)) *f 0* = *Some* (*typ-uinfo-t TYPE*('*b*), *n*)
assumes *g*: *field-lookup* (*typ-uinfo-t TYPE*('*b*)) *g 0* = *Some x*
shows *field-offset TYPE*('*a*::*mem-type*) (*f* @ *g*) =
field-offset (*TYPE*('*a*::*mem-type*)) *f* + *field-offset* (*TYPE*('*b*::*mem-type*)) *g*
⟨*proof*⟩

lemma *field-lookup-struct-offset-shift*:
field-lookup-struct st f m = *Some* (*s*, *k*) ⟹ *field-lookup-struct st f n* = *Some* (*s*, *k* + *n* − *m*)

⟨*proof*⟩

lemma *field-lookup-list-offset-shift*:
field-lookup-list ts f m = *Some* (*s*, *k*) ⟹ *field-lookup-list ts f n* = *Some* (*s*, *k* + *n* − *m*)
⟨*proof*⟩

lemma *field-lookup-tuple-offset-shift*:
field-lookup-tuple x f m = *Some* (*s*, *k*) ⟹ *field-lookup-tuple x f n* = *Some* (*s*, *k* + *n* − *m*)
⟨*proof*⟩

lemma *h-val-field-update*:
fixes *p*::*'a*::*mem-type ptr*
and *q*::*'a*::*mem-type ptr*
and *v*:: *'b*::*mem-type*
assumes *fl*: *field-ti TYPE*(*'a*) *f* = *Some t*
assumes *match*: *export-uinfo t* = *typ-uinfo-t TYPE*(*'b*)
and *valid-p*: *hrs-htd h* ⊨$_t$ *p*
and *valid-q*: *hrs-htd h* ⊨$_t$ *q*
shows (*h-val* (*heap-update* (*Ptr* &(*p*→*f*)) *v* (*hrs-mem h*)) *q*) =
(*if p* = *q then field-update* (*field-desc t*) (*to-bytes-p v*) (*h-val* (*hrs-mem h*) *p*) *else* (*h-val* (*hrs-mem h*) *q*))
⟨*proof*⟩

theorem *root-ptr-valid-field-disj-type-h-val-heap-update*:
assumes *valid-p*: *root-ptr-valid* (*hrs-htd h*) (*p*::*'b*::*mem-type ptr*)
assumes *valid-q*: *root-ptr-valid* (*hrs-htd h*) (*q*::*'a*::*mem-type ptr*)
assumes *neq*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*'b*)
assumes *f*: *field-lookup* (*typ-info-t* (*TYPE*(*'a*))) *f 0* = *Some* (*s*, *n*)
assumes *match*: *export-uinfo s* = *typ-uinfo-t* (*TYPE*(*'c*))
shows *h-val* (*heap-update* (*PTR*(*'c*::*mem-type*) &(*q*→*f*)) *v* (*hrs-mem h*)) *p* =
h-val (*hrs-mem h*) *p*
⟨*proof*⟩

lemma *ptr-empty-qualified-field-name-conv*:
fixes *p*::*'a*::*c-type ptr*
shows *p* = *PTR* (*'a*) &(*p*→[])
⟨*proof*⟩

theorem *root-ptr-valid-field-disj-h-val-heap-update-eq*:
assumes *valid-p*: *root-ptr-valid d* (*p*::*'a*::*mem-type ptr*)
assumes *valid-q*: *root-ptr-valid d* (*q*::*'b*::*mem-type ptr*)
assumes *f*: *field-lookup* (*typ-info-t* (*TYPE*(*'a*))) *f 0* = *Some* (*t*, *n*)
assumes *g*: *field-lookup* (*typ-info-t* (*TYPE*(*'b*))) *g 0* = *Some* (*s*, *m*)
assumes *match-t*: *export-uinfo t* = *typ-uinfo-t* (*TYPE*(*'c*))

assumes *match-s*: *export-uinfo s = typ-uinfo-t (TYPE('d))*
assumes *disj*: *typ-uinfo-t (TYPE('a)) = (typ-uinfo-t (TYPE('b))) ⟹*
ptr-val p = ptr-val q ⟹
ptr-span (PTR('c::mem-type) &(p→f)) ∩ ptr-span (PTR('d::mem-type) &(q→g)) = {}
shows *h-val (heap-update (PTR('c::mem-type) &(p→f)) v h) (PTR('d::mem-type) &(q→g)) =*
h-val h (PTR('d::mem-type) &(q→g))
⟨*proof*⟩

definition *PTR-MATCH p q = (p = q)*

lemma *PTR-MATCH-field*:
PTR-MATCH (PTR('a::c-type) &(p→f)) (PTR('a) &(p→f))
⟨*proof*⟩

lemma *PTR-MATCH-dummy*:
PTR-MATCH (p::'a::c-type ptr) (PTR('a) &(p→[]))
⟨*proof*⟩

theorem *root-ptr-valid-field-disj-h-val-heap-update-eq'*:
assumes *p'*: *PTR-MATCH p' (PTR('c) &(p→f))*
assumes *q'*: *PTR-MATCH q' (PTR('d) &(q→g))*
assumes *valid-p*: *root-ptr-valid d (p::'a::mem-type ptr)*
assumes *valid-q*: *root-ptr-valid d (q::'b::mem-type ptr)*
assumes *disj*: *typ-uinfo-t (TYPE('a)) = (typ-uinfo-t (TYPE('b))) ⟹*
ptr-val p = ptr-val q ⟹
ptr-span (PTR('c) &(p→f)) ∩ ptr-span (PTR('d) &(q→g)) = {}
assumes *match-s*: *export-uinfo s = typ-uinfo-t (TYPE('d::mem-type))*
assumes *match-t*: *export-uinfo t = typ-uinfo-t (TYPE('c::mem-type))*
assumes *g*: *field-lookup (typ-info-t (TYPE('b))) g 0 = Some (s, n)*
assumes *f*: *field-lookup (typ-info-t (TYPE('a))) f 0 = Some (t, m)*
shows *h-val (heap-update p' v h) q' = h-val h q'*
⟨*proof*⟩

lemma *field-lookup-single-field-disj'*:
fixes *t*::('*a*, '*b*) *typ-desc*
and *st*::('*a*, '*b*) *typ-struct*
and *ts*::('*a*, '*b*) *typ-tuple list*
and *x*::('*a*, '*b*) *typ-tuple*
shows
field-lookup t [f] n = Some (s, n + m) ⟹
field-lookup t [g] n = Some (s', n + k) ⟹ f ≠ g ⟹
{m ..< m + size-td s} ∩ {k ..<k + size-td s'} = {}
field-lookup-struct st [f] n = Some (s, n + m) ⟹

field-lookup-struct st [*g*] *n* = *Some* (*s'*, *n* + *k*) ⟹ *f* ≠ *g* ⟹
{*m* ..< *m* + *size-td s*} ∩ {*k* ..< *k* + *size-td s'*} = {}
field-lookup-list ts [*f*] *n* = *Some* (*s*, *n* + *m*) ⟹
field-lookup-list ts [*g*] *n* = *Some* (*s'*, *n* + *k*) ⟹ *f* ≠ *g* ⟹
{*m* ..< *m* + *size-td s*} ∩ {*k* ..< *k* + *size-td s'*} = {}
field-lookup-tuple x [*f*] *n* = *Some* (*s*, *n* + *m*) ⟹
field-lookup-tuple x [*g*] *n* = *Some* (*s'*, *n* + *k*) ⟹ *f* ≠ *g* ⟹
{*m* ..< *m* + *size-td s*} ∩ {*k* ..< *k* + *size-td s'*} = {}
⟨*proof*⟩

lemma *field-lookup-single-field-disj*:
assumes *fl-f*: *field-lookup t* [*f*] *0* = *Some* (*s*, *m*)
assumes *fl-g*: *field-lookup t* [*g*] *0* = *Some* (*s'*, *k*)
assumes *neq*: *f* ≠ *g*
shows {*m* ..<*m* + *size-td s*} ∩ {*k* ..<*k* + *size-td s'*} = {}
⟨*proof*⟩

lemma *field-lookup-non-prefix-disj*:
assumes *wf*: *wf-desc t*
assumes *f*: *field-lookup t f 0* = *Some* (*tf*, *n*)
assumes *g*: *field-lookup t g 0* = *Some* (*tg*, *m*)
assumes *g-f*:¬ *prefix g f*
assumes *f-g*: ¬ *prefix f g*
shows {*n* ..<*n* + *size-td tf*} ∩ {*m* ..<*m* + *size-td tg*} = {}
⟨*proof*⟩

theorem *root-ptr-valid-non-prefix-disj*:
assumes *valid-p*: *root-ptr-valid d* (*p*::'*a*::*mem-type ptr*)
assumes *f*: *field-lookup* (*typ-info-t* (*TYPE*('*a*::*mem-type*))) *f 0* = *Some* (*t*, *n*)
assumes *g*: *field-lookup* (*typ-info-t* (*TYPE*('*a*::*mem-type*))) *g 0* = *Some* (*s*, *m*)
assumes *match-t*: *export-uinfo t* = *typ-uinfo-t* (*TYPE*('*c*))
assumes *match-s*: *export-uinfo s* = *typ-uinfo-t* (*TYPE*('*d*))
assumes *non-prefix-f-g*: ¬ *prefix f g*
assumes *non-prefix-g-f*: ¬ *prefix g f*
shows *ptr-span* (*PTR*('*c*::*mem-type*) &(*p*→*f*)) ∩ *ptr-span* (*PTR*('*d*::*mem-type*)
&(*p*→*g*)) = {}
⟨*proof*⟩

theorem *root-ptr-valid-non-prefix-h-val-heap-update-eq'*:

assumes *p'*: *PTR-MATCH p'* (*PTR*('*c*) &(*p*→*f*))
assumes *q'*: *PTR-MATCH q'* (*PTR*('*d*) &(*q*→*g*))
assumes *valid-p*: *root-ptr-valid d* (*p*::'*a*::*mem-type ptr*)
assumes *valid-q*: *root-ptr-valid d* (*q*::'*b*::*mem-type ptr*)
assumes *disj*: *typ-uinfo-t* (*TYPE*('*a*)) = (*typ-uinfo-t* (*TYPE*('*b*))) ⟹
ptr-val p = *ptr-val q* ⟹

$\neg$ *prefix f g* $\wedge$ $\neg$ *prefix g f*
assumes *match-s*: *export-uinfo s* = *typ-uinfo-t* (*TYPE*('*d*::*mem-type*))
assumes *match-t*: *export-uinfo t* = *typ-uinfo-t* (*TYPE*('*c*::*mem-type*))
assumes *g*: *field-lookup* (*typ-info-t* (*TYPE*('*b*))) *g 0* = *Some* (*s*, *n*)
assumes *f*: *field-lookup* (*typ-info-t* (*TYPE*('*a*))) *f 0* = *Some* (*t*, *m*)
shows *h-val* (*heap-update p' v h*) *q'* = *h-val h q'*
⟨*proof*⟩

lemma *field-tag-sub'*:
assumes *fl*: *field-lookup* (*typ-uinfo-t TYPE*('*a*::*mem-type*)) *f 0* = *Some* (*t*, *n*)
shows {&(*p*::'*a ptr*→*f*)..+*size-td t*} ⊆ *ptr-span p*
⟨*proof*⟩

lemma *all-field-names-field-lookup*:
assumes *f* ∈ *set* (*all-field-names* (*typ-uinfo-t TYPE*('*a*::*mem-type*)))
shows ∃ *t n*. *field-lookup* (*typ-uinfo-t TYPE*('*a*::*mem-type*)) *f 0* = *Some* (*t*, *n*)
⟨*proof*⟩

lemma *field-lvalue-same-type-conv*:
assumes *valid-p*: *d*,*g* ⊨$_t$ (*p*::'*a*::*mem-type ptr*)
assumes *valid-q*: *d*,*g* ⊨$_t$ (*q*::'*a*::*mem-type ptr*)
assumes *f1*: *f1* ∈ *set* (*all-field-names* (*typ-uinfo-t TYPE*('*a*)))
assumes *f2*: *f2* ∈ *set* (*all-field-names* (*typ-uinfo-t TYPE*('*a*)))
shows (&(*p*→*f1*) = &(*q*→*f2*)) ⟷ (*p* = *q* ∧ *field-offset* (*TYPE*('*a*)) *f1* = *field-offset TYPE*('*a*) *f2*)
⟨*proof*⟩

lemma *field-lvalue-same-root-conv*:
assumes *f1*: *field-lookup* (*typ-info-t TYPE*('*a*::*mem-type*)) *f1 0* = *Some*(*t1*, *n1*)

assumes *f2*: *field-lookup* (*typ-info-t TYPE*('*a*::*mem-type*)) *f2 0* = *Some*(*t2*, *n2*)

shows (&(*p*::'*a ptr*→*f1*) = &(*p*→*f2*)) ⟷ (*n1* = *n2*)
⟨*proof*⟩

lemma *field-lvalue-same-type-conv'*:
assumes *valid-p*: *d*,*g* ⊨$_t$ (*p*::'*a*::*mem-type ptr*)
assumes *valid-q*: *d*,*g* ⊨$_t$ (*q*::'*a*::*mem-type ptr*)
assumes *f1*: *f1* ∈ *set* (*all-field-names-no-padding* (*typ-info-t TYPE*('*a*)))
assumes *f2*: *f2* ∈ *set* (*all-field-names-no-padding* (*typ-info-t TYPE*('*a*)))
shows (&(*p*→*f1*) = &(*q*→*f2*)) ⟷ (*p* = *q* ∧ *field-offset* (*TYPE*('*a*)) *f1* = *field-offset TYPE*('*a*) *f2*)
⟨*proof*⟩

lemma *field-lvalue-same-root-conv'*:
assumes *f1*: *f1* ∈ *set* (*all-field-names* (*typ-uinfo-t TYPE*('*a*::*mem-type*)))
assumes *f2*: *f2* ∈ *set* (*all-field-names* (*typ-uinfo-t TYPE*('*a*)))

shows $(\&(p::'a\ ptr \rightarrow f1) = \&(p \rightarrow f2)) \longleftrightarrow (field\text{-}offset\ (TYPE('a))\ f1 = field\text{-}offset\ TYPE('a)\ f2)$
⟨*proof*⟩

lemma *field-lvalue-same-root-conv''*:
assumes *f1*: $f1 \in set\ (all\text{-}field\text{-}names\text{-}no\text{-}padding\ (typ\text{-}info\text{-}t\ TYPE('a::mem\text{-}type)))$

assumes *f2*: $f2 \in set\ (all\text{-}field\text{-}names\text{-}no\text{-}padding\ (typ\text{-}info\text{-}t\ TYPE('a)))$
shows $(\&(p::'a\ ptr \rightarrow f1) = \&(p \rightarrow f2)) \longleftrightarrow (field\text{-}offset\ (TYPE('a))\ f1 = field\text{-}offset\ TYPE('a)\ f2)$
⟨*proof*⟩

lemma *disj-ptr-span-field-neq*:
assumes *disj*: $ptr\text{-}span\ (p::'a::mem\text{-}type\ ptr) \cap ptr\text{-}span\ (q::'b::mem\text{-}type\ ptr) = \{\}$
assumes *fl1*: $field\text{-}lookup\ (typ\text{-}info\text{-}t\ TYPE('a))\ f1\ 0 = Some\ (t1, n1)$
assumes *fl2*: $field\text{-}lookup\ (typ\text{-}info\text{-}t\ TYPE('b))\ f2\ 0 = Some\ (t2, n2)$
shows $\&(p \rightarrow f1) \neq \&(q \rightarrow f2)$
⟨*proof*⟩

lemma *disj-ptr-span-field-neq''*:
assumes *disj*: $ptr\text{-}span\ (p::'a::mem\text{-}type\ ptr) \cap ptr\text{-}span\ (q::'b::mem\text{-}type\ ptr) = \{\}$
assumes *f1*: $f1 \in set\ (all\text{-}field\text{-}names\ (typ\text{-}uinfo\text{-}t\ TYPE('a)))$
assumes *f2*: $f2 \in set\ (all\text{-}field\text{-}names\ (typ\text{-}uinfo\text{-}t\ TYPE('b)))$
shows $\&(p \rightarrow f1) \neq \&(q \rightarrow f2)$
⟨*proof*⟩

lemma *disj-ptr-span-field-neq'*:
assumes *disj*: $ptr\text{-}span\ (p::'a::mem\text{-}type\ ptr) \cap ptr\text{-}span\ (q::'b::mem\text{-}type\ ptr) = \{\}$
assumes *f1*: $f1 \in set\ (all\text{-}field\text{-}names\text{-}no\text{-}padding\ (typ\text{-}info\text{-}t\ TYPE('a)))$
assumes *f2*: $f2 \in set\ (all\text{-}field\text{-}names\text{-}no\text{-}padding\ (typ\text{-}info\text{-}t\ TYPE('b)))$
shows $\&(p \rightarrow f1) \neq \&(q \rightarrow f2)$
⟨*proof*⟩

lemma *disj-ptr-span-field-disj-ptr-span*:
assumes *disj*: $ptr\text{-}span\ (p::'a::mem\text{-}type\ ptr) \cap ptr\text{-}span\ (q::'b::mem\text{-}type\ ptr) = \{\}$
assumes *f1*: $f1 \in set\ (field\text{-}names\text{-}u\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ (typ\text{-}uinfo\text{-}t\ (TYPE('c::mem\text{-}type))))$
assumes *f2*: $f2 \in set\ (field\text{-}names\text{-}u\ (typ\text{-}uinfo\text{-}t\ TYPE('b))\ (typ\text{-}uinfo\text{-}t\ (TYPE('d::mem\text{-}type))))$
shows $ptr\text{-}span\ (PTR('c)\ \&(p \rightarrow f1)) \cap ptr\text{-}span\ (PTR('d)\ (\&(q \rightarrow f2))) = \{\}$
⟨*proof*⟩

lemma *disj-ptr-span-field-disj-ptr-span'*:
assumes *disj*: $ptr\text{-}span\ (p::'a::mem\text{-}type\ ptr) \cap ptr\text{-}span\ (q::'b::mem\text{-}type\ ptr) = \{\}$
assumes *f1*: $field\text{-}lookup\ (typ\text{-}info\text{-}t\ (TYPE('a)))\ f1\ 0 = Some\ (s, n)$

assumes *match-s*: *export-uinfo s* = *typ-uinfo-t* (*TYPE*('*c*::*mem-type*))
assumes *f2*: *field-lookup* (*typ-info-t* (*TYPE*('*b*))) *f2 0* = *Some* (*t*, *m*)
assumes *match-t*: *export-uinfo t* = *typ-uinfo-t* (*TYPE*('*d*::*mem-type*))
shows {&(*p*→*f1*)..+*size-of TYPE*('*c*)} ∩ {&(*q*→*f2*)..+*size-of TYPE*('*d*)} = {}
⟨*proof*⟩

lemma *disj-ptr-span-field-disj-ptr-span''*:
assumes *disj*: *ptr-span* (*p*::'*a*::*mem-type ptr*) ∩ *ptr-span* (*q*::'*b*::*mem-type ptr*) ≡ {}
assumes *f1*: *field-lookup* (*typ-info-t* (*TYPE*('*a*))) *f1 0* = *Some* (*s*, *n*)
assumes *match-s*: *export-uinfo s* = *typ-uinfo-t* (*TYPE*('*c*::*mem-type*))
assumes *f2*: *field-lookup* (*typ-info-t* (*TYPE*('*b*))) *f2 0* = *Some* (*t*, *m*)
assumes *match-t*: *export-uinfo t* = *typ-uinfo-t* (*TYPE*('*d*::*mem-type*))
shows {&(*p*→*f1*)..+*size-of TYPE*('*c*)} ∩ {&(*q*→*f2*)..+*size-of TYPE*('*d*)} = {}
⟨*proof*⟩

lemma *field-lvalue-empty*:
&(*p*→[]) = *ptr-val p*
⟨*proof*⟩

lemma *disj-ptr-span-field-disj-ptr-span-root1*:
assumes *disj*: *ptr-span* (*p*::'*a*::*mem-type ptr*) ∩ *ptr-span* (*q*::'*b*::*mem-type ptr*) ≡ {}
assumes *f1*: *field-lookup* (*typ-info-t* (*TYPE*('*a*))) *f1 0* = *Some* (*s*, *n*)
assumes *match-s*: *export-uinfo s* = *typ-uinfo-t* (*TYPE*('*c*::*mem-type*))
shows {&(*p*→*f1*)..+*size-of TYPE*('*c*)} ∩ *ptr-span q* ≡ {}
⟨*proof*⟩

lemma *disj-ptr-span-field-disj-ptr-span-root2*:
assumes *disj*: *ptr-span* (*p*::'*a*::*mem-type ptr*) ∩ *ptr-span* (*q*::'*b*::*mem-type ptr*) ≡ {}
assumes *f2*: *field-lookup* (*typ-info-t* (*TYPE*('*b*))) *f2 0* = *Some* (*t*, *m*)
assumes *match-t*: *export-uinfo t* = *typ-uinfo-t* (*TYPE*('*d*::*mem-type*))
shows *ptr-span p* ∩ {&(*q*→*f2*)..+*size-of TYPE*('*d*)} = {}
⟨*proof*⟩

lemma *root-ptr-valid-disj-field-lvalue-conv*:
assumes *valid-p*: *root-ptr-valid d* (*p*::'*a*::*mem-type ptr*)
assumes *valid-q*: *root-ptr-valid d* (*q*::'*b*::*mem-type ptr*)
assumes *neq*: *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*('*b*)
assumes *f1*: *f1* ∈ *set* (*all-field-names* (*typ-uinfo-t TYPE*('*a*)))
assumes *f2*: *f2* ∈ *set* (*all-field-names* (*typ-uinfo-t TYPE*('*b*)))
shows &(*p*→*f1*) = &(*q*→*f2*) = *False*
⟨*proof*⟩

lemma *root-ptr-valid-disj-field-lvalue-conv'*:
assumes *valid-p*: *root-ptr-valid d* (*p*::'*a*::*mem-type ptr*)
assumes *valid-q*: *root-ptr-valid d* (*q*::'*b*::*mem-type ptr*)
assumes *neq*: *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*('*b*)

assumes *f1*: *f1* ∈ *set* (*all-field-names* (*typ-info-t TYPE*('*a*)))
assumes *f2*: *f2* ∈ *set* (*all-field-names* (*typ-info-t TYPE*('*b*)))
shows &(*p*→*f1*) = &(*q*→*f2*) = *False*
⟨*proof*⟩

lemma *intvl-non-zero-non-empty*: *0* < *n* ⟹ {*p*..+*n*} ≠ {}
⟨*proof*⟩

lemma *disj-inter-swap*: *A* ∩ *B* = {} ⟹ *B* ∩ *A* = {}
⟨*proof*⟩

lemma *h-val-clift-field*:
clift hp (*Ptr* &(*p*→*f*)) = *Some v* ⟹ *v* = *h-val* (*hrs-mem hp*) (*Ptr* &(*p*→*f*))
⟨*proof*⟩

lemma *valid-root-footprints-no-overlap*:
assumes *valid-root-footprint d* (*ptr-val* (*p*::'*a*::*mem-type ptr*)) (*typ-uinfo-t TYPE*('*a*))
assumes *valid-root-footprint d* (*ptr-val* (*q*::'*b*::*mem-type ptr*)) (*typ-uinfo-t TYPE*('*b*))
assumes *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*('*b*)
shows *ptr-span p* ∩ *ptr-span q* = {}
⟨*proof*⟩

lemma *valid-root-footprints-overlap*:
assumes *valid-root-footprint d* (*ptr-val* (*p*::'*a*::*mem-type ptr*)) (*typ-uinfo-t TYPE*('*a*))
assumes *valid-root-footprint d* (*ptr-val* (*q*::'*b*::*mem-type ptr*)) (*typ-uinfo-t TYPE*('*b*))
assumes *ptr-span p* ∩ *ptr-span q* ≠ {}
shows (*typ-uinfo-t TYPE*('*a*)) = (*typ-uinfo-t TYPE*('*b*))
⟨*proof*⟩

lemma *field-lookup-same-type-empty*:
field-lookup t f n = *Some* (*s*, *m*) ⟹ *s* = *t* ⟷ *f* = []
field-lookup-struct st f n = *Some* (*s*, *m*) ⟹ *f* ≠ []
field-lookup-list ts f n = *Some* (*s*, *m*) ⟹ *f* ≠ []
field-lookup-tuple td f n = *Some* (*s*, *m*) ⟹ *f* ≠ []
⟨*proof*⟩

lemma *root-ptr-valid-root-only*:
assumes *valid*: *root-ptr-valid d* (*p*::'*a*::*mem-type ptr*)
assumes *non-empty*: *f* ≠ []
assumes *f*: *f* ∈ *set* (*field-names-u* (*typ-uinfo-t TYPE*('*a*)) (*typ-uinfo-t TYPE*('*b*)))

shows *root-ptr-valid d* (*PTR*('*b*::*mem-type*) &(*p*→*f*)) = *False*
⟨*proof*⟩

lemma *root-ptr-valid-root-only'*:
assumes *valid*: *root-ptr-valid d* (*p*::'*a*::*mem-type ptr*)
assumes *non-empty*: *f* ≠ []

assumes *f*: *f* ∈ *set* (*field-names* (*typ-info-t TYPE*('*a*)) (*export-uinfo* (*typ-info-t TYPE*('*b*))))
shows *root-ptr-valid d* (*PTR*('*b*::*mem-type*) &(*p*→*f*)) = *False*
⟨*proof*⟩

lemma *disj-subset*:
assumes $A \cap B = \{\}$
assumes $A' \subseteq A$
assumes $B' \subseteq B$
shows $A' \cap B' = \{\}$
⟨*proof*⟩

lemma *disj-intvl-subset*:
assumes *disj*: {*p*..+*n1*} ∩ {*q*..+*m1*} = {}
assumes *le1*: *n2* + *off1* ≤ *n1*
assumes *le2*: *m2* + *off2* ≤ *m1*
shows {*p* + *of-nat off1*..+*n2*} ∩ {*q* + *of-nat off2*..+*m2*} = {}
⟨*proof*⟩

lemma *disj-intvl-field'*:
assumes *disj*: {*ptr-val p*..+*n1*} ∩ {*ptr-val q*..+*m1*} = {}
assumes *f*: &(*p*→*f*) = *ptr-val p* + (*of-nat off1*)
assumes *g*: &(*q*→*g*) = *ptr-val q* + (*of-nat off2*)
assumes *le1*: *n2* + *off1* ≤ *n1*
assumes *le2*: *m2* + *off2* ≤ *m1*
shows {&(*p*→*f*)..+*n2*} ∩ {&(*q*→*g*)..+*m2*} = {}
⟨*proof*⟩

lemma *disj-intvl-field*:
assumes *disj*: {*ptr-val* (*p*::'*a*::*mem-type ptr*)..+*n1*} ∩ {*ptr-val* (*q*::'*b*::*mem-type ptr*)..+*m1*} = {}
assumes *le1*: *n2* + (*field-offset TYPE*('*a*) *f*) ≤ *n1*
assumes *le2*: *m2* + (*field-offset TYPE*('*b*) *g*) ≤ *m1*
shows {&(*p*→*f*)..+*n2*} ∩ {&(*q*→*g*)..+*m2*} = {}
⟨*proof*⟩

lemma *intvl-fields-disj1*:
assumes *f*: &(*p*→*f*) = *ptr-val p* + *off1*
assumes *g*: &(*p*→*g*) = *ptr-val p* + *off2*
assumes *n-sz*: *unat off1* + *n* ≤ *addr-card*
assumes *m-sz*: *unat off2* + *m* ≤ *addr-card*
assumes *le*: *unat off1* ≤ *unat off2*
assumes *disj*: *unat off1* + *n* ≤ *unat off2*
shows{&(*p*→*f*)..+*n*} ∩ {&(*p*→*g*)..+*m*} = {}
⟨*proof*⟩

lemma *intvl-fields-disj*:

assumes *f*: &(*p*→*f*) = *ptr-val p* + *off1*
assumes *g*: &(*p*→*g*) = *ptr-val p* + *off2*
assumes *n-sz*: *unat off1* + *n* ≤ *addr-card*
assumes *m-sz*: *unat off2* + *m* ≤ *addr-card*
assumes *disj*: *if unat off1* ≤ *unat off2 then unat off1* + *n* ≤ *unat off2 else unat off2* + *m* ≤ *unat off1*
shows{&(*p*→*f*)..+*n*} ∩ {&(*p*→*g*)..+*m*} = {}
⟨*proof*⟩

lemma *intvl-fields-disj-calculation*:
assumes *f*: &(*p*→*f*) = *ptr-val p* + *off1*
assumes *g*: &(*p*→*g*) = *ptr-val p* + *off2*
assumes *uoff1*: *unat off1* = *uoff1*
assumes *uoff2*: *unat off2* = *uoff2*
assumes *n-sz*: *uoff1* + *n* ≤ *addr-card*
assumes *m-sz*: *uoff2* + *m* ≤ *addr-card*
assumes *disj*: *if uoff1* ≤ *uoff2 then uoff1* + *n* ≤ *uoff2 else uoff2* + *m* ≤ *uoff1*
shows {&(*p*→*f*)..+*n*} ∩ {&(*p*→*g*)..+*m*} = {}
⟨*proof*⟩

lemma *disjoint-field-lvalue-propagation-right*:
fixes *q*::*'a*::*mem-type ptr*
assumes *disj*: *A* ∩ {*ptr-val q*..+*m*} = {}
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*t*, *k*)
assumes *m*: *m* = *size-of TYPE*(*'a*)
assumes *n*: *n* = *size-td t*
shows *A* ∩ {&(*q*→*f*)..+*n*} = {}
⟨*proof*⟩

lemma *disjoint-field-lvalue-propagation-left*:
fixes *q*::*'a*::*mem-type ptr*
assumes *disj*: {*ptr-val q*..+*m*} ∩ *A* = {}
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*t*, *k*)
assumes *m*: *m* = *size-of TYPE*(*'a*)
assumes *n*: *n* = *size-td t*
shows {&(*q*→*f*)..+*n*} ∩ *A* = {}
⟨*proof*⟩

lemma *disjoint-field-lvalue-propagation-both*:
fixes *q*::*'a*::*mem-type ptr*
fixes *p*::*'b*::*mem-type ptr*
assumes *disj*: {*ptr-val q*..+*m1*} ∩ {*ptr-val p*..+*m2*} = {}
assumes *fl1*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f1 0* = *Some* (*t1*, *k1*)
assumes *fl2*: *field-lookup* (*typ-info-t TYPE*(*'b*)) *f2 0* = *Some* (*t2*, *k2*)
assumes *m1*: *m1* = *size-of TYPE*(*'a*)
assumes *n1*: *n1* = *size-td t1*
assumes *m2*: *m2* = *size-of TYPE*(*'b*)

assumes *n2*: *n2* = *size-td t2*
shows {&(*q*→*f1*)..+*n1*} ∩ {&(*p*→*f2*)..+*n2*} = {}
⟨*proof*⟩

lemmas *disjoint-field-lvalue-propagation* =
disjoint-field-lvalue-propagation-left
disjoint-field-lvalue-propagation-right
disjoint-field-lvalue-propagation-both

lemma *disjoint-field-lvalue-neq*:
fixes *q*::'*a*::*mem-type ptr*
fixes *p*::'*b*::*mem-type ptr*
assumes *disj*: {*ptr-val q*..+*m1*} ∩ {*ptr-val p*..+*m2*} = {}
assumes *fl1*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f1 0* = *Some* (*t1*, *k1*)
assumes *fl2*: *field-lookup* (*typ-info-t TYPE*('*b*)) *f2 0* = *Some* (*t2*, *k2*)
assumes *m1*: *m1* = *size-of TYPE*('*a*)
assumes *n1*: *n1* = *size-td t1*
assumes *m2*: *m2* = *size-of TYPE*('*b*)
assumes *n2*: *n2* = *size-td t2*
shows &(*q*→*f1*) = &(*p*→*f2*) = *False*
⟨*proof*⟩

lemma *overlap-field-disj-contradiction*:
fixes *q*::'*a*::*mem-type ptr*
assumes *overlap*: *A* ∩ {&(*q*→*f*)..+*n*} ≠ {}
assumes *disj*: *A* ∩ {*ptr-val q*..+*m*} = {}
assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t*, *k*)
assumes *m*: *m* = *size-of TYPE*('*a*)
assumes *n*: *n* = *size-td t*
shows *False*
⟨*proof*⟩

thm *root-ptr-valid-non-prefix-disj*

lemma *overlap-field-prefix-left*:
fixes *p*::'*a*::*mem-type ptr*
assumes *overlap*: {&(*p*→*f*) ..+*n1*} ∩ {&(*p*→*g*) ..+*n2*} ≠ {}
assumes *less*: *length f* < *length g*
assumes *f*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t1*, *k1*)
assumes *g*: *field-lookup* (*typ-info-t TYPE*('*a*)) *g 0* = *Some* (*t2*, *k2*)
assumes *n1*: *n1* = *size-td t1*
assumes *n2*: *n2* = *size-td t2*
shows *prefix f g*
⟨*proof*⟩

lemma *overlap-field-prefix-right*:
fixes *p*::'*a*::*mem-type ptr*
assumes *overlap*: {&(*p*→*g*) ..+*n2*} ∩ {&(*p*→*f*) ..+*n1*} ≠ {}
assumes *less*: *length f* < *length g*

assumes *f*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t1*, *k1*)
assumes *g*: *field-lookup* (*typ-info-t TYPE*('*a*)) *g 0* = *Some* (*t2*, *k2*)
assumes *n1*: *n1* = *size-td t1*
assumes *n2*: *n2* = *size-td t2*
shows *prefix f g*
⟨*proof*⟩

lemma *field-lvalue-eq-non-prefix*:
fixes *p*::'*a*::*mem-type ptr*
assumes *fld-eq*: &(*p*→*f*) = &(*p*→*g*)
assumes *f*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t1*, *k1*)
assumes *g*: *field-lookup* (*typ-info-t TYPE*('*a*)) *g 0* = *Some* (*t2*, *k2*)
assumes *f-g*: ¬ *prefix f g*
assumes *g-f*: ¬ *prefix g f*
shows *f* = *g*
⟨*proof*⟩

lemma *field-lvalue-eq-non-prefix-conv*:
fixes *p*::'*a*::*mem-type ptr*
assumes *f*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*t1*, *k1*)
assumes *g*: *field-lookup* (*typ-info-t TYPE*('*a*)) *g 0* = *Some* (*t2*, *k2*)
assumes *f-g*: ¬ *prefix f g*
assumes *g-f*: ¬ *prefix g f*
shows &(*p*→*f*) = &(*p*→*g*) ⟷ *f* = *g*
⟨*proof*⟩

thm *cvalid-field-pres''*

lemma *hrs-mem-update-comp*: *hrs-mem-update f o hrs-mem-update g* = *hrs-mem-update* (*f o g*)
⟨*proof*⟩

lemma *hrs-mem-update-comp'*: *hrs-mem-update f* (*hrs-mem-update g s*) = *hrs-mem-update* (*f o g*) *s*
⟨*proof*⟩

named-theorems *plift-defs* **and**
plift-cond-simps **and**
plift-iff **and**
plift-eqI **and**
ptr-valid-h-t-valid **and**
plift-heap-update **and**
ptr-valid-intros **and**
h-val-field-plift **and**
the-plift-h-val-conv

lemma *map-option-the-conv*: *f* (*the* (*map-option g x*)) = *f* (*the x*) ⟷ (∀ *v*. *x* = *Some v* ⟶ *f* (*g v*) = *f v*)

⟨*proof*⟩

definition *the-default*:: *'a* ⇒ *'a option* ⇒ *'a* **where**
the-default x v = (*case v of Some z* ⇒ *z* | *None* ⇒ *x*)

lemma *the-default-simps*[*simp*]:
the-default x None = *x*
the-default x (*Some z*) = *z*
⟨*proof*⟩

locale *wf-ptr-valid* =
fixes *ptr-valid*::*heap-typ-desc* ⇒ *'a*::*mem-type ptr-guard*
assumes *ptr-valid-h-t-valid*[*ptr-valid-h-t-valid*]: ⋀*d*. *ptr-valid d p* ⟹ *d* ⊨$_t$ *p*
begin

definition *plift*:: *heap-raw-state* ⇒ *'a typ-heap*
where [*plift-defs*]:*plift h* = *lift-t* (*ptr-valid* (*hrs-htd h*)) *h*

lemma *plift-Some-iff* [*plift-iff*]:
shows (∃ *v*. *plift h p* = *Some v*) ⟷ *ptr-valid* (*hrs-htd h*) *p*
⟨*proof*⟩

lemma *plift-None-iff* [*plift-iff*]:
shows (*plift h p* = *None*) ⟷ ¬ *ptr-valid* (*hrs-htd h*) *p*
⟨*proof*⟩

lemma *plift-None*: ¬ *ptr-valid* (*hrs-htd h*) *p* ⟹ *plift h p* = *None*
⟨*proof*⟩

lemma *ptr-valid-c-guard*: *ptr-valid d p* ⟹ *c-guard p*
⟨*proof*⟩

lemma *plift-Some-ptr-valid*[*plift-cond-simps*]: *plift h p* = *Some v* ⟹ *ptr-valid* (*hrs-htd h*) *p*
⟨*proof*⟩

lemma *plift-Some-h-t-valid*[*plift-cond-simps*]: *plift h p* = *Some v* ⟹ *hrs-htd h* ⊨$_t$ *p*
⟨*proof*⟩

lemma *plift-Some-c-guard*[*plift-cond-simps*]: *plift h p* = *Some v* ⟹ *c-guard p*
⟨*proof*⟩

lemma *ptr-valid-heap-update-conv*[*simp*]:
ptr-valid (*hrs-htd* ((*hrs-mem-update* (*heap-update p v*) *s*))) *q* ⟷ *ptr-valid* (*hrs-htd s*) *q*
⟨*proof*⟩

lemma *ptr-valid-heap-update-padding-conv*[*simp*]:
ptr-valid (*hrs-htd* ((*hrs-mem-update* (*heap-update-padding p v bs*) *s*))) *q* ⟷ *ptr-valid* (*hrs-htd s*) *q*
⟨*proof*⟩

lemma *ptr-valid-heap-update-pres*: *ptr-valid* (*hrs-htd s*) *q* ⟹
ptr-valid (*hrs-htd* ((*hrs-mem-update* (*heap-update p v*)*s*))) *q*
⟨*proof*⟩

lemma *ptr-valid-heap-update-padding-pres*: *ptr-valid* (*hrs-htd s*) *q* ⟹
ptr-valid (*hrs-htd* ((*hrs-mem-update* (*heap-update-padding p v bs*)*s*))) *q*
⟨*proof*⟩

lemma *plift-Some-h-val*:
assumes *Some*: *plift h p* = *Some v*
shows *h-val* (*hrs-mem h*) *p* = *v*
⟨*proof*⟩

lemma *ptr-valid-plift-Some-hval*:
assumes *ptr-valid* (*hrs-htd h*) *p*
shows *plift h p* = *Some* (*h-val* (*hrs-mem h*) *p*)
⟨*proof*⟩

lemma *the-plift-hval-conv*: *ptr-valid* (*hrs-htd h*) *p* ⟹ *the* (*plift h p*) = *h-val* (*hrs-mem h*) *p*
⟨*proof*⟩

lemma *the-default-plift-hval-conv*[*the-plift-h-val-conv*]:
ptr-valid (*hrs-htd h*) *p* ⟹ *the-default c-type-class.zero* (*plift h p*) = *h-val* (*hrs-mem h*) *p*
⟨*proof*⟩

lemma *valid-h-val-the-plift-conv*:
assumes *valid*: *ptr-valid* (*hrs-htd h*) *p*
shows *h-val* (*hrs-mem h*) *p* = *the* (*plift h p*)
⟨*proof*⟩

lemma *valid-h-val-the-default-plift-conv*:
assumes *valid*: *ptr-valid* (*hrs-htd h*) *p*
shows *h-val* (*hrs-mem h*) *p* = *the-default c-type-class.zero* (*plift h p*)
⟨*proof*⟩

lemma *plift-Some-clift-Some*:
assumes *Some*: *plift h p* = *Some v*
shows *clift h p* = *Some v*
⟨*proof*⟩

lemma *clift-None-plift-None*:
assumes *clift h p = None*
shows *plift h p = None*
⟨*proof*⟩

lemma *plift-clift-conv*:
assumes *ptr-valid (hrs-htd h) p*
shows *plift h p = clift h p*
⟨*proof*⟩

lemma *the-plift-hval-fun-upd-eqI* [*plift-eqI*]:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*::*'a*::*mem-type ptr*
assumes *ptr-valid (hrs-htd h) p*
shows (*the-default c-type-class.zero* (*plift*
(*hrs-mem-update* (*heap-update p v*) *h*) *q*)) =
((λ*p*. *the-default c-type-class.zero* (*plift h p*))(*p* := *v*)) *q*
⟨*proof*⟩

lemma *the-plift-hval-fun-upd-padding-eqI* [*plift-eqI*]:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*::*'a*::*mem-type ptr*
assumes *ptr-valid (hrs-htd h) p*
assumes *length bs = size-of TYPE*('*a*)
shows (*the-default c-type-class.zero* (*plift*
(*hrs-mem-update* (*heap-update-padding p v bs*) *h*) *q*)) =
((λ*p*. *the-default c-type-class.zero* (*plift h p*))(*p* := *v*)) *q*
⟨*proof*⟩

lemma *field-the-plift-hval-fun-upd-eqI* [*plift-eqI*]:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*::*'a*::*mem-type ptr*
assumes *ptr-valid (hrs-htd h) p* ∧ *f x = v*
shows *f* (*the-default c-type-class.zero* (*plift*
(*hrs-mem-update* (*heap-update p x*) *h*) *q*)) =
((λ*p*. *f* (*the-default c-type-class.zero* (*plift h p*)))(*p* := *v*)) *q*
⟨*proof*⟩

lemma *field-the-plift-hval-fun-upd-padding-eqI* [*plift-eqI*]:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*::*'a*::*mem-type ptr*
assumes *ptr-valid (hrs-htd h) p* ∧ *f x = v*
assumes *length bs = size-of TYPE*('*a*)
shows *f* (*the-default c-type-class.zero* (*plift*
(*hrs-mem-update* (*heap-update-padding p x bs*) *h*) *q*)) =
((λ*p*. *f* (*the-default c-type-class.zero* (*plift h p*)))(*p* := *v*)) *q*
⟨*proof*⟩

lemma *the-plift-hval-eqI* [*plift-eqI*]:
fixes *p*::*'b*::*c-type ptr*
fixes *q*::*'a*::*mem-type ptr*
assumes *eq*: *ptr-valid* (*hrs-htd h*) *q* ⟹ *hrs-htd h* ⊨$_t$ *q* ⟹
(*h-val* ((*heap-update p v*) (*hrs-mem h*)) *q*) = (*h-val* (*hrs-mem h*) *q*)
shows (*the-default c-type-class.zero* (*plift*
(*hrs-mem-update* (*heap-update p v*) *h*) *q*)) =
(*the-default c-type-class.zero* (*plift h q*))
⟨*proof*⟩

lemma *the-plift-hval-padding-eqI* [*plift-eqI*]:
fixes *p*::*'b*::*c-type ptr*
fixes *q*::*'a*::*mem-type ptr*
assumes *lbs*: *length bs* = *size-of TYPE*(*'b*)
assumes *eq*: *ptr-valid* (*hrs-htd h*) *q* ⟹ *hrs-htd h* ⊨$_t$ *q* ⟹
(*h-val* ((*heap-update-padding p v bs*) (*hrs-mem h*)) *q*) = (*h-val* (*hrs-mem h*) *q*)

shows (*the-default c-type-class.zero* (*plift*
(*hrs-mem-update* (*heap-update-padding p v bs*) *h*) *q*)) =
(*the-default c-type-class.zero* (*plift h q*))
⟨*proof*⟩

lemma *field-the-plift-hval-eqI* [*plift-eqI*]:
fixes *p*::*'b*::*c-type ptr*
fixes *q*::*'a*::*mem-type ptr*
assumes *eq*: *ptr-valid* (*hrs-htd h*) *q* ⟹ *hrs-htd h* ⊨$_t$ *q* ⟹
f (*h-val* ((*heap-update p v*) (*hrs-mem h*)) *q*) = *f* (*h-val* (*hrs-mem h*) *q*)
shows *f* (*the-default c-type-class.zero* (*plift*
(*hrs-mem-update* (*heap-update p v*) *h*) *q*)) =
f (*the-default c-type-class.zero* (*plift h q*))
⟨*proof*⟩

lemma *field-the-plift-hval-padding-eqI* [*plift-eqI*]:
fixes *p*::*'b*::*c-type ptr*
fixes *q*::*'a*::*mem-type ptr*
assumes *lbs*: *length bs* = *size-of TYPE*(*'b*)
assumes *eq*: *ptr-valid* (*hrs-htd h*) *q* ⟹ *hrs-htd h* ⊨$_t$ *q* ⟹
f (*h-val* ((*heap-update-padding p v bs*) (*hrs-mem h*)) *q*) = *f* (*h-val* (*hrs-mem h*)
q)
shows *f* (*the-default c-type-class.zero* (*plift*
(*hrs-mem-update* (*heap-update-padding p v bs*) *h*) *q*)) =
f (*the-default c-type-class.zero* (*plift h q*))
⟨*proof*⟩

lemma *plift-heap-update* [*plift-heap-update*]:
⟦ *ptr-valid* (*hrs-htd h*) *p* ⟧ ⟹
plift (*hrs-mem-update* (*heap-update p v*) *h*)
= (*plift h*)(*p* := *Some* (*v*::*'a*::*mem-type*))

⟨*proof*⟩

lemma *plift-heap-update-padding* [*plift-heap-update*]:
⟦ *ptr-valid* (*hrs-htd h*) *p*; *length bs* = *size-of TYPE*('*a*) ⟧ ⟹
plift (*hrs-mem-update* (*heap-update-padding p v bs*) *h*)
= (*plift h*)(*p* := *Some* (*v*::'*a*::*mem-type*))
⟨*proof*⟩

lemma *h-val-field-plift* [*h-val-field-plift*]:
fixes *pa* :: '*a* :: *mem-type ptr*
assumes *cl*: *plift hp pa* = *Some v*
and *fl*: *field-ti TYPE*('*a*) *f* = *Some t*
and *eu*: *export-uinfo t* = *typ-uinfo-t TYPE*('*b* :: *mem-type*)
shows *h-val* (*hrs-mem hp*) (*Ptr* &(*pa*→*f*) :: '*b* :: *mem-type ptr*) = *from-bytes* (*access-ti*$_0$ *t v*)
⟨*proof*⟩

lemma *plift-disjoint-region*:
fixes *p*:: '*a* :: *mem-type ptr*
fixes *q*:: '*b* :: *mem-type ptr*
assumes *disj*: *ptr-span q* ∩ *ptr-span p* = {}
shows *plift* (*hrs-mem-update* (*heap-update q v*) *m*) *p* = *plift m p*
⟨*proof*⟩

lemma *plift-disjoint-region-padding*:
fixes *p*:: '*a* :: *mem-type ptr*
fixes *q*:: '*b* :: *mem-type ptr*
assumes *disj*: *ptr-span q* ∩ *ptr-span p* = {}
assumes *lbs*: *length bs* = *size-of TYPE*('*b*)
shows *plift* (*hrs-mem-update* (*heap-update-padding q v bs*) *m*) *p* = *plift m p*
⟨*proof*⟩

lemma *map-option-the-plift-h-val-conv*:
f (*the* (*map-option g* (*plift h p*))) = *f* (*the* (*plift h p*)) ⟷ (*ptr-valid* (*hrs-htd h*) *p* ⟶ *f* (*g* (*h-val* (*hrs-mem h*) *p*)) = *f* (*h-val* (*hrs-mem h*) *p*))
⟨*proof*⟩

lemma *invalid-plift-heap-update-skip*:
assumes *invalid-p*: ¬ *ptr-valid* (*hrs-htd h*) *p*
assumes *typed-p*: *hrs-htd h* ⊨$_t$ *p*
shows *plift* (*hrs-mem-update* (*heap-update p v*) *h*) *q* = *plift h q*
⟨*proof*⟩

lemma *invalid-plift-heap-update-padding-skip*:
assumes *invalid-p*: ¬ *ptr-valid* (*hrs-htd h*) *p*
assumes *typed-p*: *hrs-htd h* ⊨$_t$ *p*
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
shows *plift* (*hrs-mem-update* (*heap-update-padding p v bs*) *h*) *q* = *plift h q*

⟨*proof*⟩

end

global-interpretation *simple-lift*: *wf-ptr-valid root-ptr-valid* **rewrites** *simple-lift.plift* = *simple-lift*
⟨*proof*⟩

named-theorems *ptr-valid-stack-alloc* **and** *ptr-valid-stack-release* **and** *plift-stack-alloc* **and** *plift-stack-release*

locale *ptr-valid-stack-alloc* = *wf-ptr-valid* +
assumes *alloc*[*ptr-valid-stack-alloc*]:
⋀*d d′ p q*. (*p*, *d′*) ∈ *stack-alloc* $\mathcal{S}$ *TYPE*('*a*::*mem-type*) *d* ⟹
ptr-valid d′ q ⟷ (*q* = *p*) ∨ *ptr-valid d q*
begin

lemma *plift-stack-alloc-update*:
assumes *alloc*: (*p*, *d*) ∈ *stack-alloc* $\mathcal{S}$ *TYPE*('*a*::*mem-type*) (*hrs-htd m*)
shows (*plift* (*hrs-htd-update* (λ-. *d*) *m*))(*p*↦*x*) = (λ*q*. *if p* = *q then Some x else plift m q*)
⟨*proof*⟩

lemma *plift-stack-alloc-sim*[*plift-stack-alloc*]:
assumes *alloc*: (*p*, *d*) ∈ *stack-alloc* $\mathcal{S}$ *TYPE*('*a*::*mem-type*) (*hrs-htd m*)
shows (*plift* (*hrs-mem-update* (*heap-update p x*) (*hrs-htd-update* (λ-. *d*) *m*))) =
(λ*q*. *if p* = *q then Some x else plift m q*)
⟨*proof*⟩

end

locale *ptr-valid-stack-release* = *wf-ptr-valid* +
assumes *release*[*ptr-valid-stack-release*]:
⋀*d p q*. *root-ptr-valid d p* ⟹ *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*(*stack-byte*) ⟹
ptr-valid (*stack-release p d*) *q* ⟷ ((*q* ≠ *p*) ∧ *ptr-valid d q*)
begin

lemma *plift-stack-release*[*plift-stack-release*]:
assumes *root-valid*: *root-ptr-valid* (*hrs-htd m*) (*p*::'*a*::*mem-type ptr*)
assumes *p*: *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
shows *plift* (*hrs-htd-update* (*stack-release p*) *m*) *q* =
(*if p* = *q then None else plift m q*)
⟨*proof*⟩
end

locale *ptr-valid-stack* = *ptr-valid-stack-alloc* + *ptr-valid-stack-release*

global-interpretation *simple-lift*: *ptr-valid-stack root-ptr-valid*
⟨*proof*⟩

lemma *valid-root-footprint-domain*:
assumes *valid-root-footprint d a t*
assumes $\bigwedge x.\ x \in \{a\ ..+\ \textit{size-td}\ t\} \Longrightarrow d'\ x = d\ x$
shows *valid-root-footprint d′ a t*
⟨*proof*⟩

lemma *valid-root-footprint-cases*:
assumes *valid-a*: *valid-root-footprint d a T*
assumes *root-p*: *valid-root-footprint d b S*
shows $(a = b \wedge S = T) \vee \{b\ ..+\ \textit{size-td}\ S\} \cap \{a\ ..+\ \textit{size-td}\ T\} = \{\}$
⟨*proof*⟩

lemma *valid-root-footprint-domain-cases*:
assumes *valid-a*: *valid-root-footprint d′ a t*
assumes *other-eq*: $\bigwedge x.\ x \notin \{a\ ..+\ \textit{size-td}\ t\} \Longrightarrow d'\ x = d\ x$
assumes *root-p*: *valid-root-footprint d′ b s*
shows
$(a = b \wedge s = t \vee$
valid-root-footprint d b s)
⟨*proof*⟩

lemma *h-t-valid-ptr-span-preservation*:
fixes *p*::′*a*::*mem-type ptr*
assumes *valid*: $d \models_t p$
assumes *eq*: $\bigwedge a.\ a \in \textit{ptr-span}\ p \Longrightarrow d'\ a = d\ a$
shows $d' \models_t p$
⟨*proof*⟩

lemma
fixes *p*::′*a*::*mem-type ptr*
fixes *q*::′*a*::*mem-type ptr*
assumes *root-ptr-valid d p*
assumes $d \models_t q$
shows $p = q \vee \textit{ptr-span}\ p \cap \textit{ptr-span}\ q = \{\}$
⟨*proof*⟩

lemma *stack-alloc-typing-other*:
fixes *q*::′*b*::*mem-type ptr*
assumes *stack-alloc*: $(p,\ d') \in \textit{stack-alloc}\ \mathcal{S}\ (\textit{TYPE}('a::\textit{mem-type}))\ d$
assumes *no-stack*: $\textit{typ-uinfo-t}\ (\textit{TYPE}('b)) \neq \textit{typ-uinfo-t}\ (\textit{TYPE}(\textit{stack-byte}))$
assumes *other*: $\neg\ \textit{TYPE}('b) \leq_\tau (\textit{TYPE}('a))$
shows $d' \models_t q = d \models_t q$
⟨*proof*⟩

lemma $TYPE('b::c\text{-}type) \leq_\tau (TYPE('a::c\text{-}type)) \Longrightarrow typ\text{-}uinfo\text{-}t\ TYPE('b) \neq typ\text{-}uinfo\text{-}t\ TYPE('a)$
$\Longrightarrow \neg TYPE('a) \leq_\tau (TYPE('b))$

⟨*proof*⟩

named-theorems *stack-alloc-ptr-valid-parent* **and** *stack-release-ptr-valid-parent*

lemma *stack-alloc-root-ptr-valid-other*:
fixes $q::'b::mem\text{-}type\ ptr$
assumes *stack-alloc*: $(p, d') \in stack\text{-}alloc\ \mathcal{S}\ (TYPE('a::mem\text{-}type))\ d$
assumes *neq*: $typ\text{-}uinfo\text{-}t\ TYPE('b) \neq typ\text{-}uinfo\text{-}t\ TYPE('a)$
assumes *sub*: $TYPE('a) \leq_\tau (TYPE('b))$
shows $root\text{-}ptr\text{-}valid\ d'\ (q::'b::mem\text{-}type\ ptr) = root\text{-}ptr\text{-}valid\ d\ q$
⟨*proof*⟩

lemma *stack-release-root-ptr-valid-other*:
fixes $p::'a::mem\text{-}type\ ptr$
fixes $q::'b::mem\text{-}type\ ptr$
assumes *root*: $root\text{-}ptr\text{-}valid\ d\ p$
assumes *non-stack*: $typ\text{-}uinfo\text{-}t\ (TYPE('a)) \neq typ\text{-}uinfo\text{-}t\ (TYPE(stack\text{-}byte))$
assumes *neq*: $typ\text{-}uinfo\text{-}t\ TYPE('b::mem\text{-}type) \neq typ\text{-}uinfo\text{-}t\ TYPE('a)$
assumes *sub*: $TYPE('a) \leq_\tau (TYPE('b))$
shows $root\text{-}ptr\text{-}valid\ (stack\text{-}release\ p\ d)\ (q::'b::mem\text{-}type\ ptr) = root\text{-}ptr\text{-}valid\ d\ q$
⟨*proof*⟩

lemma *bex-impl-forall-conv*: $((\exists\, x{\in}A.\ P\ x) \longrightarrow Q) \longleftrightarrow (\forall\, x.\ x \in A \longrightarrow P\ x \longrightarrow Q)$
⟨*proof*⟩

end

20.3 More Stack Typing

theory *Stack-Typing*
imports *L2Defs Runs-To-VCG*
begin

definition *equal-upto*:: $'a\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool$ **where**
$equal\text{-}upto\ S\ f\ g = (\forall\, x.\ x \notin S \longrightarrow f\ x = g\ x)$

lemma *equal-upto-refl*[*simp*, *intro*]: $equal\text{-}upto\ S\ f\ f$
⟨*proof*⟩

lemma *symp-equal-upto*: $symp\ (equal\text{-}upto\ S)$
⟨*proof*⟩

lemma *equal-upto-commute*: *equal-upto S f g* $\longleftrightarrow$ *equal-upto S g f*
⟨*proof*⟩

lemma *equal-upto-trans*[*trans*]: *equal-upto S f g* $\Longrightarrow$ *equal-upto S g h* $\Longrightarrow$ *equal-upto S f h*
⟨*proof*⟩

lemma *equal-upto-mono*: *S* $\subseteq$ *T* $\Longrightarrow$ *equal-upto S f g* $\Longrightarrow$ *equal-upto T f g*
⟨*proof*⟩

definition *equal-on*:: *'a set* $\Rightarrow$ (*'a* $\Rightarrow$ *'b*) $\Rightarrow$ (*'a* $\Rightarrow$ *'b*) $\Rightarrow$ *bool* **where**
equal-on S f g = ($\forall$ *x*. *x* $\in$ *S* $\longrightarrow$ *f x* = *g x*)

lemma *equal-on-UNIV-iff*[*simp*]: *equal-on UNIV f g* $\longleftrightarrow$ *f* = *g*
⟨*proof*⟩

lemma *equal-on-refl* [*simp*, *intro*]: *equal-on S f f*
⟨*proof*⟩

lemma *symp-equal-on*: *symp* (*equal-on S*)
⟨*proof*⟩

lemma *equal-on-commute*: *equal-on S f g* $\longleftrightarrow$ *equal-on S g f*
⟨*proof*⟩

lemma *equal-on-trans*[*trans*]: *equal-on S f g* $\Longrightarrow$ *equal-on S g h* $\Longrightarrow$ *equal-on S f h*
⟨*proof*⟩

lemma *equal-on-mono*: *S* $\subseteq$ *T* $\Longrightarrow$ *equal-on T f g* $\Longrightarrow$ *equal-on S f g*
⟨*proof*⟩

lemma *equal-on-equal-upto-eq*: *equal-on S f g* $\Longrightarrow$ *equal-upto S f g* $\Longrightarrow$ *f* = *g*
⟨*proof*⟩

named-theorems *unchanged-typing-on-simps* **and** *unchanged-typing*

declare *mex-def* [*unchanged-typing-on-simps*]
declare *meq-def* [*unchanged-typing-on-simps*]

⟨*ML*⟩

lemma *stack-allocs-releases-equal-on-stack*:
(*p*, *d2*) $\in$ *stack-allocs n* $\mathcal{S}$ *TYPE*(*'a*::*mem-type*) *d1* $\Longrightarrow$
equal-on $\mathcal{S}$ *d2 d3* $\Longrightarrow$ *equal-on* $\mathcal{S}$ *d1* (*stack-releases n p d3*)
⟨*proof*⟩

lemma *stack-allocs-releases-equal-on-typing*:

$(p,\ d2) \in \mathit{stack\text{-}allocs}\ n\ \mathcal{S}\ \mathit{TYPE}('a{::}\mathit{mem\text{-}type})\ d1 \Longrightarrow$
$\mathit{equal\text{-}on}\ X\ d2\ d3 \Longrightarrow \mathit{equal\text{-}on}\ X\ d1\ (\mathit{stack\text{-}releases}\ n\ p\ d3)$
⟨*proof*⟩

lemma *stack-allocs-releases-equal-on-typing'*:
$(p,\ d2) \in \mathit{stack\text{-}allocs}\ n\ \mathcal{S}\ \mathit{TYPE}('a{::}\mathit{mem\text{-}type})\ d1 \Longrightarrow$
$\mathit{equal\text{-}on}\ X'\ d2\ d3 \Longrightarrow\ X \subseteq X' \Longrightarrow \mathit{equal\text{-}on}\ X\ d1\ (\mathit{stack\text{-}releases}\ n\ p\ d3)$
⟨*proof*⟩

context *heap-typing-state*
begin

definition *unchanged-typing-on*:: $\mathit{addr\ set} \Rightarrow 's \Rightarrow 's \Rightarrow \mathit{bool}$ **where**
$\mathit{unchanged\text{-}typing\text{-}on}\ S\ s\ t = \mathit{equal\text{-}on}\ S\ (\mathit{htd}\ s)\ (\mathit{htd}\ t)$

lemma *unchanged-typing-on-UNIV-iff*: $\mathit{unchanged\text{-}typing\text{-}on}\ \mathit{UNIV}\ s\ t \longleftrightarrow \mathit{htd}\ s = \mathit{htd}\ t$
⟨*proof*⟩

lemma *unchanged-typing-on-UNIV-I*: $\mathit{htd}\ s = \mathit{htd}\ t \Longrightarrow \mathit{unchanged\text{-}typing\text{-}on}\ \mathit{UNIV}\ s\ t$
⟨*proof*⟩

lemma *typing-eq-unchanged-typing-on*: $\mathit{htd}\ s = \mathit{htd}\ t \Longrightarrow \mathit{unchanged\text{-}typing\text{-}on}\ S\ s\ t$
⟨*proof*⟩

lemma *unchanged-typing-on-relf*[*simp*]: $\mathit{unchanged\text{-}typing\text{-}on}\ S\ s\ s$
⟨*proof*⟩

lemma *unchanged-typing-on-conv*[*unchanged-typing-on-simps*]: $\mathit{unchanged\text{-}typing\text{-}on}\ S\ s\ t \longleftrightarrow (\forall a \in S.\ \mathit{htd}\ s\ a = \mathit{htd}\ t\ a)$
⟨*proof*⟩

lemma *unchanged-typing-on-cong*: $(\bigwedge a.\ a \in S \Longrightarrow \mathit{htd}\ s\ a = \mathit{htd}\ s'\ a) \Longrightarrow$
$(\bigwedge a.\ a \in S \Longrightarrow \mathit{htd}\ t\ a = \mathit{htd}\ t'\ a)$
$\Longrightarrow \mathit{unchanged\text{-}typing\text{-}on}\ S\ s\ t \longleftrightarrow \mathit{unchanged\text{-}typing\text{-}on}\ S\ s'\ t$
⟨*proof*⟩

lemma *typing-eq-left-unchanged-typing-on*: $\mathit{htd}\ s1 = \mathit{htd}\ s2 \Longrightarrow$
$\mathit{unchanged\text{-}typing\text{-}on}\ S\ s1\ t \longleftrightarrow \mathit{unchanged\text{-}typing\text{-}on}\ S\ s2\ t$
⟨*proof*⟩

lemma *symp-unchanged-typing-on*: $\mathit{symp}\ (\mathit{unchanged\text{-}typing\text{-}on}\ S)$
⟨*proof*⟩

lemma *unchanged-typing-on-commute*: $\mathit{unchanged\text{-}typing\text{-}on}\ S\ s\ t \longleftrightarrow \mathit{unchanged\text{-}typing\text{-}on}\ S\ t\ s$

⟨*proof*⟩

lemma *unchanged-typing-on-trans*[*trans*]:
assumes *s-t*: *unchanged-typing-on S s t*
assumes *t-u*: *unchanged-typing-on S t u*
shows *unchanged-typing-on S s u*
⟨*proof*⟩

lemma *unchanged-typing-on-mono*: $S \subseteq T \Longrightarrow$ *unchanged-typing-on T s t* $\Longrightarrow$ *unchanged-typing-on S s t*
⟨*proof*⟩

lemma *unchanged-typing-on-root-ptr-valid-preservation*:
fixes *p*::*'a*::*mem-type ptr*
assumes *unchanged-typing-on S s t*
assumes *ptr-span p* $\subseteq$ *S*
assumes *root-ptr-valid* (*htd s*) *p*
shows *root-ptr-valid* (*htd t*) *p*
⟨*proof*⟩

⟨*ML*⟩
declare [[*simproc del*: *unchanged-typing-nondet*]]
end

context *stack-heap-state*
begin
lemma *with-fresh-stack-ptr-unchanged-typing*[*unchanged-typing*]:
assumes *f*[*runs-to-vcg*]:$\bigwedge p$ *s.* (*f p*) · *s ?*{|λr *t. typing.unchanged-typing-on* $\mathcal{S}$ *s t*|}
shows (*with-fresh-stack-ptr n I* (*L2-VARS f nm*)) · *s ?*{|λr *t. typing.unchanged-typing-on* $\mathcal{S}$ *s t*|}
⟨*proof*⟩
end

lemma *override-on-merge*: *override-on* (*override-on f g B*) *g A* = *override-on f g* (*A* $\cup$ *B*)
⟨*proof*⟩

lemma *override-on-id* [*simp*]: *override-on f f A* = *f*
⟨*proof*⟩

lemma *override-cancel-snd* [*simp*]: *override-on f* (*override-on g1 g0 A*) *A* = *override-on f g0 A*
⟨*proof*⟩

lemma *override-cancel-subset-snd* : *A* $\subseteq$ *B* $\Longrightarrow$ *override-on f* (*override-on g1 g0 B*) *A* = *override-on f g0 A*
⟨*proof*⟩

lemma *override-cancel-fst*[*simp*] : *override-on* (*override-on f1 f0 A*) *g A* = *override-on f1 g A*
⟨*proof*⟩

lemma *override-cancel-subset-fst* : $A \subseteq B \Longrightarrow$ *override-on* (*override-on f1 f0 A*) *g B* = *override-on f1 g B*
⟨*proof*⟩

lemma *equal-on-stack-free-preservation*:
assumes *eq*: *equal-on S d2 d1*
assumes *stack-free d1* ⊆ *S*
assumes *stack-free d2* ⊆ *S*
shows *stack-free d2* = *stack-free d1*
⟨*proof*⟩

lemma *equal-upto-disjoint-h-val*: *equal-upto P h′ h* ⟹
ptr-span p ∩ *P* = {} ⟹
h-val h′ (*p*::′*a*::*mem-type ptr*) = *h-val h p*
⟨*proof*⟩

context *heap-state*
begin

definition *equal-upto-heap-on*:: *addr set* ⇒ ′*s* ⇒ ′*s* ⇒ *bool* **where**
equal-upto-heap-on S s t = (∃ *T H*.
t = *hmem-upd* (λ-. *H*) (*htd-upd* (λ-. *T*) *s*) ∧
equal-upto S (*hmem s*) *H* ∧
equal-upto S (*htd s*) *T*)

definition *zero-heap-on*:: *addr set* ⇒ ′*s* ⇒ ′*s* **where**
zero-heap-on A s = *hmem-upd* (λ*h*. *override-on h zero-heap A*) (*htd-upd* (λ*htd*. *override-on htd stack-byte-typing A*) *s*)

lemma *equal-upto-heap-on-equal-upto-htd*: *equal-upto-heap-on S s t* ⟹ *equal-upto S* (*htd s*) (*htd t*)
⟨*proof*⟩

lemma *equal-upto-heap-on-equal-upto-hmem*: *equal-upto-heap-on S s t* ⟹ *equal-upto S* (*hmem s*) (*hmem t*)
⟨*proof*⟩

lemma *zero-heap-on-empty*[*simp*]: *zero-heap-on* {} *s* = *s*
⟨*proof*⟩

lemma *equal-upto-heap-on-zero-heap-on-subset*: *A* ⊆ *B* ⟹ *equal-upto-heap-on B s* (*zero-heap-on A s*)
⟨*proof*⟩

lemma *equal-upto-heap-on-zero-heap-on*[*simp*]: *equal-upto-heap-on A s* (*zero-heap-on A s*)
⟨*proof*⟩

lemma *equal-upto-heap-on-refl*[*simp*, *intro*]: *equal-upto-heap-on S s s*
⟨*proof*⟩

lemma *override-on-heap-update-other*:
fixes *p*::*'a*::*mem-type ptr*
shows *ptr-span p* ⊆ *A* ⟹ *override-on* (*heap-update p v h*) *f A* = *override-on h f A*
⟨*proof*⟩

lemma *override-on-heap-update-commute*:
fixes *p*::*'a*::*mem-type ptr*
shows *ptr-span p* ∩ *A* = {} ⟹
override-on (*heap-update p v h*) *f A* = *heap-update p v* (*override-on h f A*)
⟨*proof*⟩

lemma *override-on-heap-update-padding-other*:
fixes *p*::*'a*::*mem-type ptr*
shows *ptr-span p* ⊆ *A* ⟹ *length bs* = *size-of TYPE*(*'a*) ⟹ *override-on* (*heap-update-padding p v bs h*) *f A* = *override-on h f A*
⟨*proof*⟩

lemma *zero-heap-on-heap-update-other*:
fixes *p*::*'a*::*mem-type ptr*
shows *ptr-span p* ⊆ *A* ⟹ *zero-heap-on A* (*hmem-upd* (*heap-update p v*) *s*) = *zero-heap-on A s*
⟨*proof*⟩

lemma *override-on-stack-byte-typing-stack-allocs*:
assumes *alloc*: (*p*, *d*) ∈ *stack-allocs n* 𝒮 *TYPE*(*'a*::*mem-type*) *d0*
assumes *subset*: *stack-free d0* ⊆ *A*
shows *override-on d stack-byte-typing A* = *override-on d0 stack-byte-typing A*
⟨*proof*⟩

lemma *zero-heap-on-stack-allocs*:
assumes *alloc*: (*p*, *d*) ∈ *stack-allocs n* 𝒮 *TYPE*(*'a*::*mem-type*) (*htd s*)
assumes *subset*: *stack-free* (*htd s*) ⊆ *A*
shows *zero-heap-on A* (*htd-upd* (λ-. *d*) *s*) = *zero-heap-on A s*
⟨*proof*⟩

lemma *zero-heap-on-stack-releases*:

fixes p:: $'a$::*mem-type ptr*
assumes *subset*: $\{ptr\text{-}val\ p..+n * size\text{-}of\ TYPE('a::mem\text{-}type)\} \subseteq A$
shows *zero-heap-on* A (*htd-upd* (*stack-releases* n p) s) = *zero-heap-on* A s
⟨*proof*⟩

lemma *zero-heap-on-stack-releases'*:
fixes p:: $'a$::*mem-type ptr*
assumes *guard*: $(\bigwedge i.\ i < n \Longrightarrow c\text{-}null\text{-}guard\ (p +_p int\ i))$
shows *zero-heap-on* (*stack-free* (*stack-releases n p* (*htd s*)) ∪ A) (*htd-upd* (*stack-releases n p*) s) =
zero-heap-on (*stack-free* (*htd s*) ∪ {*ptr-val p..+n* * *size-of TYPE*($'a$)} ∪ A) s
⟨*proof*⟩

We want to express that certain portions of the heap (typing and values) are 'irrelevant', in particular regarding (free) stack space. The notion of 'irrelevant' is a bit vague, it means that the behaviour of the program does not depend on those locations and also that it does not modify those locations. Moreover, we prefer an abstraction function rather than an relation between states to avoid admissibility issues for refinement.

definition *frame*:: *addr set* ⇒ $'s$ ⇒ $'s$ ⇒ $'s$ **where**
frame A s_0 s =
hmem-upd (λ*h*. *override-on h* (*hmem* s_0) (A ∪ *stack-free* (*htd s*)))
(*htd-upd* (λ*d*. *override-on d* (*htd* s_0) (A − *stack-free* (*htd s*))) s)

The standard use case we have in mind is $A \cap$ *stack-free* (*htd s*) = {}, hence A − *stack-free* (*htd s*) = A, but nevertheless the intuition is:

- stack free typing from s is preserved, the framed sate has at least as many stack free addresses as the original one. So we can simulate any stack allocation.
- heap values for stack free and A are taken from reference state s_0, this captures that we do not depend on the original values in s for those addresses.
- typing for allocations in A is taken from reference state s_0, this captures that we do not depend on the original typing in s for those addresses.

  By taking the same reference state s_0 to frame two states s and s', we can express that the 'irrelevant' parts of the heap did not change in the respective framed states *frame* A s_0 s and *frame* A s_0 s'.

definition *raw-frame*:: *addr set* ⇒ $'s$ ⇒ $'s$ ⇒ $'s$ **where**
raw-frame A s_0 s =
hmem-upd (λ*h*. *override-on h* (*hmem* s_0) A)
(*htd-upd* (λ*d*. *override-on d* (*htd* s_0) A) s)

lemma *frame-heap-independent-selector*:
$(\bigwedge f\ s.\ sel\ (hmem\text{-}upd\ f\ s) = sel\ s) \Longrightarrow (\bigwedge f\ s.\ sel\ (htd\text{-}upd\ f\ s) = sel\ s) \Longrightarrow$
$sel\ (frame\ A\ s_0\ s) = sel\ s$
⟨*proof*⟩

lemma *frame-id*[*simp*]: $frame\ A\ s\ s = s$
⟨*proof*⟩

lemma *frame-hmem-UNIV*[*simp*]: $hmem\ (frame\ UNIV\ s_0\ s) = hmem\ s_0$
⟨*proof*⟩

lemma *hmem-frame*: $hmem\ (frame\ A\ s_0\ s) = (\lambda a.\ if\ a \in A \cup stack\text{-}free\ (htd\ s)\ then\ hmem\ s_0\ a\ else\ hmem\ s\ a)$
⟨*proof*⟩

lemma *htd-frame*: $htd\ (frame\ A\ s_0\ s) = (\lambda a.\ if\ a \in (A - stack\text{-}free\ (htd\ s))\ then\ htd\ s_0\ a\ else\ htd\ s\ a)$
⟨*proof*⟩

lemma *stack-free-htd-frame*: $stack\text{-}free\ (htd\ s) \subseteq stack\text{-}free\ (htd\ (frame\ A\ s_0\ s))$
⟨*proof*⟩

lemma *stack-free-htd-frame'*: $stack\text{-}free\ (htd\ s_0) \cap A = \{\} \Longrightarrow stack\text{-}free\ (htd\ (frame\ A\ s_0\ s)) = stack\text{-}free\ (htd\ s)$
⟨*proof*⟩

lemma *equal-on-hmem-frame*: $equal\text{-}on\ (A \cup stack\text{-}free\ (htd\ s))\ (hmem\ (frame\ A\ s_0\ s))\ (hmem\ s_0)$
⟨*proof*⟩

lemma *equal-upto-hmem-frame*: $equal\text{-}upto\ (A \cup stack\text{-}free\ (htd\ s))\ (hmem\ (frame\ A\ s_0\ s))\ (hmem\ s)$
⟨*proof*⟩

lemma *equal-on-htd-frame*: $stack\text{-}free\ (htd\ s) \cap A = \{\} \Longrightarrow equal\text{-}on\ A\ (htd\ (frame\ A\ s_0\ s))\ (htd\ s_0)$
⟨*proof*⟩

lemma *equal-on-htd-stack-free-frame*: $stack\text{-}free\ (htd\ s) \cap A = \{\} \Longrightarrow$
$equal\text{-}on\ (stack\text{-}free\ (htd\ s))\ (htd\ (frame\ A\ s_0\ s))\ (htd\ s)$
⟨*proof*⟩

lemma *equal-upto-htd-frame*: $equal\text{-}upto\ A\ (htd\ (frame\ A\ s_0\ s))\ (htd\ s)$
⟨*proof*⟩

lemma *equal-upto-heap-on-frame*: $equal\text{-}upto\text{-}heap\text{-}on\ (A \cup stack\text{-}free\ (htd\ s))\ (frame\ A\ s_0\ s)\ s$

⟨*proof*⟩

lemma *frame-heap-update*:
fixes *p*::*'a*::*mem-type ptr*
shows *ptr-span p* ⊆ *A* ⟹ *frame A s*$_0$ (*hmem-upd* (*heap-update p v*) *s*) = *frame A s*$_0$ *s*
⟨*proof*⟩

lemma *frame-heap-update-disjoint*:
fixes *p*::*'a*::*mem-type ptr*
shows *ptr-span p* ∩ *A* = {} ⟹ *ptr-span p* ∩ *stack-free* (*htd s*) = {} ⟹
frame A s$_0$ (*hmem-upd* (*heap-update p v*) *s*) = *hmem-upd* (*heap-update p v*) (*frame A s*$_0$ *s*)
⟨*proof*⟩

lemma *h-val-frame-disjoint'*:
fixes *p*::*'a*::*mem-type ptr*
shows *ptr-span p* ∩ *A* = {} ⟹ *ptr-span p* ∩ *stack-free* (*htd s*) = {} ⟹
ptr-span p = *ptr-span p'* ⟹
h-val (*hmem* (*frame A s*$_0$ *s*)) *p'* = *h-val* (*hmem s*) *p'*
⟨*proof*⟩

lemma *h-val-frame-disjoint*:
fixes *p*::*'a*::*mem-type ptr*
shows *ptr-span p* ∩ *A* = {} ⟹ *ptr-span p* ∩ *stack-free* (*htd s*) = {} ⟹
h-val (*hmem* (*frame A s*$_0$ *s*)) *p* = *h-val* (*hmem s*) *p*
⟨*proof*⟩

lemma *h-val-frame-disjoint-globals*:
fixes *p*::*'a*::*mem-type ptr*
assumes *G* ∩ *A* = {} *G* ∩ *stack-free* (*htd s*) = {}
assumes *ptr-span p* ⊆ *G*
shows *h-val* (*hmem* (*frame A s*$_0$ *s*)) *p* = *h-val* (*hmem s*) *p*
⟨*proof*⟩

lemma *frame-heap-update-padding*:
fixes *p*::*'a*::*mem-type ptr*
shows *ptr-span p* ⊆ *A* ⟹ *length bs* = *size-of TYPE*('*a*) ==> *frame A s*$_0$ (*hmem-upd* (*heap-update-padding p v bs*) *s*) = *frame A s*$_0$ *s*
⟨*proof*⟩

lemma *frame-stack-alloc*:
fixes *p*::*'a*::*mem-type ptr*
assumes *subset*: *ptr-span p* ⊆ *stack-free* (*htd s*)
assumes *disjoint*: *stack-free* (*htd s*) ∩ *A* = {}
assumes *alloc*: *stack-free* (*ptr-force-type p* (*htd s*)) = *stack-free* (*htd s*) − *ptr-span p*
shows
frame (*ptr-span p* ∪ *A*) (*htd-upd* (λ*d*. *override-on d stack-byte-typing* (*ptr-span*

$p))\ t_0)$
$(htd\text{-}upd\ (\lambda\text{-}.\ ptr\text{-}force\text{-}type\ p\ (htd\ s))\ s) = frame\ A\ t_0\ s$
⟨*proof*⟩

lemma *frame-stack-release*:
fixes $p::'a::mem\text{-}type\ ptr$
assumes *disjoint-free*: $ptr\text{-}span\ p \cap stack\text{-}free\ (htd\ s) = \{\}$
assumes *disjoint-old*: $ptr\text{-}span\ p \cap A = \{\}$
assumes *c-null-guard*: $c\text{-}null\text{-}guard\ p$
assumes *lbs*: $length\ bs = size\text{-}of\ TYPE\ ('a)$
shows $frame\ (ptr\text{-}span\ p \cup A)\ (htd\text{-}upd\ (\lambda d.\ override\text{-}on\ d\ stack\text{-}byte\text{-}typing\ (ptr\text{-}span\ p))\ t_0)\ s =$
$frame\ A\ t_0\ (hmem\text{-}upd\ (heap\text{-}update\text{-}list\ (ptr\text{-}val\ p)\ bs)\ (htd\text{-}upd\ (stack\text{-}releases\ (Suc\ 0)\ p)\ s))$
⟨*proof*⟩

lemma *frame-stack-release-keep*:
fixes $p::'a::mem\text{-}type\ ptr$
assumes *disjoint-free*: $ptr\text{-}span\ p \cap stack\text{-}free\ (htd\ s) = \{\}$
assumes *disjoint-old*: $ptr\text{-}span\ p \cap A = \{\}$
assumes *c-null-guard*: $c\text{-}null\text{-}guard\ p$
assumes *lbs*: $length\ bs = size\text{-}of\ TYPE('a)$
shows $hmem\text{-}upd\ (heap\text{-}update\text{-}list\ (ptr\text{-}val\ p)\ (heap\text{-}list\ (hmem\ t_0)\ (size\text{-}of\ TYPE('a))\ (ptr\text{-}val\ p)))$
$(\ htd\text{-}upd\ (stack\text{-}releases\ (Suc\ 0)\ p)\ (frame\ A\ t_0\ s)) =$
$frame\ A\ t_0$
$(hmem\text{-}upd\ (heap\text{-}update\text{-}list\ (ptr\text{-}val\ p)\ bs)$
$(htd\text{-}upd\ (stack\text{-}releases\ (Suc\ 0)\ p)\ s))$
⟨*proof*⟩

lemma *symp-equal-upto-heap-on*: $symp\ (equal\text{-}upto\text{-}heap\text{-}on\ S)$
⟨*proof*⟩

lemma *equal-upto-heap-on-commute*: $equal\text{-}upto\text{-}heap\text{-}on\ S\ s\ t \longleftrightarrow equal\text{-}upto\text{-}heap\text{-}on\ S\ s\ t$
⟨*proof*⟩

lemma *equal-upto-heap-on-trans*[*trans*]:
assumes *s-t*: $equal\text{-}upto\text{-}heap\text{-}on\ S\ s\ t$
assumes *t-u*: $equal\text{-}upto\text{-}heap\text{-}on\ S\ t\ u$
shows $equal\text{-}upto\text{-}heap\text{-}on\ S\ s\ u$
⟨*proof*⟩

lemma *equal-upto-heap-on-mono*: $S \subseteq T \Longrightarrow equal\text{-}upto\text{-}heap\text{-}on\ S\ s\ t \Longrightarrow equal\text{-}upto\text{-}heap\text{-}on\ T\ s\ t$
⟨*proof*⟩

end

lemma *runs-to-partial-L2-modify*[*runs-to-vcg*]:
(*L2-modify f*) · *s* ?⦃λ*r t*. *r* = *Result* () ∧ *t* = *f s*⦄
⟨*proof*⟩

lemma *runs-to-partial-L2-unknown*[*runs-to-vcg*]:
(⋀*x*. *P* (*Result x*) *s*) ⟹ (*L2-unknown ns*) · *s* ?⦃*P*⦄
⟨*proof*⟩

lemma *runs-to-partial-L2-seq*[*runs-to-vcg*]:
f · *s* ?⦃ λ*r t*. (∀ *a*. *r* = *Result a* ⟶ *g a* · *t* ?⦃ *Q* ⦄) ∧ (∀ *e*. *r* = *Exn e* ⟶ *Q* (*Exn e*) *t*) ⦄ ⟹
(*L2-seq f g*) · *s* ?⦃ *Q* ⦄
⟨*proof*⟩

lemma (**in** *heap-typing-state*) *runs-to-partial-L2-seq*[*unchanged-typing*]:
assumes [*runs-to-vcg*]: *f* · *s* ?⦃λ*r t*. *unchanged-typing-on S s t*⦄
assumes [*runs-to-vcg*]: ⋀*r s*. (*g r*) · *s* ?⦃λ*r t*. *unchanged-typing-on S s t*⦄
shows (*L2-seq f g*) · *s* ?⦃λ*r t*. *unchanged-typing-on S s t*⦄
⟨*proof*⟩

lemma *runs-to-partial-L2-gets*[*runs-to-vcg*]:
P (*Result* (*f s*)) *s* ⟹ (*L2-gets f ns*) · *s* ?⦃ *P* ⦄
⟨*proof*⟩

lemma *runs-to-partial-L2-condition*[*runs-to-vcg*]:
(*P s* ⟹ *f* · *s* ?⦃*Q*⦄) ⟹ (¬ *P s* ⟹ *g* · *s* ?⦃*Q*⦄) ⟹ (*L2-condition P f g*) · *s* ?⦃*Q*⦄
⟨*proof*⟩

lemma *runs-to-partial-L2-catch*[*runs-to-vcg*]:
assumes [*runs-to-vcg*]: *f* · *s* ?⦃ λ*r t*. (∀ *e*. *r* = *Exn e* ⟶ ((*g e*) · *t* ?⦃ *P* ⦄)) ∧ (∀ *v*′. *r* = *Result v*′ ⟶ *P* (*Result v*′) *t*) ⦄
shows (*L2-catch f g*) · *s* ?⦃*P* ⦄
⟨*proof*⟩

lemma (**in** *heap-typing-state*) *runs-to-partial-L2-catch*[*unchanged-typing*]:
assumes [*runs-to-vcg*]: *f* · *s* ?⦃λ*r t*. *unchanged-typing-on S s t*⦄
assumes [*runs-to-vcg*]: ⋀*r s*. (*g r*) · *s* ?⦃λ*r t*. *unchanged-typing-on S s t*⦄
shows (*L2-catch f g*) · *s* ?⦃λ*r t*. *unchanged-typing-on S s t*⦄
⟨*proof*⟩

lemma *runs-to-partial-L2-try*[*runs-to-vcg*]:
assumes [*runs-to-vcg*]:
f · *s* ?⦃ λ*r*. *Q* (*unnest-exn r*) ⦄
shows (*L2-try f*) · *s* ?⦃ *Q* ⦄

⟨*proof*⟩

lemma (**in** *heap-typing-state*) *runs-to-partial-L2-try*[*unchanged-typing*]:
assumes [*runs-to-vcg*]: $f \cdot s$?⦃$\lambda r\ t.$ *unchanged-typing-on* $S\ s\ t$⦄
shows (*L2-try* f) $\cdot\ s$?⦃$\lambda r\ t.$ *unchanged-typing-on* $S\ s\ t$⦄
⟨*proof*⟩

lemma *runs-to-partial-L2-while*:
assumes B: $\bigwedge r\ s.\ I\ (\mathit{Result}\ r)\ s \Longrightarrow C\ r\ s \Longrightarrow (B\ r) \cdot s$?⦃I⦄
assumes Qr: $\bigwedge r\ s.\ I\ (\mathit{Result}\ r)\ s \Longrightarrow \neg\ C\ r\ s \Longrightarrow Q\ (\mathit{Result}\ r)\ s$
assumes Ql: $\bigwedge e\ s.\ I\ (\mathit{Exn}\ e)\ s \Longrightarrow Q\ (\mathit{Exn}\ e)\ s$
assumes I: $I\ (\mathit{Result}\ r)\ s$
shows (*L2-while* $C\ B\ r\ ns$) $\cdot\ s$?⦃ Q ⦄
⟨*proof*⟩

lemma *runs-to-partial-L2-while-same-post*:
assumes B: $\bigwedge r\ s.\ Q\ (\mathit{Result}\ r)\ s \Longrightarrow C\ r\ s \Longrightarrow (B\ r) \cdot s$?⦃Q⦄
assumes I: $Q\ (\mathit{Result}\ r)\ s$
shows (*L2-while* $C\ B\ r\ ns$) $\cdot\ s$?⦃ Q ⦄
⟨*proof*⟩

lemma (**in** *heap-typing-state*) *runs-to-partial-L2-while-unchanged-typing*[*unchanged-typing*]:
assumes B[*runs-to-vcg*]: $\bigwedge r\ s.\ C\ r\ s \Longrightarrow (B\ r) \cdot s$?⦃$\lambda r\ t.$ *unchanged-typing-on* $S\ s\ t$⦄
shows (*L2-while* $C\ B\ r\ ns$) $\cdot\ s$?⦃$\lambda r\ t.$ *unchanged-typing-on* $S\ s\ t$⦄
⟨*proof*⟩

lemma *runs-to-partial-L2-throw*[*runs-to-vcg*]:
assumes $P\ (\mathit{Exn}\ e)\ s$
shows (*L2-throw* $e\ ns$) $\cdot\ s$?⦃ P ⦄
⟨*proof*⟩

lemma *runs-to-partial-L2-spec*[*runs-to-vcg*]: (*L2-spec* R) $\cdot\ s$?⦃$\lambda r\ t.\ (s,\ t) \in R$⦄
⟨*proof*⟩

lemma *runs-to-partial-state-L2-assume*[*runs-to-vcg*]:
(*L2-assume* R) $\cdot\ s$?⦃$\lambda r\ t.\ \exists x.\ r = \mathit{Result}\ x \wedge (x,\ t) \in R\ s$⦄
⟨*proof*⟩

lemma *runs-to-partial-L2-guard*[*runs-to-vcg*]:
shows (*L2-guard* P) $\cdot\ s$?⦃ $\lambda r\ t.\ r = \mathit{Result}\ () \wedge s = t \wedge P\ s$⦄
⟨*proof*⟩

definition *GUARDED-ASSM* :: *bool* ⇒ *bool* **where** [*remove-ASSMs*]: *GUARDED-ASSM* $P \longleftrightarrow P$
lemma *GUARDED-ASSM-D*: *GUARDED-ASSM* $P \Longrightarrow P$ ⟨*proof*⟩

lemma *runs-to-partial-L2-guarded*[*runs-to-vcg*]:
(*GUARDED-ASSM* (*P s*) $\Longrightarrow$ *c* $\cdot$ *s* ?{| *Q* |}) $\Longrightarrow$ (*L2-guarded P c*) $\cdot$ *s* ?{| *Q* |}
⟨*proof*⟩

lemma *runs-to-partial-L2-fail-conv*[*simp*]: *L2-fail* $\cdot$ *s* ?{| *R* |} $\longleftrightarrow$ *True*
⟨*proof*⟩

lemma *runs-to-partial-L2-fail*[*runs-to-vcg*]: *L2-fail* $\cdot$ *s* ?{| *R* |}
⟨*proof*⟩

lemma *runs-to-partial-L2-call*[*runs-to-vcg*]:
assumes [*runs-to-vcg*]: *m* $\cdot$ *s* ?{| $\lambda r.$ *Q* (*case r of Exn e* $\Rightarrow$ *Exn* (*f e*) | *Result r* $\Rightarrow$ *Result r*) |}
shows (*L2-call m f ns*) $\cdot$ *s* ?{| *Q* |}
⟨*proof*⟩

end

20.4 In Out Parameter Refinement

theory *In-Out-Parameters*
imports
L2ExceptionRewrite
L2Peephole
TypHeapSimple
Stack-Typing
begin

lemma *map-exn-catch-conv*: *map-value* (*map-exn f*) *m* = (*m* <*catch*> ($\lambda r.$ *throw* (*f r*)))
⟨*proof*⟩

abbreviation *L2-return x ns* ≡ *liftE* (*L2-VARS* (*return x*) *ns*)

lemma *L2-return-L2-gets-conv*: *L2-return x ns* = *L2-gets* (λ-. *x*) *ns*
⟨*proof*⟩

lemma *return-L2-gets-conv*: (*return x*) = *L2-gets* (λ-. *x*) []
⟨*proof*⟩

named-theorems *refines-right-eq*

lemma (**in** *heap-state*) *IO-modify-heap-paddingE-root-refines'*:
fixes *p*::'*a*::*xmem-type ptr*
fixes *fld-update*::('*b*::*xmem-type* ⇒ '*b*) ⇒ '*a* ⇒ '*a*
assumes *fg-cons*: *fg-cons fld* (*fld-update* ∘ (λ*x* -. *x*))

assumes *fl*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f 0* = *Some* (*adjust-ti* (*typ-info-t TYPE*('*b*::*xmem-type*)) *fld* (*fld-update* ∘ (λ*x* -. *x*)), *n*)
assumes *cgrd*: *c-guard p*
shows *refines*
(*IO-modify-heap-paddingE* (*PTR*('*b*) &(*p*→*f*)) *v*)
(*IO-modify-heap-paddingE p* (λ*s*. (*fld-update* (λ-. *v s*)) (*h-val* (*hmem s*) *p*)))
s s ((=))
⟨*proof*⟩

lemma (**in** *heap-state*) *IO-modify-heap-paddingE-root-refines''*:
fixes *p*::'*a*::*xmem-type ptr*
fixes *fld-update*::('*b*::*xmem-type* ⇒ '*b*) ⇒ '*a* ⇒ '*a*
assumes *fg-cons*: *fg-cons fld* (*fld-update* ∘ (λ*x* -. *x*))
assumes *fl*: *field-ti TYPE*('*a*) *f* = *Some* (*adjust-ti* (*typ-info-t TYPE*('*b*::*xmem-type*)) *fld* (*fld-update* ∘ (λ*x* -. *x*)))
assumes *cgrd*: *c-guard p*
shows *refines*
(*IO-modify-heap-paddingE* (*PTR*('*b*) &(*p*→*f*)) *v*)
(*IO-modify-heap-paddingE p* (λ*s*. (*fld-update* (λ-. *v s*)) (*h-val* (*hmem s*) *p*)))
s s ((=))
⟨*proof*⟩

lemma (**in** *heap-state*) *IO-modify-heap-paddingE-root-refines* [*refines-right-eq*]:
fixes *p*::'*a*::*xmem-type ptr*
fixes *fld-update*::('*b*::*xmem-type* ⇒ '*b*) ⇒ '*a* ⇒ '*a*
assumes *v'*: ⋀*s*. *v' s* = ((*fld-update* (λ-. *v s*)) (*h-val* (*hmem s*) *p*))
assumes *fg-cons*: *fg-cons fld* (*fld-update* ∘ (λ*x* -. *x*))
assumes *fl*: *field-ti TYPE*('*a*) *f* = *Some* (*adjust-ti* (*typ-info-t TYPE*('*b*::*xmem-type*)) *fld* (*fld-update* ∘ (λ*x* -. *x*)))
assumes *cgrd*: *c-guard p*
shows *refines*
(*IO-modify-heap-paddingE* (*PTR*('*b*) &(*p*→*f*)) *v*)
(*IO-modify-heap-paddingE p v'*)
s s ((=))
⟨*proof*⟩

lemma *refines-subst-right*:
assumes *f-g*: *refines f g s t Q*
assumes *refines-eq*: *refines g g' t t* ((=))
shows *refines f g' s t Q*
⟨*proof*⟩

lemma *refines-right-eq-id*: *refines f f s s* ((=))
⟨*proof*⟩

lemma *refines-subst-left*:

assumes *f-g*: *refines f g s t Q*
assumes *f-eq*: *run f s = run f' s*
shows *refines f' g s t Q*
⟨*proof*⟩

lemma *refines-right-eq-L2-seq* [*refines-right-eq*]:
assumes *f1*: *refines f1 g1 s s* ((=))
assumes *f2*: ⋀*v t*. *refines* (*f2 v*) (*g2 v*) *t t* ((=))
shows *refines* (*L2-seq f1 f2*) (*L2-seq g1 g2*) *s s* ((=))
⟨*proof*⟩

lemma *refines-right-eq-L2-guard'*:
assumes *Q s* ⟹ *P s*
assumes *Q s* ⟹ *refines X X' s s* (=)
shows *refines* (*L2-seq* (*L2-guard P*) (λ-. *X*)) (*L2-seq* (*L2-guard Q*) (λ-. *X'*)) *s s* ((=))
⟨*proof*⟩

lemma *refines-right-eq-L2-guard*:
assumes *Q* ⟹ *P*
assumes *Q* ⟹ *refines X X' s s* (=)
shows *refines* (*L2-seq* (*L2-guard* (λ-. *P*)) (λ-. *X*)) (*L2-seq* (*L2-guard* (λ-. *Q*)) (λ-. *X'*)) *s s* ((=))
⟨*proof*⟩

lemma *refines-L2-guarded*:
assumes *grd*: *g' t* ⟹ *g s*
assumes *X-X'*: *g s* ⟹ *g' t* ⟹ *refines X X' s t Q*
shows *refines* (*L2-guarded g X*) (*L2-guarded g' X'*) *s t Q*
⟨*proof*⟩

lemma *select-UNIV-L2-unknown-conv*: (*select UNIV*) = *L2-unknown ns*
⟨*proof*⟩

lemma *select-singleton-conv*: ((*select* ({*x*})) >>= *g*) = *g x*
⟨*proof*⟩

lemma *hd-UNIV*: *hd* ‘ *UNIV* ⊆ *UNIV*
⟨*proof*⟩

lemma *hd-singleton*: *hd* ‘ {[*x*]} ⊆ {*x*}
⟨*proof*⟩

lemma (**in** *heap-state*) *refines-exec-IO-modify-heap-padding-step-right*:
fixes *p*:: *'a*::*mem-type ptr*

assumes
refines (*return x*) *g s*
(*hmem-upd* (*heap-update-padding p v* (*heap-list* (*hmem s*) (*size-of TYPE*('*a*)) (*ptr-val p*))) *t*) *Q*
shows *refines* (*return x*) (*do* { - <− *IO-modify-heap-paddingE p* (λ-. *v*); *g* }) *s t Q*
⟨*proof*⟩

lemma (**in** *heap-state*) *refines-exec-IO-modify-heap-padding-single-step-right*:
fixes *p*:: '*a*::*mem-type ptr*
assumes *Q* (*Result* (), *s*) (*Result* (), *hmem-upd* (*heap-update-padding p* (*v t*) (*heap-list* (*hmem s*) (*size-of TYPE*('*a*)) (*ptr-val p*))) *t*)
shows *refines* (*return* ()) (*IO-modify-heap-paddingE p v*) *s t Q*
⟨*proof*⟩

lemma (**in** *heap-state*) *refines-exec-IO-modify-heap-padding-single-step-right-InL*:
fixes *p*:: '*a*::*mem-type ptr*
assumes *Q* (*Exn e*, *s*) (*Exn e'*, *hmem-upd* (*heap-update-padding p* (*v t*) (*heap-list* (*hmem s*) (*size-of TYPE*('*a*)) (*ptr-val p*))) *t*)
shows *refines* (*throw e*)
(*do* { - <− *IO-modify-heap-paddingE p v*;
L2-throw e' ns
})
s t Q
⟨*proof*⟩

lemma *refines-exec-gets-right*:
assumes *Q* (*Result x*, *s*) (*Result* (*g t*), *t*)
shows *refines* (*return x*) (*gets g*) *s t Q*
⟨*proof*⟩

lemma *refines-exec-L2-return-right*:
assumes *Q* (*Result x*, *s*) (*Result w*, *t*)
shows *refines* (*return* (*x*)) (*L2-return w ns*) *s t Q*
⟨*proof*⟩

lemma *refines-L2-catch-right*:
assumes *f*: *refines f g s t Q*
assumes *Res-Res*: ⋀*v v' s' t'*. *Q* (*Result v*, *s'*) (*Result v'*, *t'*) ⟹ *R* (*Result v*, *s'*) (*Result v'*, *t'*)
assumes *Res-Exn*: ⋀*v e' s' t'*. *Q* (*Result v*, *s'*) (*Exn e'*, *t'*) ⟹ *refines* (*return v*) (*h e'*) *s' t' R*
assumes *Exn-Res*: ⋀*e v' s' t'*. *Q* (*Exn e*, *s'*) (*Result v'*, *t'*) ⟹ *R* (*Exn e*, *s'*) (*Result v'*, *t'*)
assumes *Exn-Exn*: ⋀*e e' s' t'*. *Q* (*Exn e*, *s'*) (*Exn e'*, *t'*) ⟹ *refines* (*throw e*)

$(h\ e')\ s'\ t'\ R$
shows *refines f* (*L2-catch g h*) *s t R* ⟨*proof*⟩

lemma *L2-seq-gets-app-conv*: *run* (*L2-seq* (*L2-gets f ns*) *g*) *s* = *run* (*g* (*f s*)) *s*
⟨*proof*⟩

lemma *refines-project-right*:
assumes *f-g*: *refines f g s t Q*
assumes *run g t* = *run* (*g'* (*prj t*)) *t*
shows *refines f* (*L2-seq* (*L2-gets prj ns*) *g'*) *s t Q*
⟨*proof*⟩

lemma *refines-project-guard-right*:
assumes *f-g*: *refines f* (*L2-seq* (*L2-guard P*) (λ-. *g*)) *s t Q*
assumes *P t* ⟹ *run g t* = *run* (*g'* (*prj t*)) *t*
shows *refines f* (*L2-seq* (*L2-guard P*) (λ-. (*L2-seq* (*L2-gets prj ns*) *g'*))) *s t Q*
⟨*proof*⟩

named-rules *cguard-assms* **and** *alloc-assms* **and** *modifies-assms* **and** *disjoint-assms* **and**
disjoint-alloc **and** *disjoint-stack-free* **and** *stack-ptr* **and** *h-val-globals-frame-eq* **and**
rel-alloc-independent-globals **and** *keep-non-stack-ptr-eqs*
synthesize-rules *refines-in-out*
⟨*ML*⟩

lemma *refines-yield-both*[*simp*]: *refines* (*return a*) (*return b*) *s t R* ⟷ *R* (*Result a*, *s*) (*Result b*, *t*)
⟨*proof*⟩

lemma *disjoint-union-distrib*: $A \cap (B \cup C) = \{\} \longleftrightarrow A \cap B = \{\} \wedge A \cap C = \{\}$
⟨*proof*⟩

lemma *inter-commute*: $A \cap B = B \cap A$ ⟨*proof*⟩
lemma *disjoint-symmetric*: $A \cap B = \{\} \Longrightarrow B \cap A = \{\}$
⟨*proof*⟩
lemma *disjoint-symmetric'*: $A \cap B \equiv \{\} \Longrightarrow B \cap A = \{\}$
⟨*proof*⟩

definition *IOcorres*::
('*s* ⇒ *bool*) ⇒
('*s* ⇒ ('*e*,'*a*) *xval* ⇒ '*s* ⇒ *bool*) ⇒
('*s* ⇒ '*t*) ⇒
('*a* ⇒ '*s* ⇒ '*b*) ⇒
('*e* ⇒ '*s* ⇒ '*e1*) ⇒

(*'e1*, *'b*, *'t*) *exn-monad* =>
(*'e*, *'a*, *'s*) *exn-monad* => *bool* **where**
IOcorres P Q st rx ex f_a $f_c \equiv$ *corresXF-post st rx ex P Q* f_a f_c

lemma *IOcorres-id*: *IOcorres* (λ-. *True*) (λ - - -. *True*) (λ*s*. *s*) (λ*r* -. *r*) (λ*r* -. *r*) *f f*
⟨*proof*⟩
lemmas *IOcorres-skip* = *IOcorres-id*

lemma *IOcorres-trivial-from-local-var-extract*:
L2corres st rx ex P A C ⟹ *IOcorres* (λ-. *True*) (λ - - -. *True*) (λ*s*. *s*) (λ*r* -. *r*) (λ*r* -. *r*) *A A*
⟨*proof*⟩

lemma *admissible-IOcorres* [*corres-admissible*]:
ccpo.admissible Inf (≥) (λf_a. *IOcorres P Q st rx ex* f_a f_c)
⟨*proof*⟩

lemma *IOcorres-top* [*corres-top*]: *IOcorres P Q st rx ex* ⊤ f_c
⟨*proof*⟩

lemma *distinct-addresses-ptr-val-lemma*:
n < *addr-card* ⟹ *ptr-val p* + *word-of-nat n* ∉ (λ*x*. *ptr-val p* + *word-of-nat x*) ‘ {*0*..<*n*}
⟨*proof*⟩

lemma *distinct-addresses-ptr-lemma*:
assumes *bound*: *n* ≤ *addr-card*
shows *distinct* (*map* (λ*i*. *ptr-val p* + *of-nat i*) [*0*..<*n*])
⟨*proof*⟩

lemma *array-intvl-Suc-split*:{*a*..+*Suc n* ∗ *m*} = {*a*..+*n* ∗ *m*} ∪ {*a* + (*word-of-nat* (*n*∗*m*))..+*m*}
⟨*proof*⟩

definition *rel-singleton-stack* :: *'a*::*c-type ptr* ⇒ *heap-mem* ⇒ *unit* ⇒ *'a* ⇒ *bool* **where**
rel-singleton-stack p h = (λ(-::*unit*) *v*.
v = *h-val h p*)

lemma *domain-rel-singleton-stack*:
equal-on (*ptr-span p*) *h h'* ⟹ *rel-singleton-stack p h* = *rel-singleton-stack p h'*
⟨*proof*⟩

named-theorems *rel-stack-intros* **and** *rel-stack-simps* **and** *rel-entail*

lemma *rel-singleton-stack-simp* [*rel-stack-simps*]:
rel-singleton-stack p h x v ⟷ *v* = *h-val h p*
⟨*proof*⟩

lemma *rel-stack-refl* [*rel-stack-intros*]: (λ-. (=)) *h x x*
⟨*proof*⟩

lemma *rel-singleton-stackI* [*rel-stack-intros*]: *rel-singleton-stack p h x* (*h-val h p*)
⟨*proof*⟩

lemma *rel-singleton-stack-condI* [*rel-stack-intros*]: *h-val h p* = *y* ⟹ *rel-singleton-stack p h x y*
⟨*proof*⟩

lemma *fun-of-rel-singleton-stack*[*fun-of-rel-intros*]: *fun-of-rel* (*rel-singleton-stack p h*) (λ-. (*h-val h p*))
⟨*proof*⟩

lemma *funp-rel-singleton-stack*[*funp-intros*, *corres-admissible*]: *funp* (*rel-singleton-stack p h*)
⟨*proof*⟩

definition *rel-push* ::
'a::*c-type ptr* ⇒ (*heap-mem* ⇒ *'b* ⇒ *'c* ⇒ *bool*) ⇒ *heap-mem* ⇒
'b ⇒ *'a* × *'c* ⇒ *bool*
where
rel-push p R h = (λ*r* (*v*, *x*).
R h r x ∧
v = *h-val h p*)

lemma *rel-singleton-stack-rel-push-conv*: *rel-singleton-stack p* = (λ*h x y*. *rel-push p* (λ- - -. *True*) *h x* (*y*, ()))
⟨*proof*⟩

lemma *rel-push-simp*[*rel-stack-simps*]: *rel-push p R h r* (*v*, *x*) ⟷
R h r x ∧ *v* = *h-val h p*
⟨*proof*⟩

lemma *fun-of-rel-puhsh* [*fun-of-rel-intros*]:
fun-of-rel (*R h*) *f* ⟹ *fun-of-rel* (*rel-push p R h*) (λ*x*. (*h-val h p*, *f x*))
⟨*proof*⟩

lemma *funp-rel-push*[*funp-intros*, *corres-admissible*]: *funp* (*R h*) ⟹ *funp* (*rel-push*

p R h)
⟨*proof*⟩

lemma *rel-push-stackI* [*rel-stack-intros*]: *Q h x y* ⟹ *rel-push p Q h x* ((*h-val h p*), *y*)
⟨*proof*⟩

lemma *rel-push-stack-condI* [*rel-stack-intros*]: *h-val h p* = *v* ⟹ *Q h x y* ⟹ *rel-push p Q h x* (*v*, *y*)
⟨*proof*⟩

definition *rel-sum-stack L R h* = *rel-sum* (*L h*) (*R h*)
definition *rel-xval-stack L R h* = *rel-xval* (*L h*) (*R h*)

lemma *rel-sum-stack-expand-sum-eq*:
(*rel-sum-stack* (λ-. (=)) *R*) = (*rel-sum-stack* (*rel-sum-stack* (λ-. (=)) (λ-. (=))) *R*)
⟨*proof*⟩

lemma *rel-xval-stack-expand-sum-eq*:
(*rel-xval-stack* (λ-. (=)) *R*) = (*rel-xval-stack* (*rel-sum-stack* (λ-. (=)) (λ-. (=))) *R*)
⟨*proof*⟩

lemma *rel-sum-stack-expand-sum-bot*:
(λ- - -. *False*) = (*rel-sum-stack* (λ- - -. *False*) (λ- - - . *False*))
⟨*proof*⟩

lemma *rel-xval-stack-expand-xval-bot*:
(λ- - -. *False*) = (*rel-xval-stack* (λ- - -. *False*) (λ- - - . *False*))
⟨*proof*⟩

lemma *rel-sum-stack-entail*:
assumes *L*: ⋀*v v'*. *L' h v v'* ⟹ *L h v v'*
assumes *R*: ⋀*v v'*. *R' h v v'* ⟹ *R h v v'*
assumes *rel-sum-stack L' R' h x y*
shows *rel-sum-stack L R h x y*
⟨*proof*⟩

lemma *rel-xval-stack-entail*:
assumes *L*: ⋀*v v'*. *L' h v v'* ⟹ *L h v v'*
assumes *R*: ⋀*v v'*. *R' h v v'* ⟹ *R h v v'*
assumes *rel-xval-stack L' R' h x y*
shows *rel-xval-stack L R h x y*
⟨*proof*⟩

lemma *rel-sum-stack-intros*:
L h l1 l2 ⟹ *rel-sum-stack L R h* (*Inl l1*) (*Inl l2*)

R h r1 r2 $\Longrightarrow$ *rel-sum-stack L R h* (*Inr r1*) (*Inr r2*)
⟨*proof*⟩

lemma *rel-xval-stack-intros*:
L h l1 l2 $\Longrightarrow$ *rel-xval-stack L R h* (*Exn l1*) (*Exn l2*)
R h r1 r2 $\Longrightarrow$ *rel-xval-stack L R h* (*Result r1*) (*Result r2*)
⟨*proof*⟩

lemma *fun-of-rel-sum-stack*[*fun-of-rel-intros*]:
fun-of-rel (*L h*) *f-l* $\Longrightarrow$ *fun-of-rel* (*R h*) *f-r* $\Longrightarrow$ *fun-of-rel* (*rel-sum-stack L R h*) (*sum-map f-l f-r*)
⟨*proof*⟩

lemma *fun-of-rel-xval-stack*[*fun-of-rel-intros*]:
fun-of-rel (*L h*) *f-l* $\Longrightarrow$ *fun-of-rel* (*R h*) *f-r* $\Longrightarrow$ *fun-of-rel* (*rel-xval-stack L R h*) (*map-xval f-l f-r*)
⟨*proof*⟩

lemma *funp-rel-sum-stack*[*funp-intros*, *corres-admissible*]: *funp* (*L h*) $\Longrightarrow$ *funp* (*R h*) $\Longrightarrow$ *funp* (*rel-sum-stack L R h*)
⟨*proof*⟩

lemma *funp-rel-xval-stack*[*funp-intros*, *corres-admissible*]: *funp* (*L h*) $\Longrightarrow$ *funp* (*R h*) $\Longrightarrow$ *funp* (*rel-xval-stack L R h*)
⟨*proof*⟩

lemma *rel-sum-stack-cases*:
rel-sum-stack L R h x y =
((∃ *v w*. *x* = *Inl v* ∧ *y* = *Inl w* ∧ *L h v w*) ∨
(∃ *v w*. *x* = *Inr v* ∧ *y* = *Inr w* ∧ *R h v w*))
⟨*proof*⟩

lemma *rel-xval-stack-cases*:
rel-xval-stack L R h x y =
((∃ *v w*. *x* = *Exn v* ∧ *y* = *Exn w* ∧ *L h v w*) ∨
(∃ *v w*. *x* = *Result v* ∧ *y* = *Result w* ∧ *R h v w*))
⟨*proof*⟩

lemma *rel-sum-stack-simps* [*simp*]:
rel-sum-stack L R h (*Inl* l_1) (*Inl* l_2) = *L h* l_1 l_2
rel-sum-stack L R h (*Inr* r_1) (*Inr* r_2) = *R h* r_1 r_2
rel-sum-stack L R h (*Inl* l_1) (*Inr* r_2) = *False*
rel-sum-stack L R h (*Inr* r_1) (*Inl* l_2) = *False*
rel-sum-stack L R h (*Inl* l_1) *y* = (∃ *w*. *y* = *Inl w* ∧ *L h* l_1 *w*)
rel-sum-stack L R h (*Inr* r_1) *y* = (∃ *w*. *y* = *Inr w* ∧ *R h* r_1 *w*)

rel-sum-stack L R h x (Inl l₂) = (∃ v. x = Inl v ∧ L h v l₂)
rel-sum-stack L R h x (Inr r₂) = (∃ v. x = Inr v ∧ R h v r₂)
⟨*proof*⟩

lemma *rel-xval-stack-simps* [*simp*]:
rel-xval-stack L R h (Exn l₁) (Exn l₂) = L h l₁ l₂
rel-xval-stack L R h (Result r₁) (Result r₂) = R h r₁ r₂
rel-xval-stack L R h (Exn l₁) (Result r₂) = False
rel-xval-stack L R h (Result r₁) (Exn l₂) = False
rel-xval-stack L R h (Exn l₁) y = (∃ w. y = Exn w ∧ L h l₁ w)
rel-xval-stack L R h (Result r₁) y = (∃ w. y = Result w ∧ R h r₁ w)
rel-xval-stack L R h x (Exn l₂) = (∃ v. x = Exn v ∧ L h v l₂)
rel-xval-stack L R h x (Result r₂) = (∃ v. x = Result v ∧ R h v r₂)
⟨*proof*⟩

lemma *rel-sum-stack-eq-collapse*: (*rel-sum-stack* (λ-. (=)) (λ-. (=))) = ((λ-. (=)))

⟨*proof*⟩

lemma *rel-xval-stack-eq-collapse*: (*rel-xval-stack* (λ-. (=)) (λ-. (=))) = ((λ-. (=)))

⟨*proof*⟩

lemma *rel-sum-stack-InlI*: *L h l1 l2* ⟹ *rel-sum-stack L R h* (*Inl l1*) (*Inl l2*)
⟨*proof*⟩

lemma *rel-xval-stack-ExnI*: *L h l1 l2* ⟹ *rel-xval-stack L R h* (*Exn l1*) (*Exn l2*)
⟨*proof*⟩

lemma *rel-sum-stack-InrI*: *R h r1 r2* ⟹ *rel-sum-stack L R h* (*Inr r1*) (*Inr r2*)
⟨*proof*⟩

lemma *rel-xval-stack-ResultI*: *R h r1 r2* ⟹ *rel-xval-stack L R h* (*Result r1*) (*Result r2*)
⟨*proof*⟩

definition *rel-exit Q h* = (λ*v w*. ∃ *x*. *v* = *Nonlocal x* ∧ *Q h x w*)

lemma *rel-exit-simps*[*simp*]:
rel-exit Q h (*Nonlocal x*) *y* = *Q h x y*
rel-exit Q h Break *y* = *False*
rel-exit Q h Continue *y* = *False*
rel-exit Q h Return *y* = *False*
rel-exit Q h (*Goto l*) *y* = *False*
⟨*proof*⟩

lemma *rel-exit-intro*: *Q h x y* ⟹ *rel-exit Q h* (*Nonlocal x*) *y*
⟨*proof*⟩

lemma *rel-exit-entail*[*rel-entail*]:
(⋀*x y*. *R h x y* ⟹ *R′ h x y*) ⟹
rel-exit R h x y ⟹ *rel-exit R′ h x y*
⟨*proof*⟩

lemma *rel-xval-stack-rel-exit-intro*:
assumes ⋀*x*. *rel-xval-stack* (*rel-exit Q*) *R h* (*Exn* (*Nonlocal* (*the-Nonlocal x*))) (*Exn x*)
assumes *rel-exit Q h e e′*
shows *rel-xval-stack* (*rel-exit Q*) *R h* (*Exn* (*Nonlocal* (*the-Nonlocal e*))) (*Exn e′*)
⟨*proof*⟩

lemma *rel-xval-stack-rel-exit-intro′*:
assumes ⋀*x*. *rel-xval-stack* (*rel-exit Q*) *R h* (*Exn* (*Nonlocal x*)) (*Exn x*)
assumes *Q h e e′*
shows *rel-xval-stack* (*rel-exit Q*) *R h* (*Exn* (*Nonlocal e*)) (*Exn e′*)
⟨*proof*⟩

lemma *rel-exit-False-conv* [*simp*]: *rel-exit* (λ- - -. *False*) *h e e′* ⟷ *False*
⟨*proof*⟩

lemma *rel-exit-FalseE*: *rel-exit* (λ- - -. *False*) *h e e′* ⟹ *L*
⟨*proof*⟩

lemma *rel-sum-stack-generalise-left*:
rel-sum-stack L R h v w ⟹ (⋀*v w*. *L h v w* ⟹ *L′ h v w*) ⟹ *rel-sum-stack L′ R h v w*
⟨*proof*⟩

lemma *rel-xval-stack-generalise-left*:
rel-xval-stack L R h v w ⟹ (⋀*v w*. *L h v w* ⟹ *L′ h v w*) ⟹ *rel-xval-stack L′ R h v w*
⟨*proof*⟩

lemma *rel-sum-stack-generalise-both*:
rel-sum-stack L R h v w ⟹ (⋀*v w*. *L h v w* ⟹ *L′ h v w*) ⟹ (⋀*v w*. *R h v w* ⟹ *R′ h v w*) ⟹
rel-sum-stack L′ R′ h v w
⟨*proof*⟩

lemmas *generalise-unreachable-exitE* =
rel-exit-FalseE
FalseE
rel-sum-stack-generalise-both

rel-xval-stack-generalise-left

lemma *fun-of-rel-rel-exit*: *fun-of-rel* (*L h*) *f-l* $\Longrightarrow$ *fun-of-rel* (*rel-exit L h*) (λv. *case v of Nonlocal x* $\Rightarrow$ *f-l x* | - $\Rightarrow$ *undefined*)
⟨*proof*⟩

lemma *funp-rel-exit* [*funp-intros*, *corres-admissible*]: *funp* (*L h*) $\Longrightarrow$ *funp* (*rel-exit L h*)
⟨*proof*⟩

named-theorems *equal-upto-simps*
named-theorems *refines-stack-intros*

lemma *equal-uptoI*:
assumes *eq*: $\bigwedge x.\ x \notin A \Longrightarrow f\ x = g\ x$
shows *equal-upto A f g*
⟨*proof*⟩

lemma *equal-upto-heap-update*[*equal-upto-simps*]:
fixes *p*:: *'a*::*mem-type ptr*
assumes (*ptr-span p*) $\subseteq A$
shows *equal-upto A* (*heap-update p v h1*) *h2* = *equal-upto A h1 h2*
⟨*proof*⟩

lemma *equal-upto-complement*: *equal-upto B f g* = *equal-on* (− *B*) *f g*
⟨*proof*⟩

lemma *equal-upto-update-left-equalize*:
equal-on (− *F*) (*f s*) *s* $\Longrightarrow$ *equal-on F* (*f s*) *t* $\Longrightarrow$ *equal-upto* (*F* $\cup$ *A*) *s t* = *equal-upto A* (*f s*) *t*
⟨*proof*⟩

lemma *admissible-const-snd*:
assumes *admissible-fst*: *ccpo.admissible* $\bigcap$ ($\lambda x\ y.\ y \subseteq x$) ($\lambda X.\ \exists w \in X.\ Q\ w$)
shows *ccpo.admissible* $\bigcap$ ($\lambda x\ y.\ y \subseteq x$) ($\lambda X.\ \exists w.\ (w, v) \in X \wedge Q\ w$)
⟨*proof*⟩

lemma *admissible-funp*: *funp Q* $\Longrightarrow$ *ccpo.admissible* $\bigcap$ ($\lambda x\ y.\ y \subseteq x$) ($\lambda X.\ \exists w \in X.\ Q\ v\ w$)
⟨*proof*⟩

lemma *admissible-funp-conj*: *funp Q* $\Longrightarrow$ *ccpo.admissible* $\bigcap$ ($\lambda x\ y.\ y \subseteq x$) ($\lambda X.\ \exists w \in X.\ Q\ v\ w \wedge P\ w$)
⟨*proof*⟩

lemma *override-on-frame-raw-frame-lemma1*:
assumes *disj-A-B*: $A \cap B = \{\}$
assumes *sf*: *stack-free d* $\cap B = \{\}$

shows
override-on d (override-on d′ d B) (A ∪ B − stack-free d) =
override-on d d′ (A − stack-free d)
⟨*proof*⟩

lemma *override-on-frame-raw-frame-lemma2*:
assumes *disj-A-B*: $A \cap B = \{\}$
assumes *sf*: *stack-free d* $\cap B = \{\}$
shows
override-on h′ (override-on h h′ B) (A ∪ B ∪ stack-free d) =
override-on h′ h (A ∪ stack-free d)
⟨*proof*⟩

lemma *disjoint-stack-free-equal-upto-trans*:
equal-upto A d′ d ⟹
$P \cap A = \{\} \Longrightarrow$ *stack-free d* $\cap P = \{\} \Longrightarrow$
stack-free d′ $\cap P = \{\}$
⟨*proof*⟩

lemma *disjoint-stack-free-equal-on-trans*:
equal-on S d′ d ⟹
$P \subseteq S \Longrightarrow$ *stack-free d* $\cap P = \{\} \Longrightarrow$
stack-free d′ $\cap P = \{\}$
⟨*proof*⟩

lemma *refines-L2-guard-right-unconditional*:
assumes *refines f g s t Q*
shows *refines f (L2-seq (L2-guard P) (λ-. g)) s t Q*
⟨*proof*⟩

lemma *refines-L2-guard-right-unconditional′*:
assumes *refines f g s t Q*
shows *refines f (L2-seq (L2-guard (λ-. P)) (λ-. g)) s t Q*
⟨*proof*⟩

lemma *refines-L2-guard-right′*:
assumes *P t ⟹ refines f g s t Q*
shows *refines f (L2-seq (L2-guard P) (λ-. g)) s t Q*
⟨*proof*⟩

lemma *refines-L2-guard-right″*:
assumes *P t ⟹ refines f (L2-seq (L2-guard P) (λ-. g)) s t Q*
shows *refines f (L2-seq (L2-guard P) (λ-. g)) s t Q*
⟨*proof*⟩

lemma *refines-L2-guard-right‴*:
assumes *P t ⟹ refines f (L2-seq (L2-seq (L2-guard P) (λ-. g)) X) s t Q*
shows *refines f (L2-seq (L2-seq (L2-guard P) (λ-. g)) X) s t Q*

⟨*proof*⟩

lemma *refines-L2-guard-right''''*:
assumes *P t* ⟹
refines f
(*L2-seq*
(*L2-catch* (*L2-seq* (*L2-guard P*) (λ-. *g*)) *X*)
Y) *s t Q*
shows *refines f*
(*L2-seq*
(*L2-catch* (*L2-seq* (*L2-guard P*) (λ-. *g*)) *X*)
Y) *s t Q*
⟨*proof*⟩

lemma *refines-L2-guard-rightE*:
assumes *P t*
assumes *refines f* (*L2-seq* (*L2-guard P*) (λ-. *g*)) *s t Q*
shows *refines f g s t Q*
⟨*proof*⟩

lemma *refines-L2-guard-rightE'*:
assumes *P t*
assumes *refines f* (*L2-seq* (*L2-guard P*) (λ-. *g*)) *s t Q*
shows *refines f* (*L2-seq* (*L2-guard P*) (λ-. *g*)) *s t Q*
⟨*proof*⟩

lemma *refines-L2-guard-right*:
(*P* ⟹ *refines f g s t Q*) ⟹ *refines f* (*L2-seq* (*L2-guard* (λ-. *P*)) (λ-. *g*)) *s t Q*
⟨*proof*⟩

context *heap-state*
begin

lemma *equal-upto-heap-on-heap-update*[*equal-upto-simps*]:
fixes *p*:: *'a*::*mem-type ptr*
assumes *ptr-span p* ⊆ *A*
shows *equal-upto-heap-on A* (*hmem-upd* (*heap-update p v*) *s*) *t* = *equal-upto-heap-on A s t*
⟨*proof*⟩

lemma *equal-upto-heap-stack-release'*:
fixes *p*:: *'a*::*mem-type ptr*
assumes *ptr-span p* ⊆ *A*
shows *equal-upto-heap-on A* (*htd-upd* (*stack-releases* (*Suc 0*) *p*) *s*) *s*
⟨*proof*⟩

lemma *equal-upto-heap-stack-release*[*equal-upto-simps*]:

fixes *p*:: *'a*::*mem-type ptr*
assumes *ptr-span p* ⊆ *A*
shows *equal-upto-heap-on A* (*htd-upd* (*stack-releases* (*Suc 0*) *p*) *s*) *t* = *equal-upto-heap-on A s t*
⟨*proof*⟩

lemma *equal-upto-heap-on-htd-upd*:
equal-upto-heap-on A s t ⟹
equal-upto A (*f* (*htd s*)) (*htd t*) ⟹
equal-upto-heap-on A (*htd-upd f s*) *t*
⟨*proof*⟩

lemma *equal-upto-heap-on-hmem*: *equal-upto-heap-on A s t* ⟹ *equal-upto A* (*hmem s*) (*hmem t*)
⟨*proof*⟩

lemma *equal-upto-heap-on-sym*: *equal-upto-heap-on A s t* = *equal-upto-heap-on A t s*
⟨*proof*⟩

lemma *equal-upto-heap-stack-alloc'*:
fixes *p*:: *'a*::*mem-type ptr*
shows *equal-upto-heap-on* (*ptr-span p*)
(*hmem-upd* (*heap-update p v*) (*htd-upd* (λ-. *ptr-force-type p* (*htd s*)) *s*))
s
⟨*proof*⟩

lemma *equal-upto-heap-stack-alloc*:
fixes *p*:: *'a*::*mem-type ptr*
assumes *length bs* = *size-of TYPE*(*'a*)
shows *equal-upto-heap-on* (*ptr-span p*)
(*hmem-upd* (*heap-update-padding p v bs*) (*htd-upd* (λ-. *ptr-force-type p* (*htd s*)) *s*))
s
⟨*proof*⟩

definition
rel-alloc:: *addr set* ⇒ *addr set* ⇒ *addr set* ⇒ *'s* ⇒ *'s* ⇒ *'s* ⇒ *bool*
where
rel-alloc S M A t$_0$ ≡ λ*s t*.
stack-free (*htd s*) ⊆ *S* ∧
stack-free (*htd s*) ∩ *A* = {} ∧
stack-free (*htd s*) ∩ *M* = {} ∧
t = *frame A t*$_0$ *s*

lemma *rel-alloc-fold-frame*: *rel-alloc* 𝒮 *M A t*$_0$ *s t* ⟹ *frame A t*$_0$ *s* = *t*
⟨*proof*⟩

lemma *rel-alloc-modifies-antimono*: *rel-alloc S M2 A t*$_0$ *s t* ⟹ *M1* ⊆ *M2* ⟹

rel-alloc S M1 A t_0 *s t*
⟨*proof*⟩

lemma *rel-alloc-stack-free-disjoint*:
rel-alloc S M A t_0 *s t* ⟹ *ptr-span p* ⊆ *A* ⟹ *ptr-span p* ∩ *stack-free* (*htd s*) = {}
⟨*proof*⟩

lemma *rel-alloc-stack-free-disjoint-trans*:
rel-alloc S M A t_0 *s′ t* ⟹ *htd s′* = *htd s* ⟹ *ptr-span p* ⊆ *A* ⟹ *ptr-span p* ∩ *stack-free* (*htd s*) = {}
⟨*proof*⟩

lemma *rel-alloc-stack-free-disjoint′*:
rel-alloc S M A t_0 *s t* ⟹ *ptr-span p* ⊆ *A* ⟹ *stack-free* (*htd s*) ∩ *ptr-span p* = {}
⟨*proof*⟩

lemma *rel-alloc-stack-free-disjoint-trans′*:
rel-alloc S M A t_0 *s′ t* ⟹ *htd s′* = *htd s* ⟹ *ptr-span p* ⊆ *A* ⟹ *stack-free* (*htd s*) ∩ *ptr-span p* = {}
⟨*proof*⟩

lemma *rel-alloc-disj-G-S*: *G* ∩ *S* = {} ⟹ *rel-alloc S M A* t_0 *s t* ⟹ *G* ∩ *stack-free* (*htd s*) = {}
⟨*proof*⟩

lemma *rel-alloc-stack-free-disjoint-field-lvalue*:
fixes *p*:: *′a*::*mem-type ptr*
assumes *rel-alloc S M A* t_0 *s t ptr-span p* ⊆ *A*
assumes *field-lookup* (*typ-info-t TYPE*(*′a*)) *f 0* = *Some* (*T*, *n*)
assumes *export-uinfo T* = *typ-uinfo-t* (*TYPE*(*′b*))
shows {&(*p*→*f*)..+*size-of TYPE*(*′b*::*c-type*)} ∩ *stack-free* (*htd s*) = {}
⟨*proof*⟩

lemma *rel-alloc-stack-free-disjoint-field-lvalue-trans*:
fixes *p*:: *′a*::*mem-type ptr*
assumes *rel-alloc S M A* t_0 *s′ t htd s′* = *htd s ptr-span p* ⊆ *A*
assumes *field-lookup* (*typ-info-t TYPE*(*′a*)) *f 0* = *Some* (*T*, *n*)
assumes *export-uinfo T* = *typ-uinfo-t* (*TYPE*(*′b*))
shows {&(*p*→*f*)..+*size-of TYPE*(*′b*::*c-type*)} ∩ *stack-free* (*htd s*) = {}
⟨*proof*⟩

lemma *rel-alloc-stack-free-disjoint-field-lvalue′*:
fixes *p*:: *′a*::*mem-type ptr*
assumes *rel-alloc S M A* t_0 *s t ptr-span p* ⊆ *A*
assumes *field-lookup* (*typ-info-t TYPE*(*′a*)) *f 0* = *Some* (*T*, *n*)
assumes *export-uinfo T* = *typ-uinfo-t* (*TYPE*(*′b*))
shows *stack-free* (*htd s*) ∩ {&(*p*→*f*)..+*size-of TYPE*(*′b*::*c-type*)} = {}

⟨*proof*⟩

lemma *rel-alloc-stack-free-disjoint-field-lvalue-trans'*:
fixes *p*:: *'a*::*mem-type ptr*
assumes *rel-alloc S M A t*$_0$ *s' t htd s' = htd s ptr-span p* ⊆ *A*
assumes *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0 = Some* (*T*, *n*)
assumes *export-uinfo T = typ-uinfo-t* (*TYPE*(*'b*))
shows *stack-free* (*htd s*) ∩ {&(*p*→*f*)..+*size-of TYPE*(*'b*::*c-type*)} = {}
⟨*proof*⟩

lemma *h-val-rel-alloc-disjoint*:
fixes *p*::*'a*::*mem-type ptr*
shows *rel-alloc S M A t*$_0$ *s t* ⟹ *ptr-span p* ∩ *A* = {} ⟹ *ptr-span p* ∩ *stack-free* (*htd s*) = {} ⟹ *h-val* (*hmem t*) *p* = *h-val* (*hmem s*) *p*
⟨*proof*⟩

definition *rel-stack*::
addr set ⇒ *addr set* ⇒ *addr set* ⇒ *'s* ⇒ *'s* ⇒ (*heap-mem* ⇒ *'b* ⇒ *'c* ⇒ *bool*) ⇒
(*'b* × *'s*) ⇒ (*'c* × *'s*) ⇒ *bool*
where
rel-stack S M A s t$_0$ *R* = (λ(*v*, *s'*) (*w*, *t'*).
R (*hmem s'*) *v w* ∧
rel-alloc S M A t$_0$ *s' t'* ∧
equal-upto (*M* ∪ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) ∧
htd s' = htd s)

lemma *rel-stack-unchanged-typing*: *rel-alloc S M' A t*$_0$ *s t* ⟹ *rel-stack S M A s t*$_0$ *R* (*v*, *s'*) (*w*, *t'*) ⟹
equal-on S (*htd t'*) (*htd t*)
⟨*proof*⟩

lemma *rel-stack-unchanged-heap*: *rel-alloc S M' A t*$_0$ *s t* ⟹ *rel-stack S M A s t*$_0$ *R* (*v*, *s'*) (*w*, *t'*) ⟹
equal-upto (*M* ∪ *stack-free* (*htd s'*)) (*hmem t'*) (*hmem t*)
⟨*proof*⟩

lemma *rel-stack-unchanged-stack-free*:
rel-alloc S M' A t$_0$ *s t* ⟹ *rel-stack S M A s t*$_0$ *R* (*v*, *s'*) (*w*, *t'*) ⟹
stack-free (*htd s'*) = *stack-free* (*htd s*)
⟨*proof*⟩

lemma *rel-stack-unchanged-stack-free'*:
assumes *stack*: *rel-alloc S M' A t*$_0$ *s t*
assumes *rel-stack*: *rel-stack S M A s t*$_0$ *R* (*v*, *s'*) (*w*, *t'*)
shows *stack-free* (*htd t'*) = *stack-free* (*htd t*)
⟨*proof*⟩

lemma *rel-stack-unchanged-heap'*:
assumes *alloc*: *rel-alloc S {} A t₀ s t*
assumes *stack*: *rel-stack S M A s t₀ R (v, s') (w, t')*
shows *equal-upto (M ∪ stack-free (htd t')) (hmem t') (hmem t)*
⟨*proof*⟩

lemma *rel-stack-Exn*[*simp*]:
rel-stack S M A s t₀ (rel-xval-stack L R) (Exn v, s') (Exn w, t') = rel-stack S M A s t₀ L (v, s') (w, t')
⟨*proof*⟩
lemma *rel-stack-Exn-Result*[*simp*]:
rel-stack S M A s t₀ (rel-xval-stack L R) (Exn v, s') (Result w, t') = False
⟨*proof*⟩
lemma *rel-stack-Result*[*simp*]:
rel-stack S M A s t₀ (rel-xval-stack L R) (Result v, s') (Result w, t') = rel-stack S M A s t₀ R (v, s') (w, t')
⟨*proof*⟩
lemma *rel-zero-stack-Result-Exn*[*simp*]:
rel-stack S M A s t₀ (rel-xval-stack L R) (Result v, s') (Exn w, t') = False
⟨*proof*⟩

lemma *admissible-fail*: *ccpo.admissible Inf (λx y. y ≤ x) (λA. ¬ succeeds A t)*
⟨*proof*⟩

lemma *fun-lub-lem*: *(λ(A::('f ⇒ (('d, 'e) exception-or-result × 'f) post-state) set) x::'f.*
⋂f::'f ⇒ (('d, 'e) exception-or-result × 'f) post-state∈A. f x) = fun-lub (Inf)
⟨*proof*⟩

lemma *fun-ord-lem*:
(λ(a::'f ⇒ (('d, 'e) exception-or-result × 'f) post-state)
b::'f ⇒ (('d, 'e) exception-or-result × 'f) post-state. ∀ x::'f. b x ≤ a x) = fun-ord (≥)
⟨*proof*⟩

lemma *admissible-refines-funp*:
assumes ∗: *funp R*
shows *ccpo.admissible Inf (≥) (λA. refines C A s t R)*
⟨*proof*⟩

lemma *fun-of-rel-stack*:
assumes *f*: *⋀h. fun-of-rel (Q h) (f h)*
shows *fun-of-rel (rel-stack S M A s t₀ Q) (λ(r, s). (f (hmem s) r, frame A t₀ s))*
⟨*proof*⟩

lemma *funp-rel-stack*:
assumes *funp*: $\bigwedge h.\ funp\ (Q\ h)$
shows *funp* (*rel-stack* $S\ M\ A\ s\ t_0\ Q$)
⟨*proof*⟩

theorem *admissible-refines-rel-stack*[*corres-admissible*]:
assumes *funp*: $\bigwedge h.\ funp\ (Q\ h)$
shows *ccpo.admissible Inf* $(\geq)$ $(\lambda g.\ refines\ f\ g\ s'\ t'\ (rel\text{-}stack\ S\ M\ A\ s\ t_0\ Q))$
⟨*proof*⟩

lemma *admissible-rel-stack-eq*: *ccpo.admissible* $\bigcap$ $(\lambda x\ y.\ y \subseteq x)$ $(\lambda X.\ \exists w \in X.\ (\lambda\text{-}.\ (=))\ h\ v\ w)$
⟨*proof*⟩

lemma *admissible-rel-singleton-stack*:
shows *ccpo.admissible* $\bigcap$ $(\lambda x\ y.\ y \subseteq x)$ $(\lambda X.\ \exists w \in X.\ (rel\text{-}singleton\text{-}stack\ p)\ h\ v\ w)$
⟨*proof*⟩

lemma *admissible-rel-push*:
assumes *admiss*: $\bigwedge h\ v.$ *ccpo.admissible* $\bigcap$ $(\lambda x\ y.\ y \subseteq x)$ $(\lambda X.\ \exists w \in X.\ Q\ h\ v\ w)$
shows *ccpo.admissible* $\bigcap$ $(\lambda x\ y.\ y \subseteq x)$ $(\lambda X.\ \exists w \in X.\ (rel\text{-}push\ p\ Q)\ h\ v\ w)$
⟨*proof*⟩

lemma *admissible-rel-sum-stack*:
assumes *admiss-L*: $\bigwedge h\ v.$ *ccpo.admissible* $\bigcap$ $(\lambda x\ y.\ y \subseteq x)$ $(\lambda X.\ \exists w \in X.\ L\ h\ v\ w)$
assumes *admiss-R*: $\bigwedge h\ v.$ *ccpo.admissible* $\bigcap$ $(\lambda x\ y.\ y \subseteq x)$ $(\lambda X.\ \exists w \in X.\ R\ h\ v\ w)$
shows *ccpo.admissible* $\bigcap$ $(\lambda x\ y.\ y \subseteq x)$ $(\lambda X.\ \exists w \in X.\ (rel\text{-}sum\text{-}stack\ L\ R)\ h\ v\ w)$
⟨*proof*⟩

lemma *IOcorres-refines-conv*:
assumes *rel-to-prj*: $\bigwedge h.$ *fun-of-rel* $(R\ h)$ $(prj\ h)$
assumes *rel-to-prjE*: $\bigwedge h.$ *fun-of-rel* $(L\ h)$ $(prjE\ h)$
shows *IOcorres*
$(\lambda s.\ rel\text{-}alloc\ \mathcal{S}\ M\ A\ t_0\ s\ (frame\ A\ t_0\ s) \wedge P\ s)$
$(\lambda s\ r\ s'.\ stack\text{-}free\ (htd\ s') \subseteq \mathcal{S} \wedge stack\text{-}free\ (htd\ s') \cap A = \{\} \wedge stack\text{-}free\ (htd\ s') \cap M1 = \{\} \wedge$
equal-upto $(M1 \cup stack\text{-}free\ (htd\ s'))$ $(hmem\ s')$ $(hmem\ s) \wedge$
$htd\ s' = htd\ s \wedge$
$(case\ r\ of\ Exn\ l \Rightarrow \exists e.\ l = Nonlocal\ e \wedge L\ (hmem\ s')\ e\ (prjE\ (hmem\ s')\ e) \mid Result\ x \Rightarrow R\ (hmem\ s')\ x\ (prj\ (hmem\ s')\ x))$
$(frame\ A\ t_0)$
$(\lambda r\ s.\ prj\ (hmem\ s)\ r)$
$(\lambda r\ s.\ prjE\ (hmem\ s)\ (the\text{-}Nonlocal\ r))$

$g\ f$
$\longleftrightarrow$
$(\forall s\ t.\ \mathit{rel\text{-}alloc}\ \mathcal{S}\ M\ A\ t_0\ s\ t \longrightarrow P\ s \longrightarrow \mathit{refines}\ f\ g\ s\ t\ (\mathit{rel\text{-}stack}\ \mathcal{S}\ \mathit{M1}\ A\ s\ t_0\ (\mathit{rel\text{-}xval\text{-}stack}\ (\mathit{rel\text{-}exit}\ L)\ R)))$
⟨*proof*⟩

lemmas *IOcorres-to-refines* = *iffD1* [*OF* *IOcorres-refines-conv*, *rule-format*]
lemmas *refines-to-IOcorres* = *iffD2* [*OF* *IOcorres-refines-conv*, *rule-format*]

lemma *refines-rel-xval-stack-generalise-exit*:
$\mathit{refines}\ f\ g\ s\ t\ (\mathit{rel\text{-}stack}\ \mathcal{S}\ M\ A\ s\ t_0\ (\mathit{rel\text{-}xval\text{-}stack}\ L\ R)) \Longrightarrow (\bigwedge h\ v\ w.\ L\ h\ v\ w \Longrightarrow L'\ h\ v\ w) \Longrightarrow$
$\mathit{refines}\ f\ g\ s\ t\ (\mathit{rel\text{-}stack}\ \mathcal{S}\ M\ A\ s\ t_0\ (\mathit{rel\text{-}xval\text{-}stack}\ L'\ R))$
⟨*proof*⟩

lemma *L2-gets-rel-stack′*:
assumes $R\ (\mathit{hmem}\ s)\ (e\ s)\ (e'\ (\mathit{frame}\ A\ t_0\ s))$
assumes $\mathit{rel\text{-}alloc}\ \mathcal{S}\ M\ A\ t_0\ s\ t$
shows $\mathit{refines}\ (\mathit{L2\text{-}gets}\ e\ ns)\ (\mathit{L2\text{-}gets}\ e'\ ns)\ s\ t\ (\mathit{rel\text{-}stack}\ \mathcal{S}\ \{\}\ A\ s\ t_0\ (\mathit{rel\text{-}xval\text{-}stack}\ L\ R))$
⟨*proof*⟩

lemma *L2-gets-rel-stack*:
assumes $R\ (\mathit{hmem}\ s)\ (e\ s)\ (e'\ (\mathit{frame}\ A\ t_0\ s))$
assumes $\mathit{rel\text{-}alloc}\ \mathcal{S}\ M\ A\ t_0\ s\ t$
shows $\mathit{refines}\ (\mathit{L2\text{-}gets}\ e\ ns)\ (\mathit{L2\text{-}gets}\ e'\ ns')\ s\ t\ (\mathit{rel\text{-}stack}\ \mathcal{S}\ \{\}\ A\ s\ t_0\ (\mathit{rel\text{-}xval\text{-}stack}\ L\ R))$
⟨*proof*⟩

lemma *L2-gets-rel-stack-guarded*:
assumes $G \Longrightarrow R\ (\mathit{hmem}\ s)\ (e\ s)\ (e'\ (\mathit{frame}\ A\ t_0\ s))$
assumes $\mathit{rel\text{-}alloc}\ \mathcal{S}\ M\ A\ t_0\ s\ t$
shows $\mathit{refines}\ (\mathit{L2\text{-}gets}\ e\ ns)\ (\mathit{L2\text{-}seq}\ (\mathit{L2\text{-}guard}\ (\lambda\text{-}.\ G))\ (\lambda\text{-}.\ (\mathit{L2\text{-}gets}\ e'\ ns')))\ s\ t\ (\mathit{rel\text{-}stack}\ \mathcal{S}\ \{\}\ A\ s\ t_0\ (\mathit{rel\text{-}xval\text{-}stack}\ L\ R))$
⟨*proof*⟩

lemma *L2-gets-constant-trivial-rel-stack*:
assumes $\mathit{rel\text{-}alloc}\ \mathcal{S}\ M\ A\ t_0\ s\ t$
shows $\mathit{refines}\ (\mathit{L2\text{-}gets}\ (\lambda\text{-}.\ c)\ ns)\ (\mathit{L2\text{-}gets}\ (\lambda\text{-}.\ c)\ ns)\ s\ t\ (\mathit{rel\text{-}stack}\ \mathcal{S}\ \{\}\ A\ s\ t_0\ (\mathit{rel\text{-}xval\text{-}stack}\ L\ (\lambda\text{-}.\ (=))))$
⟨*proof*⟩

lemma *L2-gets-constant-rel-stack*:
assumes $R\ (\mathit{hmem}\ s)\ c\ c'$
assumes $\mathit{rel\text{-}alloc}\ \mathcal{S}\ M\ A\ t_0\ s\ t$
shows $\mathit{refines}\ (\mathit{L2\text{-}gets}\ (\lambda\text{-}.\ c)\ ns)\ (\mathit{L2\text{-}gets}\ (\lambda\text{-}.\ c')\ ns)\ s\ t\ (\mathit{rel\text{-}stack}\ \mathcal{S}\ \{\}\ A\ s\ t_0\ (\mathit{rel\text{-}xval\text{-}stack}\ L\ R))$
⟨*proof*⟩

lemma *L2-throw-rel-stack*:
assumes L (*hmem* s) c c'
assumes *rel-alloc* $\mathcal{S}$ M A t_0 s t
shows *refines* (*L2-throw* c *ns*) (*L2-throw* c' *ns'*) s t (*rel-stack* $\mathcal{S}$ {} A s t_0 (*rel-xval-stack* L R))
⟨*proof*⟩

lemma *L2-try-rel-stack*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ M A t_0 s t
assumes *refines* f g s t (*rel-stack* $\mathcal{S}$ *M1* A s t_0 (*rel-xval-stack* (*rel-sum-stack* L R) R))
shows *refines* (*L2-try* f) (*L2-try* g) s t (*rel-stack* $\mathcal{S}$ *M1* A s t_0 (*rel-xval-stack* L R))
⟨*proof*⟩

lemma *L2-try-rel-stack-merge1*:
assumes $\bigwedge h\ v\ v'.\ R'\ h\ v\ v' \Longrightarrow\ R\ h\ v\ v'$
assumes *s-t*: *rel-alloc* $\mathcal{S}$ M A t_0 s t
assumes *refines* f g s t (*rel-stack* $\mathcal{S}$ *M1* A s t_0 (*rel-xval-stack* (*rel-sum-stack* L R') R))
shows *refines* (*L2-try* f) (*L2-try* g) s t (*rel-stack* $\mathcal{S}$ *M1* A s t_0 (*rel-xval-stack* L R))
⟨*proof*⟩

lemma *L2-try-rel-stack-merge2*:
assumes $\bigwedge h\ v\ v'.\ R'\ h\ v\ v' \Longrightarrow R\ h\ v\ v'$
assumes *s-t*: *rel-alloc* $\mathcal{S}$ M A t_0 s t
assumes *refines* f g s t (*rel-stack* $\mathcal{S}$ *M1* A s t_0 (*rel-xval-stack* (*rel-sum-stack* L R) R'))
shows *refines* (*L2-try* f) (*L2-try* g) s t (*rel-stack* $\mathcal{S}$ *M1* A s t_0 (*rel-xval-stack* L R))
⟨*proof*⟩

lemma *L2-guard-rel-stack*:
assumes e' (*frame* A t_0 s) $\Longrightarrow$ e s
assumes *rel-alloc* $\mathcal{S}$ M A t_0 s t
shows *refines* (*L2-guard* e) (*L2-guard* e') s t (*rel-stack* $\mathcal{S}$ {} A s t_0 (*rel-xval-stack* L (λ-. (=))))
⟨*proof*⟩

lemma *L2-modify-heap-update-rel-stack'*:
fixes p::'a:: *mem-type ptr*
assumes R (*heap-update* p v (*hmem* s)) () (e' (*frame* A t_0 s))
assumes *ptr-span* $p \subseteq A$
assumes *rel-alloc* $\mathcal{S}$ (*ptr-span* p) A t_0 s t
shows *refines*
(*L2-modify* (*hmem-upd* (*heap-update* p v)))
(*L2-gets* e' *ns*)

$s\ t$
($rel\text{-}stack\ \mathcal{S}\ (ptr\text{-}span\ p)\ A\ s\ t_0\ (rel\text{-}xval\text{-}stack\ L\ R)$)
⟨*proof*⟩

lemma *L2-modify-heap-update-rel-stack*:
fixes $p::'a::$ *mem-type ptr*
assumes $R\ (heap\text{-}update\ p\ (v\ s)\ (hmem\ s))\ ()\ (e'\ (frame\ A\ t_0\ s))$
assumes $rel\text{-}alloc\ \mathcal{S}\ M\ A\ t_0\ s\ t$
assumes $ptr\text{-}span\ p \subseteq A$
assumes $ptr\text{-}span\ p \subseteq M$
shows *refines*
($L2\text{-}modify\ (\lambda s.\ hmem\text{-}upd\ (heap\text{-}update\ p\ (v\ s))\ s)$)
($L2\text{-}gets\ e'\ ns$)
$s\ t$
($rel\text{-}stack\ \mathcal{S}\ (ptr\text{-}span\ p)\ A\ s\ t_0\ (rel\text{-}xval\text{-}stack\ L\ R)$)
⟨*proof*⟩

lemma *L2-modify-keep-heap-update-rel-stack*:
fixes $p::'a::$ *mem-type ptr*
assumes $v\ s = v'\ (frame\ A\ t_0\ s)$
assumes $rel\text{-}alloc\ \mathcal{S}\ M\ A\ t_0\ s\ t$
assumes $ptr\text{-}span\ p \cap A = \{\}$
assumes $ptr\text{-}span\ p \cap stack\text{-}free\ (htd\ s) = \{\}$
assumes $ptr\text{-}span\ p \subseteq M$
shows *refines*
($L2\text{-}modify\ (\lambda s.\ hmem\text{-}upd\ (heap\text{-}update\ p\ (v\ s))\ s)$)
($L2\text{-}modify\ (\lambda s.\ hmem\text{-}upd\ (heap\text{-}update\ p\ (v'\ s))\ s)$)
$s\ t$
($rel\text{-}stack\ \mathcal{S}\ (ptr\text{-}span\ p)\ A\ s\ t_0\ (rel\text{-}xval\text{-}stack\ L\ (\lambda\text{-}.\ (=))))$
⟨*proof*⟩

lemma *L2-modify-keep-heap-update-rel-stack-guarded*:
fixes $p::'a::$ *mem-type ptr*
assumes $G \Longrightarrow v\ s = v'\ (frame\ A\ t_0\ s)$
assumes $rel\text{-}alloc\ \mathcal{S}\ M\ A\ t_0\ s\ t$
assumes $G \Longrightarrow ptr\text{-}span\ p \cap A = \{\}$
assumes $G \Longrightarrow ptr\text{-}span\ p \cap stack\text{-}free\ (htd\ s) = \{\}$
assumes $G \Longrightarrow ptr\text{-}span\ p \subseteq M$
shows *refines*
($L2\text{-}modify\ (\lambda s.\ hmem\text{-}upd\ (heap\text{-}update\ p\ (v\ s))\ s)$)
($L2\text{-}seq\ (L2\text{-}guard\ (\lambda\text{-}.\ G))\ (\lambda\text{-}.\ (L2\text{-}modify\ (\lambda s.\ hmem\text{-}upd\ (heap\text{-}update$
$p\ (v'\ s))\ s))))$
$s\ t$
($rel\text{-}stack\ \mathcal{S}\ (ptr\text{-}span\ p)\ A\ s\ t_0\ (rel\text{-}xval\text{-}stack\ L\ (\lambda\text{-}.\ (=))))$
⟨*proof*⟩

lemma *L2-call-rel-stack'*:
assumes $rel\text{-}alloc\ \mathcal{S}\ M\ A\ t_0\ s\ t$

assumes L': $\bigwedge e\ e'\ s.\ L\ (hmem\ s)\ e\ e' \Longrightarrow L'\ (hmem\ s)\ (emb\ e)\ (emb'\ e')$
assumes f: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
shows *refines*
(*L2-call f emb ns*)
(*L2-call g emb' ns'*)
s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L' R*))
⟨*proof*⟩

lemma *L2-call-rel-stack''*:
assumes *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes L': $\bigwedge e\ e'\ s.\ L\ (hmem\ s)\ e\ e' \Longrightarrow$ *rel-sum-stack L R' (hmem s) (emb e) (emb' e')*
assumes f: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
shows *refines*
(*L2-call f emb ns*)
(*L2-call g emb' ns'*)
s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-sum-stack L R'*) *R*))
⟨*proof*⟩

lemma *L2-call-rel-stack*:
assumes *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes L': $\bigwedge e\ e'\ s.\ L\ (hmem\ s)\ e\ e' \Longrightarrow L'\ (hmem\ s)\ (emb\ e)\ (emb\ e')$
assumes f: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
shows *refines*
(*L2-call f emb ns*)
(*L2-call g emb ns'*)
s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L' R*))
⟨*proof*⟩

Currently exceptions on the function level are terminal and are propagated to the toplevel. So the result relation for *Inl* is equality.

lemma *L2-call-rel-stack-eq-InL*:
assumes *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes f: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (λ-. (=)) *R*))
shows *refines*
(*L2-call f emb ns*)
(*L2-call g emb ns'*)
s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (λ-. (=)) *R*))
⟨*proof*⟩

lemma *L2-call-rel-stack-bot-InL*:
assumes *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (λ- - -. *False*) *R*))
shows *refines*
(*L2-call f emb ns*)
(*L2-call g emb ns'*)
s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (λ- - -. *False*) *R*))
⟨*proof*⟩

lemma *L2-seq-rel-stack''*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f1-g1*: *refines f1 g1 s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R1*))
assumes *f2-g2*: $\bigwedge$*s' t' v w. R1* (*hmem s'*) *v w* ⟹ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* ⟹
equal-upto (*M1* ∪ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) ⟹
htd s' = *htd s* ⟹

refines (*f2 v*) (*g2' s' w*) *s' t'* (*rel-stack* $\mathcal{S}$ *M2 A s'* t_0 (*rel-xval-stack L R*))
assumes *g2'-g2*: $\bigwedge$*s' t' v w. R1* (*hmem s'*) *v w* ⟹ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* ⟹
equal-upto (*M1* ∪ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) ⟹
htd s' = *htd s* ⟹
g2' s' w = *g2 w*

assumes *M1*: *M1* ⊆ *M*
assumes *M2*: *M2* ⊆ *M*
shows *refines* (*L2-seq f1 f2*) (*L2-seq g1 g2*) *s t* (*rel-stack* $\mathcal{S}$ (*M1* ∪ *M2*) *A s* t_0 (*rel-xval-stack L R*))
⟨*proof*⟩

lemma *L2-seq-rel-stack*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f1-g1*: *refines f1 g1 s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R1*))
assumes *f2-g2*: $\bigwedge$*s' t' v w. R1* (*hmem s'*) *v w* ⟹ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* ⟹
equal-upto (*M1* ∪ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) ⟹
htd s' = *htd s* ⟹

refines (*f2 v*) (*g2' w s'*) *s' t'* (*rel-stack* $\mathcal{S}$ *M2 A s'* t_0 (*rel-xval-stack L R*))
assumes *g2'-g2*: $\bigwedge$*s' t' v w. R1* (*hmem s'*) *v w* ⟹ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* ⟹
equal-upto (*M1* ∪ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) ⟹
htd s' = *htd s* ⟹
g2' w s' = *g2 w*

assumes *M1*: *M1* ⊆ *M*
assumes *M2*: *M2* ⊆ *M*
assumes *M'*: *M'* = *M1* ∪ *M2*
shows *refines* (*L2-seq f1 f2*) (*L2-seq g1 g2*) *s t* (*rel-stack* $\mathcal{S}$ *M' A s* t_0 (*rel-xval-stack L R*))

⟨*proof*⟩

lemma *L2-seq-rel-stack-g2-normalised*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f1-g1*: *refines f1 g1 s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R1*))
assumes *f2-g2*: $\bigwedge s'\ t'\ v\ w.$ *R1* (*hmem s'*) *v w* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* $\Longrightarrow$
equal-upto (*M1* $\cup$ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) $\Longrightarrow$
htd s' = *htd s* $\Longrightarrow$
refines (*f2 v*) (*g2 w*) *s' t'* (*rel-stack* $\mathcal{S}$ *M2 A s'* t_0 (*rel-xval-stack L R*))
assumes *M1*: *M1* $\subseteq$ *M*
assumes *M2*: *M2* $\subseteq$ *M*
assumes *M'*: *M'* = *M1* $\cup$ *M2*
shows *refines* (*L2-seq f1 f2*) (*L2-seq g1 g2*) *s t* (*rel-stack* $\mathcal{S}$ *M' A s* t_0 (*rel-xval-stack L R*))
⟨*proof*⟩

lemma *L2-seq-rel-stack'*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f1-g1*: *refines f1 g1 s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R1*))
assumes *f2-g2*: $\bigwedge s'\ t'\ v\ w.$ *R1* (*hmem s'*) *v w* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* $\Longrightarrow$
refines (*f2 v*) (*g2 w*) *s' t'* (*rel-stack* $\mathcal{S}$ *M2 A s'* t_0 (*rel-xval-stack L R*))
assumes *M1*: *M1* $\subseteq$ *M*
assumes *M2*: *M2* $\subseteq$ *M*
assumes *M'*: *M'* = *M1* $\cup$ *M2*
shows *refines* (*L2-seq f1 f2*) (*L2-seq g1 g2*) *s t* (*rel-stack* $\mathcal{S}$ *M' A s* t_0 (*rel-xval-stack L R*))
⟨*proof*⟩

lemma *frame-raw-frame-union*:
assumes *disj-A-B*: *A* $\cap$ *B* = {}
assumes *sf*: *stack-free* (*htd s*) $\cap$ *B* = {}
shows *frame* (*A* $\cup$ *B*) (*raw-frame B s* t_0) *s* = *frame A* t_0 *s*
⟨*proof*⟩

lemma *refines-narrow-frame'*:
assumes *disj-B-M*: *B* $\cap$ *M* = {}
assumes *disj-A-B*: *A* $\cap$ *B* = {}
assumes *spec*: $\bigwedge t_0\ s\ t.$ *rel-alloc* $\mathcal{S}$ *M1* (*A* $\cup$ *B*) t_0 *s t* $\Longrightarrow$
refines f g s t (*rel-stack* $\mathcal{S}$ *M* (*A* $\cup$ *B*) *s* t_0 *Q*)
assumes *sf*: *stack-free* (*htd s*) $\cap$ *B* = {}
assumes *rel-alloc*: *rel-alloc* $\mathcal{S}$ *M1 A* t_0 *s t*
shows *refines f g s t* (*rel-stack* $\mathcal{S}$ *M A s* t_0 *Q*)
⟨*proof*⟩

lemma *refines-narrow-frame*:
assumes *subset*: *B* $\subseteq$ *A*

assumes *disj-B-M*: $B \cap M = \{\}$
assumes *spec*: $\bigwedge t_0\ s\ t.$ *rel-alloc* $\mathcal{S}$ *M1* $A\ t_0\ s\ t \Longrightarrow$ *refines* $f\ g\ s\ t$ (*rel-stack* $\mathcal{S}$ *M* $A\ s\ t_0\ Q$)
assumes *sf*: *stack-free* (*htd* s) $\cap B = \{\}$
assumes *rel-alloc*: *rel-alloc* $\mathcal{S}$ *M1* $(A - B)\ t_0\ s\ t$
shows *refines* $f\ g\ s\ t$ (*rel-stack* $\mathcal{S}$ *M* $(A - B)\ s\ t_0\ Q$)
⟨*proof*⟩

lemma *refines-widen-modifies*:
assumes *rel-alloc* $S\ M\ A\ t_0\ s\ t$
assumes *refines* $f\ g\ s\ t$ (*rel-stack* $S\ M'\ A\ s\ t_0\ Q$)
assumes $M' \subseteq M$
shows *refines* $f\ g\ s\ t$ (*rel-stack* $S\ M\ A\ s\ t_0\ Q$)
⟨*proof*⟩

lemma *refines-widen-modifies'*:
assumes *rel-alloc* $S\ M\ A\ t_0\ s\ t \Longrightarrow$ *refines* $f\ g\ s\ t$ (*rel-stack* $S\ M'\ A\ s\ t_0\ Q$)
assumes $M' \subseteq M$
assumes *rel-alloc* $S\ M\ A\ t_0\ s\ t$
shows *refines* $f\ g\ s\ t$ (*rel-stack* $S\ M\ A\ s\ t_0\ Q$)
⟨*proof*⟩

lemma *refines-widen-modifies''*:
assumes *rel-alloc* $S\ M\ A\ t_0\ s\ t$
assumes *refines* $f\ g\ s\ t$ (*rel-stack* S *M1* $A\ s\ t_0\ Q$)
assumes *M1* $\subseteq$ *M2*
assumes *M2* $\subseteq M$
shows *refines* $f\ g\ s\ t$ (*rel-stack* S *M2* $A\ s\ t_0\ Q$)
⟨*proof*⟩

lemma *refines-widen-modifies-weaken*:
assumes *alloc*: *rel-alloc* $S\ M\ A\ t_0\ s\ t$
assumes *f*: *refines* $f\ g\ s\ t$ (*rel-stack* S *M1* $A\ s\ t_0\ Q'$)
assumes *M1-M2*: *M1* $\subseteq$ *M2*
assumes *M2-M*: *M2* $\subseteq M$
assumes *weaken*: $\bigwedge h\ r\ r'.\ Q'\ h\ r\ r' \Longrightarrow Q\ h\ r\ r'$
shows *refines* $f\ g\ s\ t$ (*rel-stack* S *M2* $A\ s\ t_0\ Q$)
⟨*proof*⟩

lemma *L2-guarded-rel-stack*:
assumes *e*: g' (*frame* $A\ t_0\ s$) $\Longrightarrow g\ s$
assumes *rel-alloc*: *rel-alloc* $\mathcal{S}$ *M* $A\ t_0\ s\ t$
assumes *X-X'*: $g\ s \Longrightarrow g'\ t \Longrightarrow$ *refines* $X\ X'\ s\ t$ (*rel-stack* $\mathcal{S}\ M'\ A\ s\ t_0$ (*rel-xval-stack* $L\ R$))
assumes *M*: $M' \subseteq M$
shows *refines* (*L2-guarded* $g\ X$) (*L2-guarded* $g'\ X'$) $s\ t$ (*rel-stack* $\mathcal{S}\ M'\ A\ s\ t_0$ (*rel-xval-stack* $L\ R$))
⟨*proof*⟩

lemma *L2-condition-rel-stack*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *c-c'*: *c s* = *c'* (*frame A* t_0 *s*)
assumes *f1-g1*: *refines f1 g1 s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
assumes *f2-g2*: *refines f2 g2 s t* (*rel-stack* $\mathcal{S}$ *M2 A s* t_0 (*rel-xval-stack L R*))
assumes *M1*: *M1* ⊆ *M*
assumes *M2*: *M2* ⊆ *M*
assumes *M'*: *M'* = *M1* ∪ *M2*
shows *refines* (*L2-condition c f1 f2*) (*L2-condition c' g1 g2*) *s t* (*rel-stack* $\mathcal{S}$ *M'* *A s* t_0 (*rel-xval-stack L R*))
⟨*proof*⟩

lemma *L2-while-rel-stack''*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *R-i*: *R* (*hmem s*) *i i'*
assumes *refines-condition*: ⋀*s v v'*. *R* (*hmem s*) *v v'* ⟹ *c' v'* (*frame A* t_0 *s*) = *c v s*
assumes *bdy*: ⋀*s t v v'*. *R* (*hmem s*) *v v'* ⟹ *rel-alloc* $\mathcal{S}$ *M1 A* t_0 *s t* ⟹ *c v s* ⟹ *c' v' t* ⟹
refines (*f v*) (*g v'*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
assumes *M1*: *M1* ⊆ *M*
shows *refines* (*L2-while c f i ns*) (*L2-while c' g i' ns'*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
⟨*proof*⟩

lemma *L2-while-rel-stack'''*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *R-i*: *R* (*hmem s*) *i i'*
assumes *refines-condition*: ⋀*s v v'*. *R* (*hmem s*) *v v'* ⟹ *c' v'* (*frame A* t_0 *s*) = *c v s*
assumes *bdy*: ⋀*s t v v'*. *R* (*hmem s*) *v v'* ⟹ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t* ⟹ *c v s* ⟹ *c' v' t* ⟹
refines (*f v*) (*g v'*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
assumes *M1*: *M1* ⊆ *M*
shows *refines* (*L2-while c f i ns*) (*L2-while c' g i' ns'*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
⟨*proof*⟩

lemma *L2-while-rel-stack'*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *refines-condition*: ⋀*s v v'*. *R* (*hmem s*) *v v'* ⟹ *c' v'* (*frame A* t_0 *s*) = *c v s*
assumes *R-i*: *R* (*hmem s*) *i i'*
assumes *bdy*: ⋀*s t v v'*. *R* (*hmem s*) *v v'* ⟹ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t* ⟹
refines (*f v*) (*g v'*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
assumes *M1*: *M1* ⊆ *M*
shows *refines* (*L2-while c f i ns*) (*L2-while c' g i' ns'*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0

(*rel-xval-stack L R*))
⟨*proof*⟩

lemma *L2-while-rel-stack*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *R-i*: *R* (*hmem s*) *i i'*
assumes *bdy*: $\bigwedge$*s' t' v w*. *R* (*hmem s'*) *v w* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* $\Longrightarrow$ *c v s'* $\Longrightarrow$ *c' w t'* $\Longrightarrow$
equal-upto (*M1* ∪ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) $\Longrightarrow$
htd s' = *htd s* $\Longrightarrow$
refines (*f v*) (*g' w s'*) *s' t'* (*rel-stack* $\mathcal{S}$ *M1 A s'* t_0 (*rel-xval-stack L R*))
assumes *refines-condition*: $\bigwedge$*s' t' v w*. *R* (*hmem s'*) *v w* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* $\Longrightarrow$
equal-upto (*M1* ∪ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) $\Longrightarrow$ *htd s'* = *htd s* $\Longrightarrow$
c' w t' = *c v s'*
assumes *g'-g*: $\bigwedge$*s' t' v w*. *R* (*hmem s'*) *v w* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* $\Longrightarrow$
equal-upto (*M1* ∪ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) $\Longrightarrow$
htd s' = *htd s* $\Longrightarrow$
g' w s' = *g w*
assumes *M1*: *M1* ⊆ *M*
shows *refines* (*L2-while c f i ns*) (*L2-while c' g i' ns'*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
⟨*proof*⟩

lemma *L2-while-rel-stack-g-normalised*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *R-i*: *R* (*hmem s*) *i i'*
assumes *bdy*: $\bigwedge$*s' t' v w*. *R* (*hmem s'*) *v w* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* $\Longrightarrow$ *c v s'* $\Longrightarrow$ *c' w t'* $\Longrightarrow$
equal-upto (*M1* ∪ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) $\Longrightarrow$
htd s' = *htd s* $\Longrightarrow$
refines (*f v*) (*g w*) *s' t'* (*rel-stack* $\mathcal{S}$ *M1 A s'* t_0 (*rel-xval-stack L R*))
assumes *refines-condition*: $\bigwedge$*s' t' v w*. *R* (*hmem s'*) *v w* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* $\Longrightarrow$
equal-upto (*M1* ∪ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) $\Longrightarrow$ *htd s'* = *htd s* $\Longrightarrow$
c' w t' = *c v s'*
assumes *M1*: *M1* ⊆ *M*
shows *refines* (*L2-while c f i ns*) (*L2-while c' g i' ns'*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
⟨*proof*⟩

lemma *L2-while-rel-stack-g-normalised-guarded*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *R-i*: *R* (*hmem s*) *i i'*

assumes *bdy*: $\bigwedge s'\ t'\ v\ w.\ R\ (hmem\ s')\ v\ w \Longrightarrow rel\text{-}alloc\ \mathcal{S}\ M\ A\ t_0\ s'\ t' \Longrightarrow c\ v\ s' \Longrightarrow c'\ w\ t' \Longrightarrow$
equal-upto $(M1 \cup stack\text{-}free\ (htd\ s'))\ (hmem\ s')\ (hmem\ s) \Longrightarrow$
$htd\ s' = htd\ s \Longrightarrow$
refines $(f\ v)\ (g\ w)\ s'\ t'\ (rel\text{-}stack\ \mathcal{S}\ M1\ A\ s'\ t_0\ (rel\text{-}xval\text{-}stack\ L\ R))$
assumes *refines-condition*: $\bigwedge s'\ t'\ \ v\ w.\ R\ (hmem\ s')\ v\ w \Longrightarrow rel\text{-}alloc\ \mathcal{S}\ M\ A\ t_0\ s'\ t' \Longrightarrow$
equal-upto $(M1 \cup stack\text{-}free\ (htd\ s'))\ (hmem\ s')\ (hmem\ s) \Longrightarrow htd\ s' = htd\ s \Longrightarrow G'\ w\ t' \Longrightarrow$
$c'\ w\ t' = c\ v\ s'$
assumes *M1*: $M1 \subseteq M$
assumes *G-G'*: $G\ t = G'\ i'\ t$
shows *refines* $(L2\text{-}while\ c\ f\ i\ ns)$
$(L2\text{-}seq\ (L2\text{-}guard\ G)$
$(\lambda\text{-}.\ (L2\text{-}while\ c'\ (\lambda v.\ L2\text{-}seq\ (g\ v)\ (\lambda res.\ L2\text{-}seq\ (L2\text{-}guard\ (G'\ res))\ (\lambda\text{-}.\ L2\text{-}gets\ (\lambda\text{-}.\ res)\ ns'))))\ i'\ ns'))\ )\ s\ t\ (rel\text{-}stack\ \mathcal{S}\ M1\ A\ s\ t_0\ (rel\text{-}xval\text{-}stack\ L\ R))$
⟨*proof*⟩

lemma *L2-unknown-rel-stack*:
assumes *s-t*: $rel\text{-}alloc\ \mathcal{S}\ M\ A\ t_0\ s\ t$
shows *refines* $(L2\text{-}unknown\ ns)\ (L2\text{-}unknown\ ns)\ s\ t\ (rel\text{-}stack\ \mathcal{S}\ \{\}\ A\ s\ t_0\ (rel\text{-}xval\text{-}stack\ L\ (\lambda\text{-}.\ (=))))$
⟨*proof*⟩

lemma *L2-fail-rel-stack*:
assumes *s-t*: $rel\text{-}alloc\ \mathcal{S}\ M\ A\ t_0\ s\ t$
shows *refines* $(L2\text{-}fail)\ (L2\text{-}fail)\ s\ t\ (rel\text{-}stack\ \mathcal{S}\ \{\}\ A\ s\ t_0\ (rel\text{-}xval\text{-}stack\ L\ R))$
⟨*proof*⟩

lemma *L2-undefined-function-rel-stack*:
assumes *s-t*: $rel\text{-}alloc\ \mathcal{S}\ M\ A\ t_0\ s\ t$
shows *refines* $((L2\text{-}guard\ (\lambda s.\ UNDEFINED\text{-}FUNCTION)))\ L2\text{-}fail\ s\ t\ (rel\text{-}stack\ \mathcal{S}\ \{\}\ A\ s\ t_0\ (rel\text{-}xval\text{-}stack\ L\ R))$
⟨*proof*⟩

lemma *L2-skip-rel-stack*:
assumes *s-t*: $rel\text{-}alloc\ \mathcal{S}\ M\ A\ t_0\ s\ t$
shows *refines L2-skip L2-skip s t* $(rel\text{-}stack\ \mathcal{S}\ \{\}\ A\ s\ t_0\ (rel\text{-}xval\text{-}stack\ L\ (\lambda\text{-}.\ (=))))$
⟨*proof*⟩

lemma *L2-spec-rel-stack*:
assumes *s-t*: $rel\text{-}alloc\ \mathcal{S}\ M\ A\ t_0\ s\ t$
assumes $\bigwedge t'.\ (frame\ A\ t_0\ s,\ t') \in r' \Longrightarrow \exists\, s'.\ (s,\ s') \in r$
assumes $\bigwedge s'.\ (s,\ s') \in r \Longrightarrow (frame\ A\ t_0\ s,\ frame\ A\ t_0\ s') \in r'$

assumes $\bigwedge s'.\ (s,\ s') \in r \Longrightarrow$ *equal-upto* $(M \cup$ *stack-free* $(htd\ s'))$ $(hmem\ s')$ $(hmem\ s)$
assumes $\bigwedge s'.\ (s,\ s') \in r \Longrightarrow htd\ s' = htd\ s$
assumes $\bigwedge s'.\ (s,\ s') \in r \Longrightarrow$ *stack-free* $(htd\ s') \subseteq \mathcal{S}$
assumes $\bigwedge s'.\ (s,\ s') \in r \Longrightarrow$ *stack-free* $(htd\ s') \cap A = \{\}$
assumes $\bigwedge s'.\ (s,\ s') \in r \Longrightarrow$ *stack-free* $(htd\ s') \cap M = \{\}$
shows *refines* $(L2\text{-}spec\ r)$ $(L2\text{-}spec\ r')\ s\ t$ $(rel\text{-}stack\ \mathcal{S}\ M\ A\ s\ t_0$ $(rel\text{-}xval\text{-}stack\ L$ $(\lambda\text{-}.\ (=))))$
⟨*proof*⟩

lemma *L2-spec-rel-stack'*:
assumes *s-t*: *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s\ t$
assumes $\bigwedge t'.$ *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s\ t \Longrightarrow (frame\ A\ t_0\ s,\ t') \in r' \Longrightarrow \exists s'.\ (s,\ s') \in r$
assumes $\bigwedge s'.$ *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s\ t \Longrightarrow (s,\ s') \in r \Longrightarrow (frame\ A\ t_0\ s, frame\ A\ t_0\ s') \in r'$
assumes $\bigwedge s'.$ *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s\ t \Longrightarrow (s,\ s') \in r \Longrightarrow$ *equal-upto* $(M \cup$ *stack-free* $(htd\ s'))$ $(hmem\ s')$ $(hmem\ s)$
assumes $\bigwedge s'.$ *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s\ t \Longrightarrow (s,\ s') \in r \Longrightarrow htd\ s' = htd\ s$
assumes $\bigwedge s'.$ *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s\ t \Longrightarrow (s,\ s') \in r \Longrightarrow$ *stack-free* $(htd\ s') \subseteq \mathcal{S}$
assumes $\bigwedge s'.$ *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s\ t \Longrightarrow (s,\ s') \in r \Longrightarrow$ *stack-free* $(htd\ s') \cap A = \{\}$
assumes $\bigwedge s'.$ *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s\ t \Longrightarrow (s,\ s') \in r \Longrightarrow$ *stack-free* $(htd\ s') \cap M = \{\}$
shows *refines* $(L2\text{-}spec\ r)$ $(L2\text{-}spec\ r')\ s\ t$ $(rel\text{-}stack\ \mathcal{S}\ M\ A\ s\ t_0$ $(rel\text{-}xval\text{-}stack\ L$ $(\lambda\text{-}.\ (=))))$
⟨*proof*⟩

lemma *L2-spec-rel-stack-same*:
assumes *s-t*: *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s\ t$
assumes $\bigwedge t'.$ *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s\ t \Longrightarrow (frame\ A\ t_0\ s,\ t') \in r \Longrightarrow \exists s'.\ (s,\ s') \in r$
assumes $\bigwedge s'.$ *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s\ t \Longrightarrow (s,\ s') \in r \Longrightarrow (frame\ A\ t_0\ s, frame\ A\ t_0\ s') \in r$
assumes $\bigwedge s'.$ *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s\ t \Longrightarrow (s,\ s') \in r \Longrightarrow$ *equal-upto* $(M \cup$ *stack-free* $(htd\ s'))$ $(hmem\ s')$ $(hmem\ s)$
assumes $\bigwedge s'.$ *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s\ t \Longrightarrow (s,\ s') \in r \Longrightarrow htd\ s' = htd\ s$
assumes $\bigwedge s'.$ *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s\ t \Longrightarrow (s,\ s') \in r \Longrightarrow$ *stack-free* $(htd\ s') \subseteq \mathcal{S}$
assumes $\bigwedge s'.$ *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s\ t \Longrightarrow (s,\ s') \in r \Longrightarrow$ *stack-free* $(htd\ s') \cap A = \{\}$
assumes $\bigwedge s'.$ *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s\ t \Longrightarrow (s,\ s') \in r \Longrightarrow$ *stack-free* $(htd\ s') \cap M = \{\}$
shows *refines* $(L2\text{-}spec\ r)$ $(L2\text{-}spec\ r)\ s\ t$ $(rel\text{-}stack\ \mathcal{S}\ M\ A\ s\ t_0$ $(rel\text{-}xval\text{-}stack\ L$ $(\lambda\text{-}.\ (=))))$
⟨*proof*⟩

lemma *L2-spec-rel-stack-heap-agnostic*:
assumes *s-t*: *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s\ t$

assumes *hmem-unchanged*: $\bigwedge s\ s'.\ (s,\ s') \in r \Longrightarrow hmem\ s' = hmem\ s$
assumes *htd-unchanged*: $\bigwedge s\ s'.\ (s,\ s') \in r \Longrightarrow htd\ s' = htd\ s$
assumes *heap-irrelevant*: $\bigwedge s\ s'\ f\ g.\ (s,\ s') \in r \Longrightarrow (hmem\text{-}upd\ g\ (htd\text{-}upd\ f\ s),$ $hmem\text{-}upd\ g\ (htd\text{-}upd\ f\ s')) \in r$
shows *refines* (*L2-spec r*) (*L2-spec r*) *s t* (*rel-stack* $\mathcal{S}$ {} *A s* t_0 (*rel-xval-stack L* (λ-. (=))))
⟨*proof*⟩

lemma *L2-assume-rel-stack*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes $\bigwedge v\ s'.\ (v,\ s') \in f\ s \Longrightarrow$
$\exists\, w.\ (w,\ frame\ A\ t_0\ s') \in g\ (frame\ A\ t_0\ s) \wedge rel\text{-}stack\ \mathcal{S}\ M\ A\ s\ t_0\ R\ (v,\ s')\ (w,$ *frame A* t_0 $s')$
shows *refines* (*L2-assume f*) (*L2-assume g*) *s t* (*rel-stack* $\mathcal{S}$ *M A s* t_0 (*rel-xval-stack L R*))
⟨*proof*⟩

lemma *L2-assume-rel-stack-heap-agnostic*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *hmem-unchanged*: $\bigwedge a\ s\ s'.\ (a,\ s') \in f\ s \Longrightarrow hmem\ s' = hmem\ s$
assumes *htd-unchanged*: $\bigwedge a\ s\ s'.\ (a,\ s') \in f\ s \Longrightarrow htd\ s' = htd\ s$
assumes *heap-irrelevant*: $\bigwedge a\ s\ s'\ g\ h.\ (a,\ s') \in f\ s \Longrightarrow (a,\ hmem\text{-}upd\ h\ (htd\text{-}upd\ g\ s')) \in f\ (hmem\text{-}upd\ h\ (htd\text{-}upd\ g\ s))$
shows *refines* (*L2-assume f*) (*L2-assume f*) *s t* (*rel-stack* $\mathcal{S}$ {} *A s* t_0 (*rel-xval-stack L* (λ-. (=))))
⟨*proof*⟩

lemma *refines-rel-stack-embed-result'*:
assumes *rel-alloc*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f*: *refines f g s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
assumes *R1*: $\bigwedge v\ s'\ w\ t'.$ *rel-alloc* $\mathcal{S}$ *M A* t_0 $s'\ t' \Longrightarrow R\ (hmem\ s')\ v\ w \Longrightarrow$
equal-upto (*M1* $\cup$ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) $\Longrightarrow$
htd $s' = htd\ s \Longrightarrow$
R1 (*hmem s'*) *v* (*emb w*)
assumes *M2-M*: *M2* $\subseteq$ *M*
assumes *M1-M2*: *M1* $\subseteq$ *M2*
shows *refines f* (*L2-seq g* (*ETA-TUPLED* ($\lambda x.$ *L2-gets* (λ-. *emb x*) *ns*))) *s t* (*rel-stack* $\mathcal{S}$ *M2 A s* t_0 (*rel-xval-stack L R1*))
⟨*proof*⟩

lemma *refines-rel-stack-root-upd-result*:
assumes *rel-alloc*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f*: *P t* $\Longrightarrow$ *refines f* (*L2-seq* (*L2-guard P*) (λ-. *g*)) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
assumes *R1*: $\bigwedge v\ s'\ w\ t'.$ *rel-alloc* $\mathcal{S}$ *M A* t_0 $s'\ t' \Longrightarrow R\ (hmem\ s')\ v\ w \Longrightarrow P\ t$

$\Longrightarrow$
equal-upto (*M1* $\cup$ *stack-free* (*htd* s')) (*hmem* s') (*hmem* s) $\Longrightarrow$
htd s' = *htd* s $\Longrightarrow$
R1 (*hmem* s') v (*emb* w)
assumes *M2-M*: $P\ t \Longrightarrow M2 \subseteq M$
assumes *M1-M2*: $P\ t \Longrightarrow M1 \subseteq M2$
shows *refines* f (*L2-seq* (*L2-guard* P) (λ-. *L2-seq* g (*ETA-TUPLED* (λx. *L2-gets* (λ-. *emb* x) *ns*)))) $s\ t$ (*rel-stack* $\mathcal{S}$ *M2* $A\ s\ t_0$ (*rel-xval-stack* L *R1*))
⟨*proof*⟩

lemma *refines-rel-stack-embed-exit*:
assumes *rel-alloc*: *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s\ t$
assumes f: *refines* $f\ g\ s\ t$ (*rel-stack* $\mathcal{S}$ *M1* $A\ s\ t_0$ (*rel-xval-stack* $L\ R$))
assumes *L1*: $\bigwedge v\ s'\ w\ t'$. *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s'\ t' \Longrightarrow L$ (*hmem* s') $v\ w \Longrightarrow$
equal-upto (*M1* $\cup$ *stack-free* (*htd* s')) (*hmem* s') (*hmem* s) $\Longrightarrow$
htd s' = *htd* s $\Longrightarrow$
L1 (*hmem* s') v (*emb* w)
assumes *M2-M*: $M2 \subseteq M$
assumes *M1-M2*: $M1 \subseteq M2$
shows *refines*
f (*L2-catch* g (*ETA-TUPLED* (λx. *L2-throw* (*emb* x) *ns*))) $s\ t$
(*rel-stack* $\mathcal{S}$ *M2* $A\ s\ t_0$ (*rel-xval-stack L1* R))
⟨*proof*⟩

lemma *refines-rel-stack-embed-both*:
assumes *stack*: *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s\ t$
assumes *f-g*: *refines* $f\ g\ s\ t$ (*rel-stack* $\mathcal{S}$ *M1* $A\ s\ t_0$ (*rel-xval-stack* $L\ R$))
assumes *L*: $\bigwedge v\ s'\ w\ t'$. *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s'\ t' \Longrightarrow L$ (*hmem* s') $v\ w \Longrightarrow$
equal-upto (*M1* $\cup$ *stack-free* (*htd* s')) (*hmem* s') (*hmem* s) $\Longrightarrow$
htd s' = *htd* s $\Longrightarrow$
L1 (*hmem* s') v (*embL* w)
assumes *R*: $\bigwedge v\ s'\ w\ t'$. *rel-alloc* $\mathcal{S}\ M\ A\ t_0\ s'\ t' \Longrightarrow R$ (*hmem* s') $v\ w \Longrightarrow$
equal-upto (*M1* $\cup$ *stack-free* (*htd* s')) (*hmem* s') (*hmem* s) $\Longrightarrow$
htd s' = *htd* s $\Longrightarrow$
R1 (*hmem* s') v (*embR* w)
assumes *M2-M*: $M2 \subseteq M$
assumes *M1-M2*: $M1 \subseteq M2$
shows *refines*
f
(*L2-seq*
(*L2-catch* g (*ETA-TUPLED* (λx. *L2-throw* (*embL* x) *ns*)))
(*ETA-TUPLED* (λx. *L2-gets* (λ-. *embR* x) *ns*))) $s\ t$
(*rel-stack* $\mathcal{S}$ *M2* $A\ s\ t_0$ (*rel-xval-stack L1 R1*))
⟨*proof*⟩
end

lemma *refines-assume-result-and-state-left*:
assumes $\bigwedge s'\ v$. $(v, s') \in A\ s \Longrightarrow$ *refines* ($f\ v$) $g\ s'\ t\ Q$

shows *refines* (*do* { v <− *assume-result-and-state* A; $f\ v$ }) $g\ s\ t\ Q$
⟨*proof*⟩

lemma *refines-assume-result-and-state-right*:
assumes $A\ t \neq \{\}$
assumes $\bigwedge t'\ v.\ (v,\ t') \in A\ t \Longrightarrow$ *refines* $f\ (g\ v)\ s\ t'\ Q$
shows *refines* f (*do* { v <− *assume-result-and-state* A; $g\ v$ }) $s\ t\ Q$
⟨*proof*⟩

lemma *refines-assume-result-and-state-both*:
assumes $B\ t = \{\} \Longrightarrow A\ s = \{\}$
assumes $\bigwedge s'\ v\ t'\ w.\ (v,\ s') \in A\ s \Longrightarrow (w,\ t') \in B\ t \Longrightarrow$ *refines* $(f\ v)\ (g\ w)\ s'\ t'$ Q
shows *refines* (*do* { v <− *assume-result-and-state* A; $f\ v$ }) (*do* {w <− *assume-result-and-state* B; $g\ w$ }) $s\ t\ Q$
⟨*proof*⟩

lemma *refines-assume-result-and-state-both-same-val′*:
assumes $\bigwedge s'\ v.\ (v,\ s') \in A\ s \Longrightarrow \exists\, t'.\ (v,\ t') \in B\ t \wedge$ *refines* $(f\ v)\ (g\ v)\ s'\ t'\ Q$
shows *refines* (*do* { v <− *assume-result-and-state* A; $f\ v$ }) (*do* { v <− *assume-result-and-state* B; $g\ v$ }) $s\ t\ Q$
⟨*proof*⟩

lemma *refines-assume-result-and-state-both-same-val*:
assumes $\bigwedge v\ s'.\ (v,\ s') \in A\ s \Longrightarrow \exists\, t'.\ (v,\ t') \in B\ t$
assumes $\bigwedge s'\ v\ t'.\ (v,\ s') \in A\ s \Longrightarrow (v,\ t') \in B\ t \Longrightarrow$ *refines* $(f\ v)\ (g\ v)\ s'\ t'\ Q$
shows *refines* (*do* { v <− *assume-result-and-state* A; $f\ v$ }) (*do* { v <− *assume-result-and-state* B; $g\ v$ }) $s\ t\ Q$
⟨*proof*⟩

lemma *refines-assume-result-and-state-both-same-val-frame*:
assumes f: $t = frm\ s$
assumes $\bigwedge s'\ v.\ (v,\ s') \in A\ s \Longrightarrow (v,\ frm\ s') \in B\ t$
assumes $\bigwedge s'\ v.\ (v,\ s') \in A\ s \Longrightarrow$ *refines* $(f\ v)\ (g\ v)\ s'\ (frm\ s')\ Q$
shows *refines* (*do* { v <− *assume-result-and-state* A; $f\ v$ }) (*do* { v <− *assume-result-and-state* B; $g\ v$ }) $s\ t\ Q$
⟨*proof*⟩

lemma *refines-on-exit-left*:
assumes *f-g*: *refines* $f\ g\ s0\ t\ Q'$
assumes *1*: $\bigwedge s.\ \exists\, s'.\ (s,\ s') \in cleanup$
assumes *2*: $\bigwedge s\ v\ t\ w\ s'.\ Q'\ (v,\ s)\ (w,\ t) \Longrightarrow (s,\ s') \in cleanup \Longrightarrow Q\ (v,\ s')\ (w,\ t)$
shows *refines* (*on-exit* f *cleanup*) g *s0* t Q
⟨*proof*⟩

lemma *refines-on-exit-same-cleanup*:

assumes *f-g*: *refines f g s0 t Q'*
assumes *1*: $\bigwedge s.\ \exists\, s'.\ (s,\ s') \in cleanup$
assumes *2*: $\bigwedge s\ v\ t\ w\ s'\ t'.\ Q'\ (v,\ s)\ (w,\ t) \Longrightarrow (s,\ s') \in cleanup \Longrightarrow (t,\ t') \in cleanup \Longrightarrow Q\ (v,\ s')\ (w,\ t')$
shows *refines* (*on-exit f cleanup*) (*on-exit g cleanup*) *s0 t Q*
⟨*proof*⟩

lemma *refines-on-exit-same-cleanup-choice*:
assumes *f-g*: *refines f g s0 t Q'*
assumes *1*: $\bigwedge s.\ \exists\, s'.\ (s,\ s') \in cleanup$
assumes *2*: $\bigwedge s\ v\ t\ w\ s'.\ Q'\ (v,\ s)\ (w,\ t) \Longrightarrow (s,\ s') \in cleanup \Longrightarrow \exists\, t'.\ (t,\ t') \in cleanup \wedge Q\ (v,\ s')\ (w,\ t')$
shows *refines* (*on-exit f cleanup*) (*on-exit g cleanup*) *s0 t Q*
⟨*proof*⟩

lemma *refines-runs-to-partial-fuse-both*:
assumes *sim*: *refines f f' s s' Q*
assumes *runs-to-f*: $f \cdot s\ ?\{\!|P1|\!\}$
assumes *runs-to-f'*: $f' \cdot s'\ ?\{\!|P2|\!\}$
shows *refines f f' s s'* $(\lambda(r,t)\ (r',t').\ Q\ (r,t)\ (r',t') \wedge P1\ r\ t \wedge P2\ r'\ t')$
⟨*proof*⟩

definition *domain-bound A Q* = $(\forall\, h\ h'.\ equal\text{-}on\ A\ h\ h' \longrightarrow Q\ h = Q\ h')$

lemma *domain-bound-equal-on*: *domain-bound A Q* $\Longrightarrow$ *equal-on A h h'* $\Longrightarrow$ *Q h = Q h'*
⟨*proof*⟩

lemma *domain-bound-equal-on-subset*: *domain-bound A Q* $\Longrightarrow A \subseteq A' \Longrightarrow$ *equal-on A' h h'* $\Longrightarrow$ *Q h = Q h'*
⟨*proof*⟩

lemma *domain-bound-heap-update-list*:
fixes *p*::*'a*::*mem-type ptr*
assumes *domain-bound A Q*
assumes *length bs = size-of TYPE('a)*
assumes *ptr-span p* $\cap$ *A* = {}
shows *Q* (*heap-update-list* (*ptr-val p*) *bs h*) = *Q h*
⟨*proof*⟩

named-theorems *domain-bound-intros*

lemma *domain-bound-mono*: $A \subseteq A' \Longrightarrow$ *domain-bound A Q* $\Longrightarrow$ *domain-bound A' Q*
⟨*proof*⟩

lemma *domain-bound-eq*: *domain-bound* {} $(\lambda\text{-}.\ (=))$
⟨*proof*⟩

lemma *domain-bound-rel-sum-stack*[*domain-bound-intros*]: *domain-bound A L* $\Longrightarrow$ *domain-bound A R* $\Longrightarrow$ *domain-bound A* (*rel-sum-stack L R*)
⟨*proof*⟩

lemma *domain-bound-rel-xval-stack*[*domain-bound-intros*]: *domain-bound A L* $\Longrightarrow$ *domain-bound A R* $\Longrightarrow$ *domain-bound A* (*rel-xval-stack L R*)
⟨*proof*⟩

lemma *domain-bound-unreachable-exit* [*domain-bound-intros*]:
domain-bound A (*rel-exit* (λ- - -. *False*))
⟨*proof*⟩

lemma *domain-bound-bot*[*domain-bound-intros*]: *domain-bound A* (λ- - -. *False*)
⟨*proof*⟩

lemma *domain-bound-eq'*[*domain-bound-intros*]: *domain-bound A* (λ-. (=))
⟨*proof*⟩

lemma *domain-bound-top*[*domain-bound-intros*]: *domain-bound A* (λ- - -. *True*)
⟨*proof*⟩

lemma *domain-bound-rel-singleton-stack*: *domain-bound* (*ptr-span p*) (*rel-singleton-stack p*)
⟨*proof*⟩

lemma *domain-bound-rel-exit*[*domain-bound-intros*]:
domain-bound A Q $\Longrightarrow$ *domain-bound A* (*rel-exit Q*)
⟨*proof*⟩

lemma *domain-bound-rel-singleton-stack'*[*domain-bound-intros*]:
ptr-span p $\subseteq$ *A* $\Longrightarrow$ *domain-bound A* (*rel-singleton-stack p*)
⟨*proof*⟩

lemma *domain-rel-push*: *domain-bound A Q* $\Longrightarrow$ *equal-on* (*ptr-span p* $\cup$ *A*) *h h'* $\Longrightarrow$ *rel-push p Q h* = *rel-push p Q h'*
⟨*proof*⟩

lemma *domain-bound-rel-push*: *domain-bound A Q* $\Longrightarrow$ *domain-bound* (*ptr-span p* $\cup$ *A*) (*rel-push p Q*)
⟨*proof*⟩

lemma *domain-bound-rel-push'*[*domain-bound-intros*]: *ptr-span p* $\subseteq$ *A* $\Longrightarrow$ *domain-bound A Q* $\Longrightarrow$ *domain-bound A* (*rel-push p Q*)
⟨*proof*⟩

context *stack-heap-state*

begin

lemma *with-fresh-stack-ptr-rel-stack''*:
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *domain-bound*: *domain-bound A Q*
assumes *f*: $\bigwedge p\ s'\ t_0$.
⟦*ptr-span p* ⊆ $\mathcal{S}$; *ptr-span p* ⊆ *stack-free* (*htd s*);
ptr-span p ∩ *A* = {}; *ptr-span p* ∩ *M* = {};
root-ptr-valid (*htd s'*) *p*;
[*h-val* (*hmem s'*) *p*] ∈ *I s*;
equal-upto-heap-on (*ptr-span p*) *s' s*;
stack-free (*htd s'*) ⊆ *stack-free* (*htd s*);
rel-alloc $\mathcal{S}$ (*ptr-span p* ∪ *M*) (*ptr-span p* ∪ *A*) t_0 *s' t*⟧
⟹ *refines* (*f p*) *g s' t* (*rel-stack* $\mathcal{S}$ (*ptr-span p* ∪ *M1*) (*ptr-span p* ∪ *A*) *s'* t_0 *Q*)
assumes *M1-M*: *M1* ⊆ *M*
shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) *I f*) *g s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 *Q*)
⟨*proof*⟩

lemma *gen-with-fresh-stack-ptr-rel-stack*:
assumes *I'*: *hd* ‘ *I s* ⊆ *I'* — There are two cases of local variables: *I s* = (λ-. *UNIV*) (unitialized) and *I s* = (λ-. {*v*}) (initialized)
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *domain-bound*: *domain-bound A Q*
assumes *f*: $\bigwedge p\ s'\ t_0\ v$.
⟦*ptr-span p* ⊆ $\mathcal{S}$; *ptr-span p* ⊆ *stack-free* (*htd s*);
ptr-span p ∩ *A* = {}; *ptr-span p* ∩ *M* = {};
root-ptr-valid (*htd s'*) *p*;
h-val (*hmem s'*) *p* = *v*;
[*v*] ∈ *I s*;
equal-upto-heap-on (*ptr-span p*) *s' s*;
stack-free (*htd s'*) ⊆ *stack-free* (*htd s*);
rel-alloc $\mathcal{S}$ (*ptr-span p* ∪ *M*) (*ptr-span p* ∪ *A*) t_0 *s' t*⟧
⟹ *refines* (*f p*) (*g v*) *s' t* (*rel-stack* $\mathcal{S}$ (*ptr-span p* ∪ *M1*) (*ptr-span p* ∪ *A*) *s'* t_0 *Q*)
assumes *M1-M*: *M1* ⊆ *M*
shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) *I f*) (*select I'* >>= *g*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 *Q*)
⟨*proof*⟩

lemma *with-fresh-stack-ptr-rel-stack-uninitialized'*:
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f*: $\bigwedge p\ s'\ t_0\ v$.
⟦*ptr-span p* ⊆ $\mathcal{S}$; *ptr-span p* ⊆ *stack-free* (*htd s*);
ptr-span p ∩ *A* = {}; *ptr-span p* ∩ *M* = {};
root-ptr-valid (*htd s'*) *p*;
h-val (*hmem s'*) *p* = *v*;
equal-upto-heap-on (*ptr-span p*) *s' s*;

stack-free (*htd* s') $\subseteq$ *stack-free* (*htd* s);
rel-alloc $\mathcal{S}$ (*ptr-span* $p \cup M$) (*ptr-span* $p \cup A$) t_0 s' t⟧
$\Longrightarrow$ *refines* (*f p*) (*g v*) s' t (*rel-stack* $\mathcal{S}$ (*ptr-span* $p \cup$ *M1*) (*ptr-span* $p \cup A$) s' t_0 Q)
assumes *M1-M*: $M1 \subseteq M$
assumes *domain-bound*: *domain-bound* A Q
shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) (λ-. *UNIV*) (*L2-VARS f ns*)) (*L2-seq* (*L2-unknown ns*) *g*) s t (*rel-stack* $\mathcal{S}$ *M1* A s t_0 Q)
⟨*proof*⟩

lemma *with-fresh-stack-ptr-rel-stack-uninitialized*:
assumes *stack*: *rel-alloc* $\mathcal{S}$ M A t_0 s t
assumes *f*: $\bigwedge p$ s' t_0.
⟦*ptr-span p* $\subseteq \mathcal{S}$; *ptr-span p* $\subseteq$ *stack-free* (*htd s*);
ptr-span p $\cap$ $A = \{\}$; *ptr-span p* $\cap$ $M = \{\}$;
root-ptr-valid (*htd* s') *p*;
equal-upto-heap-on (*ptr-span p*) s' s;
stack-free (*htd* s') $\subseteq$ *stack-free* (*htd s*);
rel-alloc $\mathcal{S}$ (*ptr-span p* $\cup$ *M*) (*ptr-span p* $\cup$ *A*) t_0 s' t⟧
$\Longrightarrow$ *refines* (*f p*) (*g' s' p*) s' t (*rel-stack* $\mathcal{S}$ (*ptr-span p* $\cup$ *M1*) (*ptr-span p* $\cup$ *A*) s' t_0 Q)
assumes *g*: $\bigwedge s'$ *p*. *ptr-span p* $\cap$ $A = \{\} \Longrightarrow$ *equal-upto* (*ptr-span p*) (*hmem* s') (*hmem s*) $\Longrightarrow$ *g' s' p* = *g* (*h-val* (*hmem* s') *p*)
assumes *M1-M*: $M1 \subseteq M$
assumes *domain-bound*: *domain-bound* A Q
shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) (λ-. *UNIV*) (*L2-VARS f ns*)) (*L2-seq* (*L2-unknown ns*) *g*) s t (*rel-stack* $\mathcal{S}$ *M1* A s t_0 Q)
⟨*proof*⟩

lemma *with-fresh-stack-ptr-rel-stack-uninitialized-g-normalised*:
assumes *stack*: *rel-alloc* $\mathcal{S}$ M A t_0 s t
assumes *f*: $\bigwedge p$ s' t_0.
⟦*ptr-span p* $\subseteq \mathcal{S}$; *ptr-span p* $\subseteq$ *stack-free* (*htd s*);
ptr-span p $\cap$ $A = \{\}$; *ptr-span p* $\cap$ $M = \{\}$;
root-ptr-valid (*htd* s') *p*;
equal-upto-heap-on (*ptr-span p*) s' s;
stack-free (*htd* s') $\subseteq$ *stack-free* (*htd s*);
rel-alloc $\mathcal{S}$ (*ptr-span p* $\cup$ *M*) (*ptr-span p* $\cup$ *A*) t_0 s' t⟧
$\Longrightarrow$ *refines* (*f p*) (*g* (*h-val* (*hmem* s') *p*)) s' t (*rel-stack* $\mathcal{S}$ (*ptr-span p* $\cup$ *M1*) (*ptr-span p* $\cup$ *A*) s' t_0 Q)
assumes *M1-M*: $M1 \subseteq M$
assumes *domain-bound*: *domain-bound* A Q
shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) (λ-. *UNIV*) (*L2-VARS f ns*)) (*L2-seq* (*L2-unknown ns*) *g*) s t (*rel-stack* $\mathcal{S}$ *M1* A s t_0 Q)
⟨*proof*⟩

lemma *with-fresh-stack-ptr-rel-stack-initialized'*:
fixes *g*:: ('*d*, '*e*, '*s*) *exn-monad*
assumes *stack*: *rel-alloc* $\mathcal{S}$ M A t_0 s t

assumes f: $\bigwedge p\ s'\ t_0$.
$[\![ptr\text{-}span\ p \subseteq \mathcal{S};\ ptr\text{-}span\ p \subseteq stack\text{-}free\ (htd\ s);$
$ptr\text{-}span\ p \cap A = \{\};\ ptr\text{-}span\ p \cap M = \{\};$
$root\text{-}ptr\text{-}valid\ (htd\ s')\ p;$
$h\text{-}val\ (hmem\ s')\ p = v;$
$equal\text{-}upto\text{-}heap\text{-}on\ (ptr\text{-}span\ p)\ s'\ s;$
$stack\text{-}free\ (htd\ s') \subseteq stack\text{-}free\ (htd\ s);$
$rel\text{-}alloc\ \mathcal{S}\ (ptr\text{-}span\ p \cup M)\ (ptr\text{-}span\ p \cup A)\ t_0\ s'\ t]\!]$
$\Longrightarrow refines\ (f\ p)\ g\ s'\ t\ (rel\text{-}stack\ \mathcal{S}\ (ptr\text{-}span\ p \cup M1)\ (ptr\text{-}span\ p \cup A)\ s'\ t_0\ Q)$
assumes *M1-M*: $M1 \subseteq M$
assumes *domain-bound*: $domain\text{-}bound\ A\ Q$
shows $refines\ (with\text{-}fresh\text{-}stack\text{-}ptr\ (Suc\ 0)\ (\lambda\text{-}.\ \{[v]\})\ (L2\text{-}VARS\ f\ ns))\ g\ s\ t$
$(rel\text{-}stack\ \mathcal{S}\ M1\ A\ s\ t_0\ Q)$
⟨*proof*⟩

lemma *with-fresh-stack-ptr-rel-stack-initialized*:
fixes g:: $('d, 'e, 's)\ exn\text{-}monad$
assumes *stack*: $rel\text{-}alloc\ \mathcal{S}\ M\ A\ t_0\ s\ t$
assumes f: $\bigwedge p\ s'\ t_0$.
$[\![ptr\text{-}span\ p \subseteq \mathcal{S};\ ptr\text{-}span\ p \subseteq stack\text{-}free\ (htd\ s);$
$ptr\text{-}span\ p \cap A = \{\};\ ptr\text{-}span\ p \cap M = \{\};$
$root\text{-}ptr\text{-}valid\ (htd\ s')\ p;$
$h\text{-}val\ (hmem\ s')\ p = v;$
$equal\text{-}upto\text{-}heap\text{-}on\ (ptr\text{-}span\ p)\ s'\ s;$
$stack\text{-}free\ (htd\ s') \subseteq stack\text{-}free\ (htd\ s);$
$rel\text{-}alloc\ \mathcal{S}\ (ptr\text{-}span\ p \cup M)\ (ptr\text{-}span\ p \cup A)\ t_0\ s'\ t]\!]$
$\Longrightarrow refines\ (f\ p)\ (g'\ s'\ p)\ s'\ t\ (rel\text{-}stack\ \mathcal{S}\ (ptr\text{-}span\ p \cup M1)\ (ptr\text{-}span\ p \cup A)$
$s'\ t_0\ Q)$
assumes *g*: $\bigwedge s'\ p.\ ptr\text{-}span\ p \cap A = \{\} \Longrightarrow equal\text{-}upto\ (ptr\text{-}span\ p)\ (hmem\ s')$
$(hmem\ s) \Longrightarrow h\text{-}val\ (hmem\ s')\ p = v \Longrightarrow g'\ s'\ p = g$
assumes *M1-M*: $M1 \subseteq M$
assumes *domain-bound*: $domain\text{-}bound\ A\ Q$
shows $refines\ (with\text{-}fresh\text{-}stack\text{-}ptr\ (Suc\ 0)\ (\lambda\text{-}.\ \{[v]\})\ (L2\text{-}VARS\ f\ ns))\ g\ s\ t$
$(rel\text{-}stack\ \mathcal{S}\ M1\ A\ s\ t_0\ Q)$
⟨*proof*⟩

lemma *with-fresh-stack-ptr-rel-stack-fix-initialized*:
fixes g:: $('d, 'e, 's)\ exn\text{-}monad$
assumes *stack*: $rel\text{-}alloc\ \mathcal{S}\ M\ A\ t_0\ s\ t$
assumes f: $\bigwedge p\ s'\ t_0$.
$[\![ptr\text{-}span\ p \subseteq \mathcal{S};\ ptr\text{-}span\ p \subseteq stack\text{-}free\ (htd\ s);$
$ptr\text{-}span\ p \cap A = \{\};\ ptr\text{-}span\ p \cap M = \{\};$
$root\text{-}ptr\text{-}valid\ (htd\ s')\ p;$
$h\text{-}val\ (hmem\ s')\ p = v;$
$equal\text{-}upto\text{-}heap\text{-}on\ (ptr\text{-}span\ p)\ s'\ s;$
$stack\text{-}free\ (htd\ s') \subseteq stack\text{-}free\ (htd\ s);$
$rel\text{-}alloc\ \mathcal{S}\ (ptr\text{-}span\ p \cup M)\ (ptr\text{-}span\ p \cup A)\ t_0\ s'\ t]\!]$
$\Longrightarrow refines\ (f\ p)\ (g'\ s'\ p)\ s'\ t\ (rel\text{-}stack\ \mathcal{S}\ (ptr\text{-}span\ p \cup M1)\ (ptr\text{-}span\ p \cup A)$
$s'\ t_0\ Q)$

assumes *g*: $\bigwedge s'\ p.\ \textit{ptr-span}\ p \cap A = \{\} \Longrightarrow \textit{equal-upto}\ (\textit{ptr-span}\ p)\ (\textit{hmem}\ s')\ (\textit{hmem}\ s) \Longrightarrow \textit{h-val}\ (\textit{hmem}\ s')\ p = v \Longrightarrow g'\ s'\ p = g$
assumes *M1-M*: $M1 \subseteq M$
assumes *I*: $I\ s = \{[v]\}$
assumes *domain-bound*: *domain-bound A Q*
shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) *I* (*L2-VARS f ns*)) *g s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 *Q*)
⟨*proof*⟩

lemma *with-fresh-stack-ptr-rel-stack-fix-initialized-g-normalised*:
fixes *g*:: (′*d*, ′*e*, ′*s*) *exn-monad*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f*: $\bigwedge p\ s'\ t_0.$
⟦*ptr-span p* ⊆ $\mathcal{S}$; *ptr-span p* ⊆ *stack-free* (*htd s*);
ptr-span p ∩ *A* = {}; *ptr-span p* ∩ *M* = {};
root-ptr-valid (*htd s*′) *p*;
h-val (*hmem s*′) *p* = *v*;
equal-upto-heap-on (*ptr-span p*) *s*′ *s*;
stack-free (*htd s*′) ⊆ *stack-free* (*htd s*);
rel-alloc $\mathcal{S}$ (*ptr-span p* ∪ *M*) (*ptr-span p* ∪ *A*) t_0 *s*′ *t*⟧
⟹ *refines* (*f p*) *g s*′ *t* (*rel-stack* $\mathcal{S}$ (*ptr-span p* ∪ *M1*) (*ptr-span p* ∪ *A*) *s*′ t_0 *Q*)
assumes *M1-M*: $M1 \subseteq M$
assumes *I*: $I\ s = \{[v]\}$
assumes *domain-bound*: *domain-bound A Q*
shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) *I* (*L2-VARS f ns*)) *g s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 *Q*)
⟨*proof*⟩

lemma *with-fresh-stack-ptr-rel-stack*:
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f*: $\bigwedge p\ s'\ t_0.$
⟦
ptr-span p ⊆ $\mathcal{S}$; *ptr-span p* ⊆ *stack-free* (*htd s*);
ptr-span p ∩ *A* = {}; *ptr-span p* ∩ *M* = {};
root-ptr-valid (*htd s*′) *p*;
[*h-val* (*hmem s*′) *p*] ∈ *I s*;
equal-upto-heap-on (*ptr-span p*) *s*′ *s*;
stack-free (*htd s*′) ⊆ *stack-free* (*htd s*);
rel-alloc $\mathcal{S}$ (*ptr-span p* ∪ *M*) (*ptr-span p* ∪ *A*) t_0 *s*′ *t*⟧
⟹ *refines* (*f p*) (*g*′ *s*′ *p*) *s*′ *t* (*rel-stack* $\mathcal{S}$ (*ptr-span p* ∪ *M1*) (*ptr-span p* ∪ *A*) *s*′ t_0 *Q*)
assumes *g*: $\bigwedge s'\ p.\ \textit{ptr-span}\ p \cap A = \{\} \Longrightarrow \textit{equal-upto}\ (\textit{ptr-span}\ p)\ (\textit{hmem}\ s')\ (\textit{hmem}\ s) \Longrightarrow [\textit{h-val}\ (\textit{hmem}\ s')\ p] \in I\ s \Longrightarrow g'\ s'\ p = g$
assumes *M1-M*: $M1 \subseteq M$
assumes *domain-bound*: *domain-bound A Q*
shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) *I* (*L2-VARS f ns*)) *g s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 *Q*)
⟨*proof*⟩

lemma *refines-rel-stack-project-result*:
assumes *refines f g s t* (*rel-stack* $\mathcal{S}$ *M A s* t_0 (*rel-xval-stack L R*))
assumes $\bigwedge h\ x\ y.\ R\ h\ x\ y \Longrightarrow R'\ h\ x$ (*prj y*)
shows *refines f* (*L2-seq g* (*ETA-TUPLED* ($\lambda x.$ *L2-gets* (λ-. *prj x*) *ns*))) *s t* (*rel-stack* $\mathcal{S}$ *M A s* t_0 (*rel-xval-stack L R'*))
⟨*proof*⟩

lemma *refines-rel-stack-adjust-result*:
assumes *refines f g s t* (*rel-stack* $\mathcal{S}$ *M A s* t_0 (*rel-xval-stack L R*))
assumes $\bigwedge s'\ t'\ v\ w.\ R$ (*hmem s'*) *v w* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M A* t_0 s' $t' \Longrightarrow$
equal-upto (*M* $\cup$ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) $\Longrightarrow$
equal-upto M (*htd s'*) (*htd s*) $\Longrightarrow$
equal-on $\mathcal{S}$ (*htd s'*) (*htd s*) $\Longrightarrow R'$ (*hmem s'*) *v* (*adj w*)
shows *refines f* (*L2-seq g* (*ETA-TUPLED* ($\lambda x.$ *L2-gets* (λ-. *adj x*) *ns*))) *s t* (*rel-stack* $\mathcal{S}$ *M A s* t_0 (*rel-xval-stack L R'*))
⟨*proof*⟩

lemma *stack-allocs-frame*:
assumes *alloc*: (*p*, *d*) ∈ *stack-allocs* (*Suc 0*) $\mathcal{S}$ *TYPE*('a::*mem-type*) (*htd s*)
shows ∃ *d*. (*p*, *d*) ∈ *stack-allocs* (*Suc 0*) $\mathcal{S}$ *TYPE*('a::*mem-type*) (*htd* (*frame A* t_0 *s*))
⟨*proof*⟩

lemma *heap-list-update-nth*:
$\bigwedge h\ p.$ *length v* ≤ *addr-card* $\Longrightarrow$
i < *length v* $\Longrightarrow$
(*heap-update-list p v h*) (*p* + *of-nat i*) = *v*!*i*
⟨*proof*⟩

lemma *stack-alloc-simulation-aux*:
fixes *p*::'*a*::*mem-type ptr*
assumes *neq-stack-byte*: *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *disjnt*: *stack-free* (*htd s*) ∩ *A* = {}
assumes *stack-free*: ∀ *a*∈*ptr-span p*. *root-ptr-valid* (*htd s*) (*PTR*(*stack-byte*) *a*)
assumes *not-null*: *0* ∉ *ptr-span p*
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
assumes *root-ptr*: *root-ptr-valid* (*ptr-force-type p* (*htd s*)) *p*
shows
htd-upd ($\lambda x.$ *override-on* (*ptr-force-type p* (*htd s*)) (*htd* t_0) (*A* − *stack-free* (*ptr-force-type p* (*htd s*))))
(*hmem-upd* ($\lambda x.$ *override-on* (*heap-update-padding p* (*vs* ! *0*) *bs x*) (*hmem* t_0) (*A* ∪ *stack-free* (*ptr-force-type p* (*htd s*)))) *s*) =
htd-upd (λ-. *ptr-force-type p* (*override-on* (*htd s*) (*htd* t_0) (*A* − *stack-free* (*htd s*))))

(*hmem-upd* (λ*x*. *heap-update-padding p* (*vs* ! *0*) *bs* (*override-on x* (*hmem* t_0) (*A* ∪ *stack-free* (*htd s*)))) *s*)
⟨*proof*⟩

lemma *keep-with-fresh-stack-ptr-rel-stack'*:
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *I*: *I* (*frame A* t_0 *s*) = *I s*
assumes *f*: ⋀*p s' t*$_0$ *t*. — I could use the fixed t_0
⟦*ptr-span p* ⊆ $\mathcal{S}$; *ptr-span p* ⊆ *stack-free* (*htd s*);
ptr-span p ∩ *A* = {}; *ptr-span p* ∩ *M* = {};
root-ptr-valid (*htd s'*) *p*;
[*h-val* (*hmem s'*) *p*] ∈ *I s*;
equal-upto-heap-on (*ptr-span p*) *s' s*;
stack-free (*htd s'*) ⊆ *stack-free* (*htd s*);
rel-alloc $\mathcal{S}$ (*ptr-span p* ∪ *M*) *A* t_0 *s' t*;
ptr-span p ∩ *stack-free* (*htd s'*) = {}⟧
⟹ *refines* (*f p*) (*g p*) *s' t* (*rel-stack* $\mathcal{S}$ (*ptr-span p* ∪ *M1*) *A s'* t_0 *Q*)
assumes *M1-M*: *M1* ⊆ *M*
assumes *domain-bound*: *domain-bound A Q*
shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) *I f*) (*with-fresh-stack-ptr* (*Suc 0*) *I g*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 *Q*)
⟨*proof*⟩

lemma *keep-with-fresh-stack-ptr-rel-stack*:
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *I*: *I s* = *I* (*frame A* t_0 *s*) — FIXME: make refinement I' similar to condition in *L2-condition* / *L2-while*?
assumes *f*: ⋀*p s' t*$_0$ *t*. — I could use the fixed t_0
⟦*ptr-span p* ⊆ $\mathcal{S}$; *ptr-span p* ⊆ *stack-free* (*htd s*);
ptr-span p ∩ *A* = {}; *ptr-span p* ∩ *M* = {};
root-ptr-valid (*htd s'*) *p*;
[*h-val* (*hmem s'*) *p*] ∈ *I s*;
equal-upto-heap-on (*ptr-span p*) *s' s*;
stack-free (*htd s'*) ⊆ *stack-free* (*htd s*);
rel-alloc $\mathcal{S}$ (*ptr-span p* ∪ *M*) *A* t_0 *s' t*;
ptr-span p ∩ *stack-free* (*htd s'*) = {}⟧
⟹ *refines* (*f p*) (*g' s' p*) *s' t* (*rel-stack* $\mathcal{S}$ (*ptr-span p* ∪ *M1*) *A s'* t_0 *Q*)
assumes *g*: ⋀*s' p*. *ptr-span p* ∩ *A* = {} ⟹ *equal-upto* (*ptr-span p*) (*hmem s'*) (*hmem s*) ⟹ [*h-val* (*hmem s'*) *p*] ∈ *I s*
⟹ *g' s' p* = *g p*
assumes *M1-M*: *M1* ⊆ *M*
assumes *domain-bound*: *domain-bound A Q*
shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) *I* (*L2-VARS f ns*)) (*with-fresh-stack-ptr* (*Suc 0*) *I* (*L2-VARS g ns*)) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 *Q*)
⟨*proof*⟩

lemma *keep-with-fresh-stack-ptr-rel-stack-g-normalised*:

assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *I*: *I s* = *I* (*frame A* t_0 *s*) — FIXME: make refinement I' similar to condition in *L2-condition* / *L2-while*?
assumes *f*: $\bigwedge p\ s'\ t_0\ t$. — I could use the fixed t_0
⟦*ptr-span p* ⊆ $\mathcal{S}$; *ptr-span p* ⊆ *stack-free* (*htd s*);
ptr-span p ∩ *A* = {}; *ptr-span p* ∩ *M* = {};
root-ptr-valid (*htd s'*) *p*;
[*h-val* (*hmem s'*) *p*] ∈ *I s*;
equal-upto-heap-on (*ptr-span p*) *s' s*;
stack-free (*htd s'*) ⊆ *stack-free* (*htd s*);
rel-alloc $\mathcal{S}$ (*ptr-span p* ∪ *M*) *A* t_0 *s' t*;
ptr-span p ∩ *stack-free* (*htd s'*) = {}⟧
⟹ *refines* (*f p*) (*g p*) *s' t* (*rel-stack* $\mathcal{S}$ (*ptr-span p* ∪ *M1*) *A s'* t_0 *Q*)
assumes *M1-M*: *M1* ⊆ *M*
assumes *domain-bound*: *domain-bound A Q*
shows *refines* (*with-fresh-stack-ptr* (*Suc 0*) *I* (*L2-VARS f ns*)) (*with-fresh-stack-ptr* (*Suc 0*) *I* (*L2-VARS g ns*)) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 *Q*)
⟨*proof*⟩

lemma *refines-rel-stack-adapt-right*:
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *M1-M*: *M1* ⊆ *M*
assumes *f-g*: *refines f g s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
assumes *h*: $\bigwedge s'\ v\ t'\ w$.
R (*hmem s'*) *v w* ⟹
rel-alloc $\mathcal{S}$ *M1 A* t_0 *s' t'* ⟹
equal-upto (*M1* ∪ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) ⟹
htd s' = *htd s* ⟹
refines (*L2-gets* (λ-. *v*) *ns*) (*h w*) *s' t'* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R'*))
shows *refines f* (*L2-seq g h*) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R'*))
⟨*proof*⟩

lemma *refines-handleE'-both-sides*:
assumes *refines f g s t Q*
assumes $\bigwedge v\ v'\ s'\ t'$. *Q* (*Result v*, *s'*) (*Result v'*, *t'*) ⟹ *R* (*Result v*, *s'*) (*Result v'*, *t'*)
assumes $\bigwedge v\ e'\ s'\ t'$. *Q* (*Result v*, *s'*) (*Exn e'*, *t'*) ⟹ *refines* (*return v*) (*h' e'*) *s' t' R*
assumes $\bigwedge e\ v'\ s'\ t'$. *Q* (*Exn e*, *s'*) (*Result v'*, *t'*) ⟹ *refines* (*h e*) (*return v'*) *s' t' R*
assumes $\bigwedge e\ e'\ s'\ t'$. *Q* (*Exn e*, *s'*) (*Exn e'*, *t'*) ⟹ *refines* (*h e*) (*h' e'*) *s' t' R*
shows *refines* (*f* <*catch*> *h*) (*g* <*catch*> *h'*) *s t R*
⟨*proof*⟩

lemma *refines-rel-stack-map-exn*:
assumes *f-g*: *refines f g s t* (*rel-stack* $\mathcal{S}$ *M A s* t_0 (*rel-xval-stack L R*))

assumes *L*: $\bigwedge e\ e'\ s'.\ L\ (hmem\ s')\ e\ e' \Longrightarrow L'\ (hmem\ s')\ (emb\ e)\ (emb\ e')$
shows *refines* (*map-value* (*map-exn emb*) *f*) (*map-value* (*map-exn emb*) *g*) *s t* (*rel-stack* $\mathcal{S}$ *M A s* t_0 (*rel-xval-stack L' R*))
⟨*proof*⟩

lemma *refines-rel-stack-map-exn-exit*:
assumes *f-g*: *refines f g s t* (*rel-stack* $\mathcal{S}$ *M A s* t_0 (*rel-xval-stack* (*rel-exit L*) *R*))
assumes *L*: $\bigwedge e\ e'\ h.\ L\ h\ e\ e' \Longrightarrow L'\ h\ (emb\ e)\ (emb'\ e')$
shows *refines* (*map-value* (*map-exn* (*emb o the-Nonlocal*)) *f*) (*map-value* (*map-exn* (*emb'*)) *g*) *s t* (*rel-stack* $\mathcal{S}$ *M A s* t_0 (*rel-xval-stack L' R*))
⟨*proof*⟩

lemma *throw-L2-throw-conv*: *throw x* = *L2-throw x* []
⟨*proof*⟩

lemma *L2-catch-join-exn-conv*:
(*L2-catch*
(*L2-seq*
(*L2-catch g* (λ*x*. *L2-seq* (*liftE* (*h x*)) (λ-. *L2-throw* (*prj x*) *ns1*)))
(λ*s*. *liftE* (*g1 s*)))
(λ*r*. *L2-throw* (*emb r*) *ns2*)) =
(*L2-seq*
(*L2-catch g* (λ*x*. *L2-seq* (*liftE* (*h x*)) (λ-. *L2-throw* (*emb* (*prj x*)) *ns2*)))
(λ*s*. *liftE* (*g1 s*)))
⟨*proof*⟩

lemma *L2-call-rel-stack-embed-exit*:
assumes *L*: $\bigwedge e\ e'\ h.\ L\ h\ e\ e' \Longrightarrow L'\ h\ (emb\ e)\ (emb'\ e')$
assumes *f-g*: *refines*
f
(*L2-seq*
(*L2-catch g* (λ*x*. (*L2-seq* (*liftE* (*h x*)) (λ-. *L2-throw* (*prj x*) *ns'*))))
(λ*s*. *liftE* (*g1 s*)))
s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-exit L*) *R*))

shows *refines*
(*L2-call f* (*emb o the-Nonlocal*) *ns*)
(*L2-seq*
(*L2-catch g* (λ*x*. (*L2-seq* (*liftE* (*h x*)) (λ-. *L2-throw* (*emb'* (*prj x*)) *ns'*))))
(λ*s*. *liftE* (*g1 s*)))
s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L' R*))
⟨*proof*⟩

lemma *L2-call-rel-stack-nest-exit-guarded*:
assumes *f-g*: *P t* ⟹ *refines*

```
    f
    (L2-seq (L2-guard P) (λ-.
      (L2-seq
        (L2-catch g (λx. (L2-seq (liftE (h x)) (λ-. L2-throw (prj x) ns'))))
        (λs. liftE (g1 s)))))
    s t
    (rel-stack 𝒮 M1 A s t₀ (rel-xval-stack (rel-exit L) R))
  assumes L: ⋀e e' h. P t ⟹ L h e e' ⟹ L' h (emb e) (emb' e')
  shows refines
    (L2-call f (emb o the-Nonlocal) ns)
    (L2-seq (L2-guard P) (λ-.
      (L2-seq
        (L2-catch g (λx. (L2-seq (liftE (h x)) (λ-. L2-throw (emb' (prj x)) ns'))))
        (λs. liftE (g1 s)))))
    s t
    (rel-stack 𝒮 M1 A s t₀ (rel-xval-stack L' R))
  ⟨proof⟩

lemma bind-catch-liftE-assoc:
  (do {v ← (g <catch> (λx. do {
                                y ← liftE (h x);
                                throw (exn x y)
                              }));
       liftE (g1 v)
      }) =
      (do {v ← g; liftE (g1 v)} <catch> (λx. do {
                                y ← liftE (h x);
                                throw (exn x y)
                              }))
  ⟨proof⟩

lemma bind-catch-liftE-split-catch: (do {
                s ← g;
                liftE (g1 s)
              } <catch> (λx. do {
                            - ← liftE (h x);
                            throw (emb (prj x))
                          })) =
((do {
                s ← g;
                liftE (g1 s)
              } <catch> (λx. do {
                            - ← liftE (h x);
                            throw (prj x) }))
             <catch> (λx. throw (emb x)))
  ⟨proof⟩
```

lemma *refines-catch-right-trans*:
fixes *f*:: ('*e*, '*a*, '*s*) *exn-monad*
assumes *f-g*: *refines f g s t Q*
assumes *R*: $\bigwedge$ *r s' e t'*. *Q* (*r*, *s'*) (*Exn e*, *t'*) $\Longrightarrow$ *refines* (*yield r*) (*h e*) *s' t' R*
assumes *Exn-Res*: $\bigwedge$*e s' v' t'*. *Q* (*Exn e*, *s'*) (*Result v'*, *t'*) $\Longrightarrow$ *R* (*Exn e*, *s'*) (*Result v'*, *t'*)
assumes *Res-Res*: $\bigwedge$*v v' s' t'*. *Q* (*Result v*, *s'*) (*Result v'*, *t'*) $\Longrightarrow$ *R* (*Result v*, *s'*) (*Result v'*, *t'*)
shows *refines f* (*g* <*catch*> *h*) *s t R*
⟨*proof*⟩

lemma *L2-call-rel-stack-embellish-exit*:
assumes *s-t*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f-g*: *P t* $\Longrightarrow$ *refines*
f
(*L2-seq* (*L2-guard P*) (λ-.
(*L2-seq*
(*L2-catch g* (λ*x*. (*L2-seq* (*liftE* (*h x*)) (λ-. *L2-throw* (*prj x*) *ns*))))
(λ*s*. *liftE* (*g1 s*)))))
s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-exit L*) *R*))
assumes *L*: $\bigwedge$*s' t' e e'*. *L* (*hmem s'*) *e e'* $\Longrightarrow$ *P t* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M A* t_0 *s' t'* $\Longrightarrow$
equal-upto (*M1* ∪ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*) $\Longrightarrow$ *htd s'* = *htd s* $\Longrightarrow$
L' (*hmem s'*) *e* (*emb e'*)
assumes *M1*: *P t* $\Longrightarrow$ *M1* ⊆ *M*
shows *refines*
f
(*L2-seq* (*L2-guard P*) (λ-.
(*L2-seq*
(*L2-catch g* (λ*x*. (*L2-seq* (*liftE* (*h x*)) (λ-. *L2-throw* (*emb* (*prj x*)) *ns-exit*))))
(λ*s*. *liftE* (*g1 s*)))))
s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-exit L'*) *R*))
⟨*proof*⟩

definition *override-heap-on P t1 t2* ≡
hmem-upd (λ*h*. *override-on h* (*hmem t2*) *P*)
(*htd-upd* (λ*d*. *override-on d* (*htd t2*) *P*) *t1*)

lemma *override-heap-on-empty* [*simp*]: *override-heap-on* {} *t1 t2* = *t1*
⟨*proof*⟩

lemma *refines-rel-stack-override-heap-on-exit*:
fixes *p*:: '*a*::*xmem-type ptr*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*

assumes *f-g*: $\bigwedge s\ t\ t_0$. *rel-alloc* $\mathcal{S}$ *M* $(P \cup A)$ t_0 *s t* $\Longrightarrow$
refines f (*g s*) *s t* (*rel-stack* $\mathcal{S}$ *M1* $(P \cup A)$ *s* t_0 *Q*)
assumes *disj-A*: $P \cap A = \{\}$
assumes *M1-M*: $M1 \subseteq M$
assumes *disj-stack-free-s*: $P \cap$ *stack-free* (*htd s*) $= \{\}$
shows *refines f* (*g s*) *s t*
(*rel-stack* $\mathcal{S}$ *M1* $(P \cup A)$ *s* (*override-heap-on P* t_0 *t*) *Q*)
⟨*proof*⟩

lemma *refines-rel-stack-override-heap-on-exit-guarded*:
fixes *p*:: *'a*::*xmem-type ptr*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f-g*: $\bigwedge s\ t\ t_0$. *G* $\Longrightarrow$ *rel-alloc* $\mathcal{S}$ *M* $(P \cup A)$ t_0 *s t* $\Longrightarrow$
refines f (*L2-seq* (*L2-guard* (λ-. *G*)) (λ-. *g s*)) *s t* (*rel-stack* $\mathcal{S}$ *M1* $(P \cup A)$ *s* t_0 *Q*)
assumes *disj-A*: *G* $\Longrightarrow$ $P \cap A = \{\}$
assumes *M1-M*: *G* $\Longrightarrow$ $M1 \subseteq M$
assumes *disj-stack-free-s*: *G* $\Longrightarrow$ $P \cap$ *stack-free* (*htd s*) $= \{\}$
shows *refines f* (*L2-seq* (*L2-guard* (λ-. *G*)) (λ-. *g s*)) *s t*
(*rel-stack* $\mathcal{S}$ *M1* $(P \cup A)$ *s* (*override-heap-on P* t_0 *t*) *Q*)
⟨*proof*⟩

lemma *override-heap-on-merge*:
(*override-heap-on* $(P1 \cup P2)$ t_0 *t*) = *override-heap-on P2* (*override-heap-on P1* t_0 *t*) *t*
⟨*proof*⟩

lemma *refines-rel-stack-override-heap-on-unmodified''*:
fixes *p*:: *'a*::*xmem-type ptr*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f-g*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1* $((P1 \cup P2) \cup A)$ *s* (*override-heap-on* $(P1 \cup P2)$ t_0 *t*) *Q*)
assumes *disj-A*: $(P1 \cup P2) \cap A = \{\}$
assumes *unmodified*: $P1 \cap M1 = \{\}$
assumes *M1-M*: $M1 \subseteq M$
assumes *disj-stack-free-s*: $(P1 \cup P2) \cap$ *stack-free* (*htd s*) $= \{\}$
shows *refines f g s t* (*rel-stack* $\mathcal{S}$ *M1* $(P2 \cup A)$ *s* (*override-heap-on P2* t_0 *t*) *Q*)
⟨*proof*⟩

lemma *refines-rel-stack-override-heap-on-unmodified'*:
fixes *p*:: *'a*::*xmem-type ptr*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f-g*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1* $(P \cup A)$ *s* (*override-heap-on P* t_0 *t*) *Q*)

assumes *disj-A*: $P \cap A = \{\}$
assumes *unmodified*: $P \cap M1 = \{\}$
assumes *M1-M*: $M1 \subseteq M$

assumes *disj-stack-free-s*: $P \cap$ *stack-free* (*htd s*) $= \{\}$
shows *refines f g s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 *Q*)
⟨*proof*⟩

lemma *refines-rel-stack-override-heap-on-unmodified*:
fixes *p*:: *'a*::*xmem-type ptr*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f-g*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1* (*P* ∪ *A*) *s* (*override-heap-on P* t_0 *t*) *Q*)
assumes *P*: *P* = (*P1* ∪ *P2*)
assumes *disj-A*: *P* ∩ *A* = {}
assumes *unmodified*: *P1* ∩ *M1* = {}
assumes *M1-M*: *M1* ⊆ *M*
assumes *disj-stack-free-s*: *P* ∩ *stack-free* (*htd s*) = {}
shows *refines f g s t* (*rel-stack* $\mathcal{S}$ *M1* (*P2* ∪ *A*) *s* (*override-heap-on P2* t_0 *t*) *Q*)
⟨*proof*⟩

lemma *refines-rel-stack-override-heap-on-exit-unmodified*:
fixes *p*:: *'a*::*xmem-type ptr*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f-g*: $\bigwedge s\ t\ t_0$. *rel-alloc* $\mathcal{S}$ *M* (*P* ∪ *A*) t_0 *s t* ⟹
refines f (*g s*) *s t* (*rel-stack* $\mathcal{S}$ *M1* (*P* ∪ *A*) *s* t_0 *Q*)
assumes *P*: *P* = (*P1* ∪ *P2*)
assumes *disj-A*: *P* ∩ *A* = {}
assumes *unmodified*: *P1* ∩ *M1* = {}
assumes *M1-M*: *M1* ⊆ *M*
assumes *disj-stack-free-s*: *P* ∩ *stack-free* (*htd s*) = {}
shows *refines f* (*g s*) *s t*
(*rel-stack* $\mathcal{S}$ *M1* (*P2* ∪ *A*) *s* (*override-heap-on P2* t_0 *t*) *Q*)
⟨*proof*⟩

lemma *refines-rel-stack-override-heap-on-exit-guarded-unmodified*:
fixes *p*:: *'a*::*xmem-type ptr*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f-g*: $\bigwedge s\ t\ t_0$. *G* ⟹ *rel-alloc* $\mathcal{S}$ *M* (*P* ∪ *A*) t_0 *s t* ⟹
refines f (*L2-seq* (*L2-guard* (λ-. *G*)) (λ-. *g s*)) *s t* (*rel-stack* $\mathcal{S}$ *M1* (*P* ∪ *A*) *s* t_0 *Q*)
assumes *P*: *G* ⟹ *P* = (*P1* ∪ *P2*)
assumes *disj-A*: *G* ⟹ *P* ∩ *A* = {}
assumes *unmodified*: *G* ⟹ *P1* ∩ *M1* = {}
assumes *M1-M*: *G* ⟹ *M1* ⊆ *M*
assumes *disj-stack-free-s*: *G* ⟹ *P* ∩ *stack-free* (*htd s*) = {}
shows *refines f* (*L2-seq* (*L2-guard* (λ-. *G*)) (λ-. *g s*)) *s t*
(*rel-stack* $\mathcal{S}$ *M1* (*P2* ∪ *A*) *s* (*override-heap-on P2* t_0 *t*) *Q*)
⟨*proof*⟩

lemma *refines-rel-stack-override-heap-emptyI*:

assumes *refines f g s t* (*rel-stack* $\mathcal{S}$ *M* (*P* ∪ *A*) *s* (*override-heap-on P* t_0 *t*) *Q*)
shows *refines f g s t* (*rel-stack* $\mathcal{S}$ *M* (*P* ∪ {} ∪ *A*) *s* (*override-heap-on* (*P* ∪ {}) t_0 *t*) *Q*)
⟨*proof*⟩

lemma *refines-rel-stack-pop-heap-both*:
fixes *p*:: *'a*::*xmem-type ptr*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f-g*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1* (*ptr-span p* ∪ *P* ∪ *A*) *s*
(*override-heap-on* (*ptr-span p* ∪ *P*) t_0 *t*)
(*rel-xval-stack* (*rel-exit* (*rel-push p L*)) (*rel-push p R*)))
assumes *disj-P-A*: *P* ∩ *A* = {}
assumes *disj-p-A*: *ptr-span p* ∩ *A* = {}
assumes *disj-p-P*: *ptr-span p* ∩ *P* = {}
assumes *disj-stack-free-s-p*: *ptr-span p* ∩ *stack-free* (*htd s*) = {}
assumes *disj-stack-free-s-P*: *P* ∩ *stack-free* (*htd s*) = {}
shows *refines*
f
(*L2-seq*
(*L2-catch g* (λ*x*. (*L2-seq* (*IO-modify-heap-paddingE p* (λ-. (*fst x*))) (λ-. *L2-throw* (*snd x*) *ns-exit*))))
(λ*x*. (*L2-seq* (*IO-modify-heap-paddingE p* (λ-. (*fst x*))) (λ-. *L2-return* (*snd x*) *ns*)))) *s t*
(*rel-stack* $\mathcal{S}$ *M1* (*P* ∪ *A*) *s* (*override-heap-on P* t_0 *t*) (*rel-xval-stack* (*rel-exit L*) *R*))
⟨*proof*⟩

lemma *refines-rel-stack-pop-heap-both-guarded*:
fixes *p*:: *'a*::*xmem-type ptr*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f-g*: *refines f* (*L2-seq* (*L2-guard* (λ-. *G*)) (λ-. *g*)) *s t*
(*rel-stack* $\mathcal{S}$ *M1* (*ptr-span p* ∪ *P* ∪ *A*) *s*
(*override-heap-on* (*ptr-span p* ∪ *P*) t_0 *t*)
(*rel-xval-stack* (*rel-exit* (*rel-push p L*)) (*rel-push p R*)))
assumes *disj-P-A*: *G* ⟹ *P* ∩ *A* = {}
assumes *disj-p-A*: *G* ⟹ *ptr-span p* ∩ *A* = {}
assumes *disj-p-P*: *G* ⟹ *ptr-span p* ∩ *P* = {}
assumes *disj-stack-free-s-p*: *G* ⟹ *ptr-span p* ∩ *stack-free* (*htd s*) = {}
assumes *disj-stack-free-s-P*: *G* ⟹ *P* ∩ *stack-free* (*htd s*) = {}
shows *refines*
f
(*L2-seq*
(*L2-catch* (*L2-seq* (*L2-guard* (λ-. *G*)) (λ-. *g*))
(λ*x*. (*L2-seq* (*IO-modify-heap-paddingE p* (λ-. (*fst x*))) (λ-. *L2-throw* (*snd x*) *ns-exit*))))
(λ*x*. (*L2-seq* (*IO-modify-heap-paddingE p* (λ-. (*fst x*))) (λ-. *L2-return* (*snd x*) *ns*)))) *s t*
(*rel-stack* $\mathcal{S}$ *M1* (*P* ∪ *A*) *s* (*override-heap-on P* t_0 *t*) (*rel-xval-stack* (*rel-exit L*)

R))
⟨*proof*⟩

lemma *refines-rel-stack-pop-heap-both-singleton*:
fixes *p*:: *'a*::*xmem-type ptr*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f-g*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1* (*ptr-span p* ∪ *P* ∪ *A*) *s*
(*override-heap-on* (*ptr-span p* ∪ *P*) t_0 *t*)
(*rel-xval-stack* (*rel-exit* (*rel-push p L*)) (*rel-singleton-stack p*)))
assumes *disj-P-A*: *P* ∩ *A* = {}
assumes *disj-p-A*: *ptr-span p* ∩ *A* = {}
assumes *disj-p-P*: *ptr-span p* ∩ *P* = {}
assumes *disj-stack-free-s-p*: *ptr-span p* ∩ *stack-free* (*htd s*) = {}
assumes *disj-stack-free-s-P*: *P* ∩ *stack-free* (*htd s*) = {}
shows *refines*
f
(*L2-seq*
(*L2-catch g* (λ*x*. (*L2-seq* (*IO-modify-heap-paddingE p* (λ-. (*fst x*))) (λ-. *L2-throw* (*snd x*) *ns-exit*))))
(λ*x*. (*IO-modify-heap-paddingE p* (λ-. *x*)))) *s t*
(*rel-stack* $\mathcal{S}$ *M1* (*P* ∪ *A*) *s* (*override-heap-on P* t_0 *t*) (*rel-xval-stack* (*rel-exit L*) (λ-. (=))))
⟨*proof*⟩

lemma *refines-rel-stack-pop-heap-both-singleton-guarded*:
fixes *p*:: *'a*::*xmem-type ptr*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f-g*: *refines f* (*L2-seq* (*L2-guard* (λ-. *G*)) (λ-. *g*)) *s t*
(*rel-stack* $\mathcal{S}$ *M1* (*ptr-span p* ∪ *P* ∪ *A*) *s*
(*override-heap-on* (*ptr-span p* ∪ *P*) t_0 *t*)
(*rel-xval-stack* (*rel-exit* (*rel-push p L*)) (*rel-singleton-stack p*)))
assumes *disj-P-A*: *G* ⟹ *P* ∩ *A* = {}
assumes *disj-p-A*: *G* ⟹ *ptr-span p* ∩ *A* = {}
assumes *disj-p-P*: *G* ⟹ *ptr-span p* ∩ *P* = {}
assumes *disj-stack-free-s-p*: *G* ⟹ *ptr-span p* ∩ *stack-free* (*htd s*) = {}
assumes *disj-stack-free-s-P*: *G* ⟹ *P* ∩ *stack-free* (*htd s*) = {}
shows *refines*
f
(*L2-seq*
(*L2-catch* (*L2-seq* (*L2-guard* (λ-. *G*)) (λ-. *g*))
(λ*x*. (*L2-seq* (*IO-modify-heap-paddingE p* (λ-. (*fst x*))) (λ-. *L2-throw* (*snd x*) *ns-exit*))))
(λ*x*. (*IO-modify-heap-paddingE p* (λ-. *x*)))) *s t*
(*rel-stack* $\mathcal{S}$ *M1* (*P* ∪ *A*) *s* (*override-heap-on P* t_0 *t*) (*rel-xval-stack* (*rel-exit L*) (λ-. (=))))
⟨*proof*⟩

lemma *refines-rel-stack-pop-heap-no-exit*:

fixes *p*:: *'a*::*xmem-type ptr*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f-g*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1* (*ptr-span p* ∪ *P* ∪ *A*) *s*
(*override-heap-on* (*ptr-span p* ∪ *P*) t_0 *t*)
(*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) (*rel-push p R*)))
assumes *disj-P-A*: *P* ∩ *A* = {}
assumes *disj-p-A*: *ptr-span p* ∩ *A* = {}
assumes *disj-p-P*: *ptr-span p* ∩ *P* = {}
assumes *disj-stack-free-s-p*: *ptr-span p* ∩ *stack-free* (*htd s*) = {}
assumes *disj-stack-free-s-P*: *P* ∩ *stack-free* (*htd s*) = {}
shows *refines*
f
(*L2-seq g* (λ*x*. (*L2-seq* (*IO-modify-heap-paddingE p* (λ-. (*fst x*))) (λ-. *L2-return* (*snd x*) *ns*)))) *s t*
(*rel-stack* $\mathcal{S}$ *M1* (*P* ∪ *A*) *s* (*override-heap-on P* t_0 *t*) (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) *R*))
⟨*proof*⟩

lemma *refines-rel-stack-pop-heap-no-exit-guarded*:
fixes *p*:: *'a*::*xmem-type ptr*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f-g*: *refines f* (*L2-seq* (*L2-guard* (λ-. *G*)) (λ-. *g*)) *s t*
(*rel-stack* $\mathcal{S}$ *M1* (*ptr-span p* ∪ *P* ∪ *A*) *s*
(*override-heap-on* (*ptr-span p* ∪ *P*) t_0 *t*)
(*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) (*rel-push p R*)))
assumes *disj-P-A*: *G* ⟹ *P* ∩ *A* = {}
assumes *disj-p-A*: *G* ⟹ *ptr-span p* ∩ *A* = {}
assumes *disj-p-P*: *G* ⟹ *ptr-span p* ∩ *P* = {}
assumes *disj-stack-free-s-p*: *G* ⟹ *ptr-span p* ∩ *stack-free* (*htd s*) = {}
assumes *disj-stack-free-s-P*: *G* ⟹ *P* ∩ *stack-free* (*htd s*) = {}
shows *refines*
f
(*L2-seq* (*L2-seq* (*L2-guard* (λ-. *G*)) (λ-. *g*)) (λ*x*. (*L2-seq* (*IO-modify-heap-paddingE p* (λ-. (*fst x*))) (λ-. *L2-return* (*snd x*) *ns*)))) *s t*
(*rel-stack* $\mathcal{S}$ *M1* (*P* ∪ *A*) *s* (*override-heap-on P* t_0 *t*) (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) *R*))
⟨*proof*⟩

lemma *refines-rel-stack-pop-heap-no-exit-singleton*:
fixes *p*:: *'a*::*xmem-type ptr*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f-g*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1* (*ptr-span p* ∪ *P* ∪ *A*) *s*
(*override-heap-on* (*ptr-span p* ∪ *P*) t_0 *t*)
(*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) (*rel-singleton-stack p*)))
assumes *disj-P-A*: *P* ∩ *A* = {}
assumes *disj-p-A*: *ptr-span p* ∩ *A* = {}

assumes *disj-p-P*: *ptr-span p* ∩ *P* = {}
assumes *disj-stack-free-s-p*: *ptr-span p* ∩ *stack-free* (*htd s*) = {}
assumes *disj-stack-free-s-P*: *P* ∩ *stack-free* (*htd s*) = {}
shows *refines*
f
(*L2-seq g* (λ*x*. (*IO-modify-heap-paddingE p* (λ-. *x*)))) *s t*
(*rel-stack* $\mathcal{S}$ *M1* (*P* ∪ *A*) *s* (*override-heap-on P* t_0 *t*) (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) (λ-. (=))))
⟨*proof*⟩

lemma *refines-rel-stack-pop-heap-no-exit-singleton-guarded*:
fixes *p*:: *'a*::*xmem-type ptr*
assumes *stack*: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t*
assumes *f-g*: *refines f* (*L2-seq* (*L2-guard* (λ-. *G*)) (λ-. *g*)) *s t*
(*rel-stack* $\mathcal{S}$ *M1* (*ptr-span p* ∪ *P* ∪ *A*) *s*
(*override-heap-on* (*ptr-span p* ∪ *P*) t_0 *t*)
(*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) (*rel-singleton-stack p*)))
assumes *disj-P-A*: *G* ⟹ *P* ∩ *A* = {}
assumes *disj-p-A*: *G* ⟹ *ptr-span p* ∩ *A* = {}
assumes *disj-p-P*: *G* ⟹ *ptr-span p* ∩ *P* = {}
assumes *disj-stack-free-s-p*: *G* ⟹ *ptr-span p* ∩ *stack-free* (*htd s*) = {}
assumes *disj-stack-free-s-P*: *G* ⟹ *P* ∩ *stack-free* (*htd s*) = {}
shows *refines*
f
(*L2-seq* (*L2-seq* (*L2-guard* (λ-. *G*)) (λ-. *g*)) (λ*x*. (*IO-modify-heap-paddingE p* (λ-. *x*)))) *s t*
(*rel-stack* $\mathcal{S}$ *M1* (*P* ∪ *A*) *s* (*override-heap-on P* t_0 *t*) (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) (λ-. (=))))
⟨*proof*⟩

lemma *refines-rel-stack-shuffle-both*:
assumes *refines f g s t* (*rel-stack* $\mathcal{S}$ *M A s* t_0 (*rel-xval-stack* (*rel-exit L*) *R*))
assumes ⋀*h e e'*. *L h e e'* ⟹ *L' h e* (*shuffle-exit e'*)
assumes ⋀*h v v'*. *R h v v'* ⟹ *R' h v* (*shuffle v'*)
shows *refines*
f
(*L2-seq*
(*L2-catch g* (λ*e*. *L2-throw* (*shuffle-exit e*) *ns-exit*))
(λ*x*. *L2-return* (*shuffle x*) *ns*))
s t
(*rel-stack* $\mathcal{S}$ *M A s* t_0 (*rel-xval-stack* (*rel-exit L'*) *R'*))
⟨*proof*⟩

lemma *refines-rel-stack-shuffle-no-exit*:
assumes *refines f g s t* (*rel-stack* $\mathcal{S}$ *M A s* t_0 (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) *R*))
assumes ⋀*h v v'*. *R h v v'* ⟹ *R' h v* (*shuffle v'*)
shows *refines*
f

(*L2-seq*
g
(*λx. L2-return* (*shuffle x*) *ns*))
s t
(*rel-stack* $\mathcal{S}$ *M A s* t_0 (*rel-xval-stack* (*rel-exit* (*λ- - -. False*)) *R′*))
⟨*proof*⟩

lemma *L2-call-rel-stack-bare′*:
assumes *f*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
shows *refines*
f
(*L2-call g* (*λx. x*) *ns′*)
s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))
⟨*proof*⟩

lemma *L2-call-rel-stack-bare-retype-unreachable-exit′*:
assumes *f*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-exit* (*λ- - -. False*)) *R*))
shows *refines*
f
(*L2-call g emb ns′*)
s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-exit* (*λ- - -. False*)) *R*))
⟨*proof*⟩

lemma *L2-call-rel-stack-bare-retype-unreachable-exit″*:
assumes *f*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-exit* (*λ- - -. False*)) *R*))
shows *refines*
f
(*L2-seq* (*L2-guard* (*λ-. P*)) (*λ-.*
(*L2-seq* (*L2-catch* (*L2-call g emb ns*)
(*λx. L2-seq* (*liftE* (*return* ())) (*λ-. L2-throw x ns-exit*))))
(*λv. L2-return v ns*)))
s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-exit* (*λ- - -. False*)) *R*))
⟨*proof*⟩

lemma *L2-call-rel-stack-bare-retype-unreachable-exit*:
assumes *f*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-exit* (*λ- - -. False*)) *R*))
shows *refines*
f
(*L2-seq* (*L2-guard* (*λ-. P*)) (*λ-.*
(*L2-seq* (*L2-catch* (*L2-call g undefined ns*)
(*λx. L2-seq* (*liftE* (*return* ())) (*λ-. L2-throw x ns-exit*))))

(λv. L2-return v ns)))
s t
(rel-stack $\mathcal{S}$ M1 A s t_0 (rel-xval-stack (rel-exit (λ- - -. False)) R))
⟨proof⟩

lemma *L2-call-rel-stack-bare-retype-unreachable-exit-extend-modifies'*:
assumes *f*: *refines f g s t*
(rel-stack $\mathcal{S}$ M1 A s t_0 (rel-xval-stack (rel-exit (λ- - -. False)) R))
assumes $P \Longrightarrow$ *stack-free* (*htd s*) ∩ *M2* = {}
assumes *M1-M2*: $P \Longrightarrow$ *M1* ⊆ *M2*
shows *refines*
f
(*L2-seq* (*L2-guard* (λ-. *P*)) (λ-.
(*L2-seq* (*L2-catch* (*L2-call g emb ns*)
(λx. *L2-seq* (*liftE* (*return* ())) (λ-. *L2-throw x ns-exit*))))
(λv. *L2-return v ns*)))
s t
(*rel-stack* $\mathcal{S}$ *M2 A s* t_0 (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) *R*))
⟨*proof*⟩

lemma *L2-call-rel-stack-bare-retype-unreachable-exit-extend-modifies*:
assumes *f*: *refines f g s t*
(rel-stack $\mathcal{S}$ M1 A s t_0 (rel-xval-stack (rel-exit (λ- - -. False)) R))
assumes *sf*: $P \Longrightarrow$ *stack-free* (*htd s*) ∩ *M2* = {}
assumes *M1-M2*: $P \Longrightarrow$ *M1* ⊆ *M2*
shows *refines*
f
(*L2-seq* (*L2-guard* (λ-. *P*)) (λ-.
(*L2-seq* (*L2-catch* (*L2-call g undefined ns*)
(λx. *L2-seq* (*liftE* (*return* ())) (λ-. *L2-throw x ns-exit*))))
(λv. *L2-return v ns*)))
s t
(*rel-stack* $\mathcal{S}$ *M2 A s* t_0 (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) *R*))
⟨*proof*⟩

lemma *L2-call-rel-stack-bare*:
assumes *f*: *refines f g s t*
(rel-stack $\mathcal{S}$ M1 A s t_0 (rel-xval-stack (rel-exit L) R))
shows *refines*
f
(*L2-seq* (*L2-guard* (λ-. *P*)) (λ-.
(*L2-seq* (*L2-catch* (*L2-call g* (λx. *x*) *ns*)
(λx. *L2-seq* (*liftE* (*return* ())) (λ-. *L2-throw x ns-exit*))))
(λv. *L2-return v ns*)))
s t
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-exit L*) *R*))
⟨*proof*⟩

lemma *L2-call-rel-stack-bare-extend-modifies*:
assumes *f*: *refines f g s t*
(*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack* (*rel-exit L*) *R*))
assumes *sf*: $P \Longrightarrow$ *stack-free* (*htd s*) $\cap$ *M2* = {}
assumes *M1-M2*: $P \Longrightarrow$ *M1* $\subseteq$ *M2*
shows *refines*
f
(*L2-seq* (*L2-guard* (λ-. *P*)) (λ-.
(*L2-seq* (*L2-catch* (*L2-call g* (λ*x*. *x*) *ns*)
(λ*x*. *L2-seq* (*liftE* (*return* ())) (λ-. *L2-throw x ns-exit*))))
(λ*v*. *L2-return v ns*)))
s t
(*rel-stack* $\mathcal{S}$ *M2 A s* t_0 (*rel-xval-stack* (*rel-exit L*) *R*))
⟨*proof*⟩

lemma *refines-rel-stack-extend-modifies*:
assumes *f*: *refines f* (*L2-seq* (*L2-guard* (λ-. *P*)) (λ-. *g*)) *s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 *Q*)
assumes *sf*: *P* $\Longrightarrow$ *M2* $\cap$ *stack-free* (*htd s*) = {}
assumes *M1-M2*: *P* $\Longrightarrow$ *M1* $\subseteq$ *M2*
shows *refines f* (*L2-seq* (*L2-guard* (λ-. *P*)) (λ-. *g*)) *s t* (*rel-stack* $\mathcal{S}$ *M2 A s* t_0 *Q*)
⟨*proof*⟩

lemma *refines-rel-sum-stack-pop-exit'*:
assumes *f-g*:
refines f g s t
(*rel-stack* $\mathcal{S}$ *M A s* t_0
(*rel-xval-stack*
(*rel-exit* (*rel-push p L*))
R))
shows
refines f (*L2-catch g* (λ(*x*, *p*). *L2-throw p ns*)) *s t*
(*rel-stack* $\mathcal{S}$ *M A s* t_0
(*rel-xval-stack*
(*rel-exit L*)
R))
⟨*proof*⟩

lemma *refines-rel-sum-stack-pop-exit*:
assumes *f-g*:
refines f g s t
(*rel-stack* $\mathcal{S}$ *M A s* t_0
(*rel-xval-stack*
(*rel-exit* (*rel-push p L*))
R))
shows
refines f (*L2-catch g* (λ*x*. *L2-throw* (*snd x*) *ns*)) *s t*
(*rel-stack* $\mathcal{S}$ *M A s* t_0

(*rel-xval-stack*
(*rel-exit L*)
R))
⟨*proof*⟩

end

lemma *L2-call-fuse-handlers1*:
L2-seq
(*L2-catch*
(*L2-seq*
(*L2-catch g* (λ*x*. *L2-seq* (*liftE* (*h1 x*)) (λ-. *L2-throw* (*prj1 x*) *ns1*)))
(λ*x*. *liftE* (*g1 x*)))
(λ*x*. *L2-seq* (*liftE* (*h2 x*)) (λ-. *L2-throw* (*prj2 x*) *ns2*)))
(λ*x*. *liftE* (*g2 x*)) =
(*L2-seq*
(*L2-catch g* (λ*x*. *L2-seq* (*L2-seq* (*liftE* (*h1 x*)) (λ-. *liftE* (*h2* (*prj1 x*)))) (λ-. *L2-throw* (*prj2* (*prj1 x*)) *ns2*)))
(λ*x*. *L2-seq* (*liftE* (*g1 x*)) (λ*x*. *liftE* (*g2 x*))))
⟨*proof*⟩

lemma *L2-call-fuse-handlers2*:
(*L2-catch*
(*L2-seq*
(*L2-catch g* (λ*x*. *L2-seq* (*liftE* (*h1 x*)) (λ-. *L2-throw* (*prj1 x*) *ns1*)))
(λ*x*. *liftE* (*g1 x*)))
(λ*x*. *L2-throw* (*prj2 x*) *ns2*))
=
L2-seq
(*L2-catch g* (λ*x*. *L2-seq* (*liftE* (*h1 x*)) (λ-. *L2-throw* (*prj2* (*prj1 x*)) *ns2*)))
(λ*x*. (*liftE* (*g1 x*)))
⟨*proof*⟩

lemma *L2-call-triv-conv*: *L2-call f* (λ*x*. *x*) *ns* = *f*
⟨*proof*⟩

lemma *L2-catch-L2-call-conv*:
L2-catch
(*L2-call f* (λ*e*. *e*) *ns*)
(λ*e*. *L2-throw* (*emb e*) *ns*′) =
L2-call f (*ETA-TUPLED emb*) *ns*
⟨*proof*⟩

lemma *L2-seq-return-conv*:
L2-seq (*liftE* (*return x*)) (λ*x*. *liftE* (*return* (*f x*))) = *liftE* (*return* (*f x*))
⟨*proof*⟩

lemma *L2-seq-return-unused-conv*:
L2-seq (*liftE* (*return x*)) (λ-. *g*) = *g*
⟨*proof*⟩

lemma *L2-seq-L2-return-unused-conv*:
L2-seq (*L2-return x ns*) (λ-. *g*) = *g*
⟨*proof*⟩

lemma *L2-seq-return-id-conv*:
L2-seq f (λ*x*. *liftE* (*return x*)) = *f*
⟨*proof*⟩

lemma *L2-seq-L2-return-id-conv*:
L2-seq f (λ*x*. *L2-return x ns*) = *f*
⟨*proof*⟩

lemma *L2-catch-L2-guard-out-conv*: *L2-catch* (*L2-seq* (*L2-guard P*) (λ-. *X*)) *Y* = *L2-seq* (*L2-guard P*) (λ-. *L2-catch X Y*)
⟨*proof*⟩

lemma *L2-seq-guard-out-conv*:
L2-seq (*L2-seq* (*L2-guard P*) (λ-. *X*)) *Y* = (*L2-seq* (*L2-guard P*) (λ-. *L2-seq X Y*))
⟨*proof*⟩

lemma *L2-seq-L2-catch-assoc*:
L2-seq (*L2-seq* (*L2-catch X Y*) *Z1*) *Z2* = *L2-seq* (*L2-catch X Y*) (λ*x*. *L2-seq* (*Z1 x*) *Z2*)
⟨*proof*⟩

lemma *L2-catch-throw-id*: *L2-catch f* (λ*x*. *L2-throw x ns*) = *f*
⟨*proof*⟩

lemma *L2-catch-call-throw*:
(*L2-catch*
(*L2-call f emb ns*)
(λ*x*. *L2-throw* (*g x*) *ns-exit*)) =
L2-call f (*ETA-TUPLED* (λ*x*. (*g* (*emb x*)))) *ns*
⟨*proof*⟩

lemma *liftE-bind-L2-seq*: (*liftE* (*A* >>= *B*)) = *L2-seq* (*liftE A*) (*ETA-TUPLED* (λ*x*. *liftE* (*B x*)))
⟨*proof*⟩

lemma *liftE-L2-seq*: *L2-seq* (*liftE A*) (λ*x*. *liftE* (*B x*)) = (*liftE* (*A* >>= *B*))
⟨*proof*⟩

lemma *liftE-return-L2-gets-conv*: *liftE* (*return x*) = *L2-gets* (λ-. *x*) []
⟨*proof*⟩

lemma *L2-call-shuffle-conv*:
L2-call (*L2-seq* (*L2-catch f A*) *B*) (λ*x*. *x*) *ns* =
(*L2-seq* (*L2-catch* (*L2-call f* (λ*x*. *x*) *ns*) *A*) *B*)
⟨*proof*⟩
lemmas *L2-call-canonical-convs* =
L2-call-shuffle-conv
L2-seq-return-conv
L2-call-fuse-handlers2
L2-call-fuse-handlers1
L2-guard-join-nested
liftE-L2-seq

L2-seq-L2-catch-assoc
L2-catch-L2-guard-out-conv
L2-seq-guard-out-conv

bind-assoc

lemmas *L2-call-pre-final-convs* =
L2-seq-return-unused-conv
L2-seq-L2-return-unused-conv
L2-seq-return-id-conv
L2-seq-L2-return-id-conv
L2-catch-call-throw

lemmas *L2-call-final-convs* =
L2-seq-return-unused-conv
L2-seq-L2-return-unused-conv
L2-seq-return-id-conv
L2-seq-L2-return-id-conv
L2-catch-L2-call-conv

liftE-bind-L2-seq
L2-gets-unbound
L2-catch-throw-id
L2-return-L2-gets-conv
liftE-return-L2-gets-conv

L2-seq-L2-catch-assoc
L2-catch-L2-guard-out-conv
L2-seq-guard-out-conv

lemma *L2-call-ETA-TUPLED1*: *L2-seq* (*L2-catch g h*) *g1* ≡ *L2-seq* (*L2-catch g* (*ETA-TUPLED h*)) (*ETA-TUPLED g1*)
⟨*proof*⟩

lemma *L2-call-ETA-TUPLED2*: *L2-seq* (*L2-call g emb ns*) *g1* ≡ *L2-seq* (*L2-call g emb ns*) (*ETA-TUPLED g1*)
⟨*proof*⟩

lemma *distinct-sets-consI*: *distinct-sets ps* ⟹ *distinct-sets* (*p*#*ps*) ⟷ *p* ∩ ⋃ (*set ps*) = {}
⟨*proof*⟩

lemma *disjoint-subset-simps*:
ASSUMPTION (*A* ∩ *B* = {}) ⟹ *A'* ⊆ *A* ⟹ *B* ∩ *A'* = {} ⟷ *True*
ASSUMPTION (*B* ∩ *A* = {}) ⟹ *A'* ⊆ *A* ⟹ *B* ∩ *A'* = {} ⟷ *True*
ASSUMPTION (*A* ∩ *B* = {}) ⟹ *B'* ⊆ *B* ⟹ *B'* ∩ *A* = {} ⟷ *True*
ASSUMPTION (*B* ∩ *A* = {}) ⟹ *B'* ⊆ *B* ⟹ *B'* ∩ *A* = {} ⟷ *True*
⟨*proof*⟩

lemma *disjoint-subset-simps'*:
(*B* ∩ *A* = {}) ⟹ *A'* ⊆ *A* ⟹ *B* ∩ *A'* = {} ⟷ *True*
(*A* ∩ *B* = {}) ⟹ *B'* ⊆ *B* ⟹ *B'* ∩ *A* = {} ⟷ *True*
(*B* ∩ *A* = {}) ⟹ *B'* ⊆ *B* ⟹ *B'* ∩ *A* = {} ⟷ *True*
(*A* ∩ *B* = {}) ⟹ *A'* ⊆ *A* ⟹ *B* ∩ *A'* = {} ⟷ *True*
⟨*proof*⟩

lemma *field-lvalue-ptr-span-trans*:
fixes *p*:: *'a*::*mem-type ptr*
assumes *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*t*, *n*)
assumes *export-uinfo t* = *typ-uinfo-t* (*TYPE*(*'b*))
assumes *ptr-span p* ⊆ *A*
shows {&(*p*→*f*)..+*size-of TYPE*(*'b*::*c-type*)} ⊆ *A*
⟨*proof*⟩

lemma *field-lvalue-ptr-span-root-contained*:
fixes *p*:: *'a*::*mem-type ptr*
assumes *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*t*, *n*)
assumes *export-uinfo t* = *typ-uinfo-t* (*TYPE*(*'b*))
shows {&(*p*→*f*)..+*size-of TYPE*(*'b*::*c-type*)} ⊆ *ptr-span p*
⟨*proof*⟩

lemma *field-lvalue-disjoint-fields-same-root*:
fixes *p*:: *'a*::*mem-type ptr*
assumes *f*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *f 0* = *Some* (*T1*, *n*)
assumes *exp1*: *export-uinfo T1* = *typ-uinfo-t* (*TYPE*(*'b*::*c-type*))
assumes *g*: *field-lookup* (*typ-info-t TYPE*(*'a*)) *g 0* = *Some* (*T2*, *m*)
assumes *exp2*: *export-uinfo T2* = *typ-uinfo-t* (*TYPE*(*'c*::*c-type*))
assumes *prfx1*: ¬ *prefix f g*

assumes *prfx2*: ¬ *prefix g f*
shows {&(*p*→*f*)..+*size-of TYPE*('*b*::*c-type*)} ∩ {&(*p*→*g*)..+*size-of TYPE*('*c*::*c-type*)} = {}
⟨*proof*⟩

lemma *ptr-coerce-index-array-ptr-index-conv*:
ptr-coerce p +$_p$ *uint i* = *array-ptr-index p False* (*nat* (*uint i*))
⟨*proof*⟩

lemma *ptr-coerce-index-array-ptr-index-numeral-conv*:
ptr-coerce p +$_p$ (*numeral i*) = *array-ptr-index p False* (*numeral i*)
⟨*proof*⟩

lemma *ptr-coerce-index-array-ptr-index-numeral-conv'*:
fixes *p*:: (('*a* :: *c-type*)['*b* :: *finite*]) *ptr*
assumes *sz-eq*: *size-of TYPE*('*c*::*c-type*) = *size-of TYPE*('*a*)
shows *PTR-COERCE*('*a*['*b*] → '*c*::*c-type*) *p* +$_p$ (*numeral n*) = *PTR-COERCE*('*a* → '*c*) (*array-ptr-index p False* (*numeral n*))
⟨*proof*⟩

lemma *ptr-coerce-index-array-ptr-index-numeral-conv''*:
fixes *p*:: (('*a* :: *c-type*)['*b* :: *finite*]) *ptr*
assumes *sz-eq*: *size-of TYPE*('*c*::*c-type*) = *size-of TYPE*('*a*)
assumes *bound*: *numeral n* < *CARD*('*b*)
shows *PTR-COERCE*('*a*['*b*] → '*c*::*c-type*) *p* +$_p$ (*numeral n*) = *PTR*('*c*) &(*p*→[*replicate* (*numeral n*) *CHR* ''*1*''])
⟨*proof*⟩

lemma *ptr-coerce-index-array-ptr-index-0-conv*:
ptr-coerce p +$_p$ *0* = *array-ptr-index p False 0*
⟨*proof*⟩

lemma *ptr-coerce-index-array-ptr-index-1-conv*:
ptr-coerce p +$_p$ *1* = *array-ptr-index p False 1*
⟨*proof*⟩

lemma *ptr-coerce-index-array-ptr-index-sint-conv*:
0 ≤*s i* ⟹ *ptr-coerce p* +$_p$ *sint i* = *array-ptr-index p False* (*nat* (*sint i*))
⟨*proof*⟩

lemma *heap-access-Array-element''*:
fixes *p* :: ('*a*::*mem-type*['*b*::*finite*]) *ptr*
assumes *less*: *n* < *CARD*('*b*)
shows
index (*h-val hp p*) *n*
= *h-val hp* (*array-ptr-index p False n*)
⟨*proof*⟩

lemma *index-fupdate-split*: *i* < *CARD*('*n*) ⟹ *j* < *CARD*('*n*) ⟹

index (*fupdate i f* (*x*::'*a*['*n*::*finite*])) *j* = (*if i* ≠ *j then* (*x*.[*j*]) *else f* (*index x i*))
⟨*proof*⟩

lemma *root-disjoint-field-lvalue-disjoint1*:
fixes *p*::'*a*::*mem-type ptr*
assumes *field-lookup*: *field-lookup* (*typ-info-t TYPE*('*a*)) *path 0* = *Some* (*t*, *n*)
assumes *match*: *export-uinfo t* = *typ-uinfo-t TYPE*('*f*::*c-type*)
assumes *ptr-span p* ∩ *A* = {}
shows *A* ∩ {&(*p*→*path*)..+*size-of TYPE*('*f*)} = {}
⟨*proof*⟩

lemma *root-disjoint-field-lvalue-disjoint2*:
fixes *p*::'*a*::*mem-type ptr*
assumes *field-lookup*: *field-lookup* (*typ-info-t TYPE*('*a*)) *path 0* = *Some* (*t*, *n*)
assumes *match*: *export-uinfo t* = *typ-uinfo-t TYPE*('*f*::*c-type*)
assumes *ptr-span p* ∩ *A* = {}
shows {&(*p*→*path*)..+*size-of TYPE*('*f*)} ∩ *A* = {}
⟨*proof*⟩

context *heap-state-global*
begin

lemma *global-update-frame-commute*:
shows *glob-upd g* (*frame A t*$_0$ *s*) =
frame A t$_0$ (*glob-upd g s*)
⟨*proof*⟩

lemma *L2-modify-no-heap-update-rel-stack*[*synthesize-rule refines-in-out*]:
assumes *rel-alloc* $\mathcal{S}$ *M A t*$_0$ *s t*
assumes *v s* = *v*' *t*
shows *refines*
(*L2-modify* (λ*s*. *glob-upd* (λ-. *v s*) *s*))
(*L2-modify* (λ*s*. *glob-upd* (λ-. *v*' *s*) *s*))
s t
(*rel-stack* $\mathcal{S}$ {} *A s t*$_0$ (*rel-xval-stack L* (λ-. (=))))
⟨*proof*⟩

lemma *rel-alloc-glabal-independent-eq* [*rel-alloc-independent-globals*]:
assumes *rel-alloc* $\mathcal{S}$ *M A t*$_0$ *s t*
shows *glob t* = *glob s*
⟨*proof*⟩

end

lemma *L2-seq-guard-cong-stateless*:
(⋀*s*. *P s* = *P*' *s*) ⟹ (⋀*s*. *P*' *s* ⟹ *X* = *X*') ⟹
L2-seq (*L2-guard P*) (λ-. *X*) = *L2-seq* (*L2-guard P*') (λ-. *X*')
⟨*proof*⟩

context *globals-stack-heap-state*
begin
lemma *globals-subset-trans*: *NO-MATCH* $\mathcal{G}$ $X \Longrightarrow$ *NO-MATCH* $\mathcal{G}$ $Y \Longrightarrow X \subseteq \mathcal{G} \Longrightarrow \mathcal{G} \subseteq Y \Longrightarrow X \subseteq Y$
⟨*proof*⟩

lemma *globals-disjoint-subset-left*: *NO-MATCH* $\mathcal{G}$ $X \Longrightarrow X \subseteq \mathcal{G} \Longrightarrow \mathcal{G} \cap A = \{\} \Longrightarrow X \cap A = \{\}$
⟨*proof*⟩

lemma *globals-disjoint-subset-right*: *NO-MATCH* $\mathcal{G}$ $X \Longrightarrow X \subseteq \mathcal{G} \Longrightarrow A \cap \mathcal{G} = \{\} \Longrightarrow A \cap X = \{\}$
⟨*proof*⟩

end

lemma *Union-Diff-right-conv'*: $X \cup Y = X \cup (Y - X)$
⟨*proof*⟩

lemma *Union-Diff-right-conv*: $(Y - X) \equiv Z \Longrightarrow X \cup Y \equiv X \cup Z$
⟨*proof*⟩

lemma *Union-assoc*: $X \cup Y \cup Z = X \cup (Y \cup Z)$
⟨*proof*⟩

⟨*ML*⟩
declare [[*simproc del*: *Union-Diff-conv*]]

lemma
ptr-span q ∩ *ptr-span p* = {} ⟹ *ptr-span x* ∩ *ptr-span p* = {} ⟹
(*ptr-span p* ∪ (*ptr-span q* ∪ (*ptr-span x* ∪ *ptr-span p*))) = *ptr-span p* ∪ (*ptr-span q* ∪ *ptr-span x*)
⟨*proof*⟩

lemma
ptr-span q ∩ *ptr-span p* = {} ⟹ *ptr-span x* ∩ *ptr-span p* = {} ⟹
((*ptr-span p* ∪ *ptr-span q*) ∪ (*ptr-span x* ∪ *ptr-span p*)) = *ptr-span p* ∪ (*ptr-span q* ∪ *ptr-span x*)
⟨*proof*⟩

lemma
assumes *disj*: *ptr-span q* ∩ *ptr-span p* = {}
ptr-span p ∩ *ptr-span q* = {}
ptr-span x ∩ *ptr-span p* = {}
ptr-span x ∩ *ptr-span q* = {}
shows ((*ptr-span p* ∪ *ptr-span q*) ∪ ((*ptr-span p* ∪ *ptr-span q*) ∪ *ptr-span x* ∪ *ptr-span p*)) = *ptr-span p* ∪ (*ptr-span q* ∪ *ptr-span x*)
⟨*proof*⟩

lemma *subset-union-left*: $X \subseteq L \Longrightarrow X \subseteq L \cup R$
⟨*proof*⟩

lemma *subset-union-right*: $X \subseteq R \Longrightarrow X \subseteq L \cup R$
⟨*proof*⟩

lemma *disjoint-globals-stack*: $G \cap S = \{\} \Longrightarrow x \subseteq S \Longrightarrow G \cap x = \{\}$
⟨*proof*⟩
end

Chapter 21

HL phase: Heap Lifting / Split Heap

theory *Split-Heap*
imports
TypHeapSimple
begin

⟨*ML*⟩

definition *array-fields* :: *nat* ⇒ *qualified-field-name list* **where**
array-fields n = *map* (λ*n*. [*replicate n CHR ''1''*]) [*0*..<*n*]

lemma *Nil-nmem-array-fields*[*simp*]: [] ∉ *set* (*array-fields n*)
⟨*proof*⟩

lemma *distinct-prop-disj-fn-array-fields*[*simp*]: *distinct-prop disj-fn* (*array-fields n*)
⟨*proof*⟩

lemma *field-lookup-field-ti'*:
field-lookup (*typ-info-t TYPE*('*a* :: *c-type*)) *f 0* = *Some* (*a*, *b*) ⟹ *field-ti TYPE*('*a*) *f* = *Some a*
⟨*proof*⟩

lemma *field-lookup-append-add*:
wf-desc t ⟹
field-lookup t (*f* @ *g*) *n* =
Option.bind (*field-lookup t f n*) (λ(*t'*, *m*).
Option.bind (*field-lookup t' g 0*) (λ(*t''*, *m'*). *Some* (*t''*, *m* + *m'*)))
⟨*proof*⟩

lemma *nth-array-fields*: *i* < *n* ⟹ *array-fields n* ! *i* = [*replicate i CHR ''1''*]
⟨*proof*⟩
lemma *array-fields-Suc*: *array-fields* (*Suc n*) = *array-fields n* @ [[*replicate n CHR ''1''*]]

⟨*proof*⟩

lemma *find-append*: *find P* (*xs* @ *ys*) = (*case find P xs of None* ⇒ *find P ys* | *Some x* => *Some x*)
⟨*proof*⟩

lemma *length-array-fields*: *length* (*array-fields n*) = *n*
⟨*proof*⟩

lemma *set-array-fields*: *set* (*array-fields n*) = (⋃ *i*<*n*. {[*replicate i CHR* ''1'']})
⟨*proof*⟩
lemma *find-array-fields-Some*:
find P (*array-fields n*) = *Some y* ⟷
(∃ *i* < *n*. *y* = [*replicate i CHR* ''1''] ∧ *P y* ∧ (∀ *j*<*i*. ¬ *P* ([*replicate j CHR* ''1'']))))
⟨*proof*⟩

lemma *Bex-intvl-conv*: (∃ *x* ∈ {*0*..<*n*::*nat*}. *P x*) ⟷ (∃ *i*. *i* < *n* ∧ *P i*)
⟨*proof*⟩

lemma *Bex-union-conv*: (∃ *x* ∈ *A* ∪ *B*. *P x*) ⟷ ((∃ *x* ∈ *A*. *P x*) ∨ (∃ *x* ∈ *B*. *P x*))
⟨*proof*⟩

lemma *ex-intvl-conj-distribR*: ((∃ *x*∈{*0*..<*n*}. *P x*) ∧ *Q*) ⟷ (∃ *x*∈{*0*..<*n*}. *P x* ∧ *Q*)
⟨*proof*⟩

lemma *ex-less-conj-distribR*: ((∃ *i*<*n*::*nat*. *P i*) ∧ *Q*) ⟷ (∃ *i*<*n*. *P i* ∧ *Q*)
⟨*proof*⟩

lemma *map-filter-map-Some-conv*:
assumes *all-Some*: ⋀*x*. *x* ∈ *set xs* ⟹ *f x* = *Some* (*g x*)
shows *List.map-filter f xs* = *map g xs*
⟨*proof*⟩

lemma *map-filter-map-compose*:
List.map-filter f (*map g xs*) = *List.map-filter* (*f o g*) *xs*
⟨*proof*⟩

lemma *map-filter-fun-eq-conv*:
assumes *all-same*: ⋀*x*. *x* ∈ *set xs* ⟹ *f x* = *g x*
shows *List.map-filter f xs* = *List.map-filter g xs*
⟨*proof*⟩

lemma *map-filter-empty*[*simp*]: *List.map-filter Map.empty xs* = []
⟨*proof*⟩

lemma *map-filter-Some*[*simp*]: *List.map-filter* (λ*x*. *Some* (*f x*)) *xs* = *map f xs*

⟨*proof*⟩

lemma *list-all-field-lookup*[*simp*]:
CARD('*n*) = *m* ⟹
list-all
(λ*f*. ∃ *a b*. *field-lookup* (*typ-uinfo-t TYPE*('*a*::*c-type*['*n*::*finite*])) *f* 0 = *Some* (*a*, *b*))
(*map* (λ*n*. [*replicate n CHR* ''1''])[0..<*m*])
⟨*proof*⟩

lemma *ptr-span-with-stack-byte-type-subset-field*[*simp*]:
∀ *a*∈*ptr-span* (*p*::'*a*::*mem-type ptr*). *root-ptr-valid h* (*PTR*(*stack-byte*) *a*) ⟹
∃ *u*. *field-ti TYPE*('*a*::*mem-type*) *f* = *Some u* ∧ *size-td u* = *sz* ⟹
(∀ *a*∈{&(*p*→*f*)..+*sz*}. *root-ptr-valid h* (*PTR*(*stack-byte*) *a*)) ⟷ *True*
⟨*proof*⟩

lemma (**in** *pointer-lense*) *pointer-lense-field-lvalue*:
assumes *f*: *field-ti TYPE*('*c*::*mem-type*) *f* = *Some u*
and *u*: *size-td u* = *size-of TYPE*('*a*)
shows *pointer-lense* (λ*h* (*p*::'*c ptr*). *r h* (*PTR*('*a*) &(*p*→*f*))) (λ*p*. *w* (*PTR*('*a*) &(*p*→*f*)))
⟨*proof*⟩

lemma *exists-nat-numeral*: ∃ *x*::*nat*. *x* < *numeral k*
⟨*proof*⟩

lemma *fun-upd-eq-cases*: *f*(*p*:=*x*) = *g* ⟷ (*g p* = *x* ∧ (∀ *q*. *p*≠*q* ⟶ *f q* = *g q*))
⟨*proof*⟩

lemma *fun-upd-apply-eq-cases*: (*f*(*p*:=*x*)) *q* = *g q* ⟷ ((*p* = *q* ⟶ *g q* = *x*) ∧ (*p* ≠ *q* ⟶ *f q* = *g q*))
⟨*proof*⟩

lemma *comp-fun-upd-same-fuse*:
(λ*f*. *f*(*p* := *x*)) *o* (λ*f*. *f*(*p* := *y*)) = (λ*f*. *f*(*p*:=*x*))
⟨*proof*⟩

lemma *comp-fun-upd-other-commute*:
p ≠ *q* ⟹ (λ*f*. *f*(*q* := *y*)) *o* (λ*f*. *f*(*p* := *x*)) = (λ*f*. *f*(*p* := *x*)) *o* (λ*f*. *f*(*q* := *y*))
⟨*proof*⟩

lemma *map-td-compose*:
fixes *t*::('*a*, '*b*) *typ-desc*
and *st*::('*a*, '*b*) *typ-struct*
and *ts*::('*a*, '*b*) *typ-tuple list*
and *x*::('*a*, '*b*) *typ-tuple*
shows
map-td f1 g1 (*map-td f2 g2 t*) = *map-td* (λ*n algn*. *f1 n algn o f2 n algn*) (*g1 o g2*) *t* **and**

map-td-struct f1 g1 (map-td-struct f2 g2 st) = map-td-struct (λn algn. f1 n algn o f2 n algn) (g1 o g2) st **and**
map-td-list f1 g1 (map-td-list f2 g2 ts) = map-td-list (λn algn. f1 n algn o f2 n algn) (g1 o g2) ts **and**
map-td-tuple f1 g1 (map-td-tuple f2 g2 x) = map-td-tuple (λn algn. f1 n algn o f2 n algn) (g1 o g2) x
⟨*proof*⟩

lemma (**in** *mem-type*) *distinct-all-field-names*:
distinct (all-field-names (typ-info-t TYPE('a)))
⟨*proof*⟩

lemma *field-lookup-same-type-disjoint*:
⟦*field-lookup t f m = Some (d,n)*;
field-lookup t f' m = Some (d',n'); *f ≠ f'*; *export-uinfo d = export-uinfo d'*;
wf-desc t; *size-td t < addr-card*⟧ ⟹
{(*of-nat n*)::*addr*..+*size-td* (*d*::('*a*,'*b*) *typ-info*)} ∩ {*of-nat n'*..+*size-td d'*} = {} **and**

⟦*field-lookup-struct st f m = Some (d,n)*;
field-lookup-struct st f' m = Some (d',n'); *f ≠ f'*; *export-uinfo d = export-uinfo d'*;
wf-desc-struct st; *size-td-struct st < addr-card* ⟧ ⟹
{(*of-nat n*)::*addr*..+*size-td* (*d*::('*a*,'*b*) *typ-info*)} ∩ {*of-nat n'*..+*size-td d'*} = {} **and**

⟦*field-lookup-list ts f m = Some (d,n)*;
field-lookup-list ts f' m = Some (d',n'); *f ≠ f'*; *export-uinfo d = export-uinfo d'*;
wf-desc-list ts; *size-td-list ts < addr-card*⟧ ⟹
{(*of-nat n*)::*addr*..+*size-td* (*d*::('*a*,'*b*) *typ-info*)} ∩ {*of-nat n'*..+*size-td d'*} = {} **and**
⟦*field-lookup-tuple x f m = Some (d,n)* ;
field-lookup-tuple x f' m = Some (d',n'); *f ≠ f'*; *export-uinfo d = export-uinfo d'*;
wf-desc-tuple x; *size-td-tuple x < addr-card*⟧ ⟹
{(*of-nat n*)::*addr*..+*size-td* (*d*::('*a*,'*b*) *typ-info*)} ∩ {*of-nat n'*..+*size-td d'*} = {}
⟨*proof*⟩

lemma *field-lookup-same-type-u-disjoint*:
⟦*field-lookup t f m = Some (d,n)*;
field-lookup t f' m = Some (d,n'); *f ≠ f'*;
wf-desc t; *size-td t < addr-card*⟧ ⟹
{(*of-nat n*)::*addr*..+*size-td d*} ∩ {*of-nat n'*..+*size-td d*} = {} **and**

⟦*field-lookup-struct st f m = Some (d,n)*;

field-lookup-struct st f′ m = Some (d,n′); f ≠ f′;
wf-desc-struct st; size-td-struct st < addr-card ⟧ ⟹
{(*of-nat n*)::*addr*..+*size-td d*} ∩ {*of-nat n′*..+*size-td d*} = {} **and**

⟦*field-lookup-list ts f m = Some (d,n);*
field-lookup-list ts f′ m = Some (d,n′); f ≠ f′;
wf-desc-list ts; size-td-list ts < addr-card⟧ ⟹
{(*of-nat n*)::*addr*..+*size-td d*} ∩ {*of-nat n′*..+*size-td d*} = {} **and**
⟦*field-lookup-tuple x f m = Some (d,n)* ;
field-lookup-tuple x f′ m = Some (d,n′); f ≠ f′;
wf-desc-tuple x; size-td-tuple x < addr-card⟧ ⟹
{(*of-nat n*)::*addr*..+*size-td d*} ∩ {*of-nat n′*..+*size-td d*} = {}
⟨*proof*⟩

lemma *td-set-list-intvl-sub-nat*:
(*d,n*) ∈ *td-set-list t m* ⟹ {*n*..< *n* +*size-td d*} ⊆ {*m*..< *m* +*size-td-list t*}
⟨*proof*⟩

lemma *td-set-tuple-intvl-sub-nat*:
(*d,n*) ∈ *td-set-tuple t m* ⟹ {*n*..<*n*+*size-td d*} ⊆ {*m*..<*m*+*size-td-tuple t*}
⟨*proof*⟩

lemma *field-lookup-same-type-u-disjoint-nat*:
⟦*field-lookup t f m = Some (d,n);*
field-lookup t f′ m = Some (d,n′); f ≠ f′;
wf-desc t⟧ ⟹
{*n*..<*n* + *size-td d*} ∩ {*n′*..<*n′* + *size-td d*} = {} **and**

⟦*field-lookup-struct st f m = Some (d,n);*
field-lookup-struct st f′ m = Some (d,n′); f ≠ f′;
wf-desc-struct st ⟧ ⟹
{*n*..< *n* + *size-td d*} ∩ {*n′*..< *n′* + *size-td d*} = {} **and**

⟦*field-lookup-list ts f m = Some (d,n);*
field-lookup-list ts f′ m = Some (d,n′); f ≠ f′;
wf-desc-list ts⟧ ⟹
{*n*..<*n* + *size-td d*} ∩ {*n′*..<*n′* + *size-td d*} = {} **and**
⟦*field-lookup-tuple x f m = Some (d,n)* ;
field-lookup-tuple x f′ m = Some (d,n′); f ≠ f′;
wf-desc-tuple x⟧ ⟹
{*n*..<*n* + *size-td d*} ∩ {*n′*..<*n′* +*size-td d*} = {}
⟨*proof*⟩

lemma (**in** *mem-type*) *field-lookup-same-type-disjoint*:
assumes *f*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f m* = *Some* (*t*, *n*)
assumes *f′*: *field-lookup* (*typ-info-t TYPE*('*a*)) *f′ m* = *Some* (*t′*, *n′*)
assumes *neq*: *f* ≠ *f′*
assumes *same*: *export-uinfo t* = *export-uinfo t′*
shows {(*of-nat n*)::*addr*..+*size-td t*} ∩ {*of-nat n′*..+*size-td t′*} = {}

⟨*proof*⟩

lemma (**in** *mem-type*) *field-lookup-same-type-ptr-span-disjoint*:
fixes p::$'a$ *ptr*
assumes f: *field-lookup* (*typ-info-t TYPE*($'a$)) f *0* = *Some* (t, n)
assumes f': *field-lookup* (*typ-info-t TYPE*($'a$)) f' *0* = *Some* (t', n')
assumes *neq*: $f \neq f'$
assumes t: *export-uinfo* t = *typ-uinfo-t* (*TYPE*($'b$::*c-type*))
assumes t': *export-uinfo* t' = *typ-uinfo-t* (*TYPE*($'b$::*c-type*))
shows *ptr-span* (*PTR*($'b$) &(p→f)) ∩ *ptr-span* (*PTR*($'b$) &(p→f')) = {}
⟨*proof*⟩

lemma *sub-type-valid-field-lvalue-overlap*:
fixes p::$'a$::*mem-type ptr*
fixes q::$'b$::*mem-type ptr*
assumes *subtype*: *TYPE*($'b$) ≤$_\tau$ *TYPE*($'a$)
assumes *valid-p*: $d \models_t p$
assumes *overlap*: *ptr-span* p ∩ *ptr-span* $q \neq \{\}$
shows $d \models_t q \longleftrightarrow$
(∃f t n. *field-lookup* (*typ-info-t TYPE*($'a$)) f *0* = *Some* (t, n) ∧
export-uinfo t = *typ-uinfo-t TYPE*($'b$) ∧
q = *PTR*($'b$) (&(p→f)))
⟨*proof*⟩

lemma *field-lookup-intvl-contained-left*:
fixes p::$'a$::*mem-type ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*($'a$)) f *0* = *Some* (t, k)
assumes n: n = *size-td* t
assumes m: m = *size-of TYPE*($'a$)
shows {&(p→f)..+n} ∩ {*ptr-val* p..+m} ≠ {}
⟨*proof*⟩

lemma *field-lookup-intvl-contained-right*:
fixes p::$'a$::*mem-type ptr*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*($'a$)) f *0* = *Some* (t, k)
assumes n: n = *size-td* t
assumes m: m = *size-of TYPE*($'a$)
shows {*ptr-val* p..+m} ∩ {&(p→f)..+n} ≠ {}
⟨*proof*⟩

lemma *field-overlap-right*:
fixes p::$'a$::*mem-type ptr*
assumes *field-lookup*: *field-lookup* (*typ-info-t TYPE*($'a$)) *path 0* = *Some* (t, n)
assumes *match*: *export-uinfo* t = *typ-uinfo-t TYPE*($'f$)
shows (*ptr-span* p) ∩ *ptr-span* (*PTR*($'f$::*c-type*) &(p→*path*)) ≠ {}
⟨*proof*⟩

locale *nested-field′* =
  fixes *t* :: *'a::mem-type xtyp-info*
  fixes *path* :: *string list*
  fixes *sel* :: *'a::mem-type ⇒ 'f::mem-type*
  fixes *upd* :: *'f ⇒ 'a ⇒ 'a*

  assumes *field-ti*: *field-ti TYPE('a) path = Some t*
  assumes *field-typ-match*: *export-uinfo t = typ-uinfo-t TYPE('f)*

  assumes *sel-def*: *sel ≡ from-bytes o access-ti$_0$ t*
  assumes *upd-def*: *upd ≡ update-ti t o to-bytes-p*
begin

lemma *sub-typ*: *TYPE('f) ≤$_\tau$ TYPE('a)*
  ⟨*proof*⟩

lemma *h-val-field*:
  shows *h-val h (PTR('f) &(p→path)) = sel (h-val h p)*
  ⟨*proof*⟩

lemma *clift-field-update*:
  assumes *typed*: *hrs-htd h ⊨$_t$ p*
  shows *clift (hrs-mem-update (heap-update (PTR('f) &(p→path)) x) h) =*
        (clift h)(p↦ upd x (h-val (hrs-mem h) p))
⟨*proof*⟩

lemma *clift-field-update-padding*:
  assumes *typed*: *hrs-htd h ⊨$_t$ p*
  assumes *lbs*: *length bs = size-of TYPE('f)*
  shows *clift (hrs-mem-update (heap-update-padding (PTR('f) &(p→path)) x bs) h) =*
        (clift h)(p↦ upd x (h-val (hrs-mem h) p))
⟨*proof*⟩

lemma *h-val-field-lvalue-update*: *d ⊨$_t$ p ⟹ d ⊨$_t$ q ⟹*
  h-val (heap-update (PTR('f) &(p→path)) x h) q = ((h-val h)(p := upd x (h-val h p))) q
  ⟨*proof*⟩

lemma *h-val-field-lvalue-update-padding*: *d ⊨$_t$ p ⟹ d ⊨$_t$ q ⟹ length bs = size-of TYPE('f) ⟹*
  h-val (heap-update-padding (PTR('f) &(p→path)) x bs h) q = ((h-val h)(p := upd x (h-val h p))) q
  ⟨*proof*⟩

end

lemma *align-addr-card*:
assumes *wf-size-desc*: *wf-size-desc t*
assumes *max-size*: *size-td t* < *addr-card*
assumes *align-size-of*: *2* ^ *align-td t dvd size-td t*
shows *2* ^ *align-td t dvd addr-card*
⟨*proof*⟩

lemma *ptr-aligned-u-field-lookup*:
assumes *wf-size-desc*: *wf-size-desc t*
assumes *wf-align*: *wf-align t*
assumes *align-field*: *align-field t*
assumes *max-size*: *size-td t* < *addr-card*
assumes *align-size-of*: *2* ^ *align-td t dvd size-td t*
assumes *align-size-of-u*: *2* ^ *align-td u dvd size-td u* — does not follow from a well-formedness condition on t
assumes *fl*: *field-lookup t path 0* = *Some* (*u*, *n*)
assumes *ptr-aligned-u*: *ptr-aligned-u t a*
shows *ptr-aligned-u u* (*a* + *of-nat n*)
⟨*proof*⟩

lemma *c-guard-u-field-looup*:
assumes *wf-size-desc*: *wf-size-desc t*
assumes *wf-align*: *wf-align t*
assumes *align-field*: *align-field t*
assumes *max-size*: *size-td t* < *addr-card*
assumes *align-size-of*: *2* ^ *align-td t dvd size-td t*
assumes *align-size-of-u*: *2* ^ *align-td u dvd size-td u* — does not follow from a well-formedness condition on t
assumes *fl*: *field-lookup t path 0* = *Some* (*u*, *n*)
assumes *c-guard-u*: *c-guard-u t a*
shows *c-guard-u u* (*a* + *of-nat n*)
⟨*proof*⟩

lemma *cvalid-u-field-lookup*:
assumes *wf-size-desc*: *wf-size-desc t*
assumes *wf-align*: *wf-align t*
assumes *align-field*: *align-field t*
assumes *max-size*: *size-td t* < *addr-card*
assumes *align-size-of*: *2* ^ *align-td t dvd size-td t*
assumes *align-size-of-u*: *2* ^ *align-td u dvd size-td u* — does not follow from a well-formedness condition on t
assumes *fl*: *field-lookup t path 0* = *Some* (*u*, *n*)
assumes *cvalid-u*: *cvalid-u t d a*
shows *cvalid-u u d* (*a* + *of-nat n*)
⟨*proof*⟩

locale *mem-type-u* =

fixes *t*::*typ-uinfo*
assumes *wf-desc*: *wf-desc t*
assumes *wf-size-desc*: *wf-size-desc t*
assumes *wf-align*: *wf-align t*
assumes *align-field*: *align-field t*
assumes *max-size*: *size-td t* < *addr-card*
assumes *align-size*: *2* ^ *align-td t dvd size-td t*
begin
lemmas *ptr-aligned-u-field-lookup* = *ptr-aligned-u-field-lookup* [*OF wf-size-desc wf-align align-field max-size align-size*]
lemmas *c-guard-u-field-looup* = *c-guard-u-field-looup* [*OF wf-size-desc wf-align align-field max-size align-size*]
lemmas *cvalid-u-field-lookup* = *cvalid-u-field-lookup* [*OF wf-size-desc wf-align align-field max-size align-size*]

end

lemma *wf-size-desc-export-uinfo*:
fixes *t*::('*a*, '*b*) *typ-info*
and *st*::('*a*, '*b*) *typ-info-struct*
and *ts*::('*a*, '*b*) *typ-info-tuple list*
and *x*::('*a*, '*b*) *typ-info-tuple*
shows
wf-size-desc t ⟹ *wf-size-desc* (*export-uinfo t*) **and**
wf-size-desc-struct st ⟹ *wf-size-desc-struct* (*map-td-struct field-norm* (λ-. ()) *st*) **and**
wf-size-desc-list ts ⟹ *wf-size-desc-list* (*map-td-list field-norm* (λ-. ()) *ts*) **and**
wf-size-desc-tuple x ⟹ *wf-size-desc-tuple* (*map-td-tuple field-norm* (λ-. ()) *x*)
⟨*proof*⟩

context *mem-type*
begin

lemma *typ-uinfo-t-mem-type*[*simp*]: *mem-type-u* (*typ-uinfo-t*(*TYPE*('*a*)))
⟨*proof*⟩

sublocale *u*: *mem-type-u typ-uinfo-t*(*TYPE*('*a*))
⟨*proof*⟩

end

lemma *field-names-u-composeI*:
assumes *wf-t*: *wf-desc t*
assumes *path1*: *path1* ∈ *set* (*field-names-u t u*)
assumes *path2*: *path2* ∈ *set* (*field-names-u u v*)

shows *path1 @ path2 ∈ set (field-names-u t v)*
⟨*proof*⟩

lemma *field-names-u-composeD*:
assumes *wf-t*: *wf-desc t*
assumes *append*: *path1 @ path2 ∈ set (field-names-u t v)*
shows ∃ *u*. *path1 ∈ set (field-names-u t u) ∧ path2 ∈ set (field-names-u u v)*
⟨*proof*⟩

lemma *field-names-u-field-offset-untyped-append*:
assumes *wf-desc root*
assumes *path1*: *path1 ∈ set (field-names-u root outer)*
assumes *path2*: *path2 ∈ set (field-names-u outer inner)*
shows *field-offset-untyped root (path1 @ path2) = field-offset-untyped root path1 + field-offset-untyped outer path2*
⟨*proof*⟩

lemma *root-ptr-valid-u-cases*:
assumes *root-ptr-valid-u t1 d a1*
assumes *root-ptr-valid-u t2 d a2*
shows *(t1 = t2 ∧ a1 = a2) ∨ ({a1 ..+ size-td t1} ∩ {a2 ..+ size-td t2} = {})*
⟨*proof*⟩

lemma *field-offset-untyped-empty*[*simp*]: *field-offset-untyped t [] = 0*
⟨*proof*⟩

lemma *field-names-u-refl*[*simp*]: *field-names-u t t = [[]]*
⟨*proof*⟩

lemma *field-names-u-size-td-bounds*:
assumes *wf-t*: *wf-desc t*
assumes *path*: *path ∈ set (field-names-u t u)*
shows *field-offset-untyped t path + size-td u ≤ size-td t*
⟨*proof*⟩

lemma *field-lookup-path-cases*:
assumes *fl1*: *field-lookup t path 0 = Some (u, n)*
shows *(t = u ∧ path = []) ∨ ((t ≠ u) ∧ path ≠ [])*
⟨*proof*⟩

lemma *field-lookup-cycle-cases*:
assumes *fl1*: *field-lookup t path1 0 = Some (u, n1)*
assumes *fl2*: *field-lookup u path2 0 = Some (v, n2)*
shows *(t = v ∧ u = v ∧ path1 = [] ∧ path2 = [] ∧ n1 = 0 ∧ n2 = 0) ∨ (t ≠ v)*
⟨*proof*⟩

lemma *field-lookup-refl-iff*: *field-lookup t p n = Some (t, m)* $\longleftrightarrow$ *p = [] ∧ n = m*
⟨*proof*⟩

lemma *valid-root-footprint-contained-sub-typ*:
assumes *valid-root-x*: *valid-root-footprint d x t*
assumes *valid-y*: *valid-footprint d y s*
assumes *contained*: *{y ..+ size-td s} ⊆ {x ..+ size-td t}*
shows *s ≤ t*
⟨*proof*⟩

lemma *c-null-guard-u-no-overlow*: *c-null-guard-u t a ⟹ unat a + size-td t ≤ addr-card*
⟨*proof*⟩

lemma *c-null-guard-u-size-td-limit*: *c-null-guard-u t a ⟹ size-td t < addr-card*
⟨*proof*⟩

lemma *c-null-guard-u-intvl-nat-conv*:
assumes *c-null-guard*: *c-null-guard-u t a*
shows *{a ..+ size-td t} = {x. (unat a ≤ unat x ∧ unat x < (unat a + size-td t))}*
⟨*proof*⟩

lemma *valid-footprint-overlap-sub-typ*:
assumes *valid-x*: *valid-footprint d x t*
assumes *valid-y*: *valid-footprint d y s*
assumes *overlap*: *{x ..+ size-td t} ∩ {y ..+ size-td s} ≠ {}*
shows *s ≤ t ∨ t ≤ s*
⟨*proof*⟩

lemma *valid-footprint-overlap-sub-typ-cases* [*consumes 3*, *case-names eq less1 less2*]:
assumes *valid-x*: *valid-footprint d x t*
assumes *valid-y*: *valid-footprint d y s*
assumes *overlap*: *{x ..+ size-td t} ∩ {y ..+ size-td s} ≠ {}*
assumes *eq*: *s = t ⟹ P*
assumes *less-s-t*: *s < t ⟹ P*
assumes *less-t-s*: *t < s ⟹ P*
shows *P*
⟨*proof*⟩

lemma *valid-footprint-contained-sub-typ*:
assumes *valid-x*: *valid-footprint d x t*
assumes *valid-y*: *valid-footprint d y s*
assumes *contained*: *{x ..+ size-td t} ⊆ {y ..+ size-td s}*
shows *s ≤ t ∨ t ≤ s*
⟨*proof*⟩

lemma *valid-footprint-contained-sub-typ-cases*:
assumes *valid-x*: *valid-footprint d x t*
assumes *valid-y*: *valid-footprint d y s*
assumes *contained*: $\{x\ ..+\ size\text{-}td\ t\} \subseteq \{y\ ..+\ size\text{-}td\ s\}$
assumes *eq*: $s = t \Longrightarrow P$
assumes *less-s-t*: $s < t \Longrightarrow P$
assumes *less-t-s*: $t < s \Longrightarrow P$
shows *P*
⟨*proof*⟩

lemma *all-field-names-field-lookup'*:
fixes *t*::('*a*, '*b*) *typ-desc*
and *st*::('*a*, '*b*) *typ-struct*
and *ts*::('*a*, '*b*) *typ-tuple list*
and *x*::('*a*, '*b*) *typ-tuple*
shows
wf-desc t $\Longrightarrow$ *f* $\in$ *set* (*all-field-names t*) $\Longrightarrow$ $\exists$ *u n*. *field-lookup t f 0* = *Some* (*u*, *n*) **and**
wf-desc-struct st $\Longrightarrow$ *f* $\in$ *set* (*all-field-names-struct st*) $\Longrightarrow$ $\exists$ *u n*. *field-lookup-struct st f 0* = *Some* (*u*, *n*) **and**
wf-desc-list ts $\Longrightarrow$ *f* $\in$ *set* (*all-field-names-list ts*) $\Longrightarrow$ $\exists$ *u n*. *field-lookup-list ts f 0* = *Some* (*u*, *n*) **and**
wf-desc-tuple x $\Longrightarrow$ *f* $\in$ *set* (*all-field-names-tuple x*) $\Longrightarrow$ $\exists$ *u n*. *field-lookup-tuple x f 0* = *Some* (*u*, *n*)
⟨*proof*⟩

lemma *valid-foot-print-intvl-self*: *valid-footprint d a t* $\Longrightarrow$ *a* $\in$ $\{a\ ..+\ size\text{-}td\ t\}$
⟨*proof*⟩

lemma *field-lookup-Some-field-names-u*:
fixes *t* :: *typ-uinfo* **and**
st :: *typ-uinfo-struct* **and**
ts :: *typ-uinfo-tuple list* **and**
x :: *typ-uinfo-tuple*
shows
field-lookup t f n = *Some* (*s*, *m*) $\Longrightarrow$ *f* $\in$ *set* (*field-names-u t s*)
field-lookup-struct st f n = *Some* (*s*, *m*) $\Longrightarrow$ *f* $\in$ *set* (*field-names-struct-u st s*)
field-lookup-list ts f n = *Some* (*s*, *m*) $\Longrightarrow$ *f* $\in$ *set* (*field-names-list-u ts s*)
field-lookup-tuple x f n = *Some* (*s*, *m*) $\Longrightarrow$ *f* $\in$ *set* (*field-names-tuple-u x s*)
⟨*proof*⟩

lemma *field-lookup-non-prefix-disj'*:
assumes *wf*: *mem-type-u t*
assumes *f*: *field-lookup t f 0* = *Some* (*tf*, *n*)
assumes *g*: *field-lookup t g 0* = *Some* (*tg*, *m*)
assumes *f-g*: *disj-fn f g*
shows *disjnt* {(*of-nat n*::*addr*) ..+ *size-td tf*} {*of-nat m* ..+ *size-td tg*}
⟨*proof*⟩

lemma *sub-typ-stack-byte-u*:
$t \leq$ *typ-uinfo-t* (*TYPE*(*stack-byte*)) $\Longrightarrow$ $t =$ *typ-uinfo-t TYPE*(*stack-byte*)
⟨*proof*⟩

lemma *root-ptr-valid-not-subtype-disjoint-u*:
⟦ *root-ptr-valid d* (*p*::'*a*::*mem-type ptr*);
cvalid-u t d q;
¬ $t \leq$ *typ-uinfo-t TYPE*('*a*) ⟧ $\Longrightarrow$
ptr-span p ∩ {*q* ..+ *size-td t*} = {}
⟨*proof*⟩

lemma *stack-allocs-disjoint-u*:
assumes *stack-alloc*: (*p*, *d'*) ∈ *stack-allocs n* $\mathcal{S}$ (*TYPE*('*a*::*mem-type*)) *d*
assumes *no-stack*: $t \neq$ *typ-uinfo-t* (*TYPE*(*stack-byte*))
assumes *typed*: *cvalid-u t d q*
shows {*ptr-val p* ..+ *n* ∗ *size-of TYPE*('*a*)} ∩ {*q* ..+ *size-td t*} = {}
⟨*proof*⟩

lemma *fold-ptr-retyp-disjoint-u*:
fixes *p*::'*a*::*mem-type ptr*
shows ⟦ *cvalid-u t d q*; {*ptr-val p*..+ *n* ∗ *size-of TYPE*('*a*)} ∩ {*q* ..+ *size-td t*}
= {} ⟧ $\Longrightarrow$
cvalid-u t (*fold* (λ*i*. *ptr-retyp* (*p* +$_p$ *int i*)) [*0*..<*n*] *d*) *q*
⟨*proof*⟩

lemma *fold-ptr-force-type-disjoint-u*:
fixes *p*::'*a*::*mem-type ptr*
shows ⟦ *cvalid-u t d q*; {*ptr-val p*..+ *n* ∗ *size-of TYPE*('*a*)} ∩ {*q* ..+ *size-td t*}
= {} ⟧ $\Longrightarrow$
cvalid-u t (*fold* (λ*i*. *ptr-force-type* (*p* +$_p$ *int i*)) [*0*..<*n*] *d*) *q*
⟨*proof*⟩

lemma *fold-ptr-retyp-disjoint2-u*:
fixes *p*::'*a*::*mem-type ptr*
assumes *no-overflow*: *0* ∉ {*ptr-val p*..+ *n* ∗ *size-of TYPE*('*a*)}
shows ⟦*cvalid-u t* (*fold* (λ*i*. *ptr-retyp* (*p* +$_p$ *int i*)) [*0*..<*n*] *d*) *q*;
{*ptr-val p*..+ *n* ∗ *size-of TYPE*('*a*)} ∩ {*q* ..+ *size-td t*}= {} ⟧
$\Longrightarrow$ *cvalid-u t d q*
⟨*proof*⟩

lemma *fold-ptr-force-type-disjoint2-u*:
fixes *p*::'*a*::*mem-type ptr*
assumes *no-overflow*: *0* ∉ {*ptr-val p*..+ *n* ∗ *size-of TYPE*('*a*)}
shows ⟦*cvalid-u t* (*fold* (λ*i*. *ptr-force-type* (*p* +$_p$ *int i*)) [*0*..<*n*] *d*) *q*;
{*ptr-val p*..+ *n* ∗ *size-of TYPE*('*a*)} ∩ {*q* ..+ *size-td t*}= {} ⟧
$\Longrightarrow$ *cvalid-u t d q*
⟨*proof*⟩

lemma *fold-ptr-retyp-disjoint-iff-u*:
fixes *p*::*'a*::*mem-type ptr*
assumes *no-overflow*: *0* $\notin$ {*ptr-val p..+ n * size-of TYPE('a)*}
shows {*ptr-val p..+ n * size-of TYPE('a)*} $\cap$ {*q ..+ size-td t*} = {}
$\Longrightarrow$ *cvalid-u t* (*fold* (λ*i. ptr-retyp* (*p* +$_p$ *int i*)) [*0..<n*] *d*) *q* = *cvalid-u t d q*
⟨*proof*⟩

lemma *fold-ptr-force-type-disjoint-iff-u*:
fixes *p*::*'a*::*mem-type ptr*
assumes *no-overflow*: *0* $\notin$ {*ptr-val p..+ n * size-of TYPE('a)*}
shows {*ptr-val p..+ n * size-of TYPE('a)*} $\cap$ {*q ..+ size-td t*} = {}
$\Longrightarrow$ *cvalid-u t* (*fold* (λ*i. ptr-force-type* (*p* +$_p$ *int i*)) [*0..<n*] *d*) *q* = *cvalid-u t d q*
⟨*proof*⟩

lemma *stack-allocs-preserves-typing-u*:
assumes *stack-alloc*: (*p*, *d'*) $\in$ *stack-allocs n* $\mathcal{S}$ (*TYPE*(*'a*::*mem-type*)) *d*
assumes *no-stack*: *t* $\neq$ *typ-uinfo-t* (*TYPE*(*stack-byte*))
assumes *typed*: *cvalid-u t d q*
shows *cvalid-u t d' q*
⟨*proof*⟩

lemma *stack-allocs-root-ptr-valid-u-new-cases*:
assumes *stack-alloc*: (*p*, *d'*) $\in$ *stack-allocs n* $\mathcal{S}$ (*TYPE*(*'a*::*mem-type*)) *d*
assumes *valid*: *root-ptr-valid-u t d' q*
shows ($\exists$ *i*<*n*. *q* = *ptr-val* (*p* +$_p$ *int i*) $\wedge$ *t* = *typ-uinfo-t* (*TYPE*(*'a*))) $\vee$ *root-ptr-valid-u t d q*
⟨*proof*⟩

lemma *cvalid-u-c-guard-u*:
cvalid-u t d a $\Longrightarrow$ *c-guard-u t a*
⟨*proof*⟩

lemma *stack-allocs-preserves-root-ptr-valid-u*:
assumes *stack-alloc*: (*p*, *d'*) $\in$ *stack-allocs n* $\mathcal{S}$ (*TYPE*(*'a*::*mem-type*)) *d*
assumes *no-stack*: *t* $\neq$ *typ-uinfo-t* (*TYPE*(*stack-byte*))
assumes *typed*: *root-ptr-valid-u t d q*
shows *root-ptr-valid-u t d' q*
⟨*proof*⟩

lemma *stack-allocs-root-ptr-valid-u-other*:
assumes *stack-alloc*: (*p*, *d'*) $\in$ *stack-allocs n* $\mathcal{S}$ (*TYPE*(*'a*::*mem-type*)) *d*
assumes *valid-d*: *root-ptr-valid-u t d q*
assumes *non-stack*: *t* $\neq$ *typ-uinfo-t* (*TYPE*(*stack-byte*))
shows *root-ptr-valid-u t d' q*
⟨*proof*⟩

lemma *stack-allocs-root-ptr-valid-u-same*:
assumes *stack-alloc*: (*p*, *d'*) $\in$ *stack-allocs n* $\mathcal{S}$ (*TYPE*(*'a*::*mem-type*)) *d*

assumes *i*: $i < n$
assumes *addr-eq*: $q = ptr\text{-}val\ (p +_p int\ i)$
assumes *match*: $t = typ\text{-}uinfo\text{-}t\ (TYPE('a))$
shows *root-ptr-valid-u t d′ q*
⟨*proof*⟩

lemma *stack-allocs-root-ptr-valid-u-cases*:
assumes *stack-alloc*: $(p,\ d') \in stack\text{-}allocs\ n\ \mathcal{S}\ (TYPE('a::mem\text{-}type))\ d$
assumes *non-stack-byte*: $t \neq typ\text{-}uinfo\text{-}t\ (TYPE(stack\text{-}byte))$
shows *root-ptr-valid-u t d′ q* ⟷
$(\exists\, i{<}n.\ q = ptr\text{-}val\ (p +_p int\ i) \wedge t = typ\text{-}uinfo\text{-}t\ (TYPE('a))) \vee$
root-ptr-valid-u t d q

⟨*proof*⟩

lemma *stack-releases-root-ptr-valid-u1*:
fixes *p*::′*a*::*mem-type ptr*
assumes *non-stack-p*: $typ\text{-}uinfo\text{-}t\ TYPE('a) \neq typ\text{-}uinfo\text{-}t\ TYPE(stack\text{-}byte)$
assumes *non-stack-q*: $t \neq typ\text{-}uinfo\text{-}t\ TYPE(stack\text{-}byte)$
assumes *root-q*: *root-ptr-valid-u t* (*stack-releases n p d*) *q*
shows $\{ptr\text{-}val\ p\ ..+\ n * size\text{-}of\ TYPE('a::mem\text{-}type)\} \cap \{q\ ..+\ size\text{-}td\ t\} = \{\}$
$\wedge$ *root-ptr-valid-u t d q*
⟨*proof*⟩

lemma *stack-releases-root-ptr-valid-u2*:
fixes *p*::′*a*::*mem-type ptr*
assumes *disj*: $\{ptr\text{-}val\ p\ ..+\ n * size\text{-}of\ TYPE('a::mem\text{-}type)\} \cap \{q\ ..+\ size\text{-}td\ t\} = \{\}$
assumes *valid-q*: *root-ptr-valid-u t d q*
shows *root-ptr-valid-u t* (*stack-releases n p d*) *q*
⟨*proof*⟩

lemma *stack-release-root-ptr-valid-u2*:
fixes *p*::′*a*::*mem-type ptr*
assumes *disj*: $ptr\text{-}span\ p \cap \{q\ ..+\ size\text{-}td\ t\} = \{\}$
assumes *valid-q*: *root-ptr-valid-u t d q*
shows *root-ptr-valid-u t* (*stack-release p d*) *q*
⟨*proof*⟩

lemma *stack-releases-root-ptr-valid-u-cases*:
fixes *p*::′*a*::*mem-type ptr*
assumes *non-stack-p*: $typ\text{-}uinfo\text{-}t\ TYPE('a) \neq typ\text{-}uinfo\text{-}t\ TYPE(stack\text{-}byte)$
assumes *non-stack-q*: $t \neq typ\text{-}uinfo\text{-}t\ TYPE(stack\text{-}byte)$
shows *root-ptr-valid-u t* (*stack-releases n p d*) *q* ⟷
$\{ptr\text{-}val\ p\ ..+\ n * size\text{-}of\ TYPE('a::mem\text{-}type)\} \cap \{q\ ..+\ size\text{-}td\ t\} = \{\} \wedge$
root-ptr-valid-u t d q
⟨*proof*⟩

lemma *valid-root-footprint-is-root*:

assumes *wf*: *wf-desc t*
assumes *wf-size*: *wf-size-desc t*
assumes *f*: *field-lookup t path 0 = Some* (*s*, *n*)
assumes *footprint-t*: *valid-footprint d a t*
assumes *root-s*: *valid-root-footprint d* (*a* + *of-nat n*) *s*
shows (*t* = *s* ∧ *path* = [])
⟨*proof*⟩

corollary *root-ptr-valid-u-is-root*:
assumes *wf*: *wf-desc t*
assumes *wf-size*: *wf-size-desc t*
assumes *f*: *field-lookup t path 0 = Some* (*s*, *n*)
assumes *cvalid-u-t*: *cvalid-u t d a*
assumes *root-s*: *root-ptr-valid-u s d* (*a* + *of-nat n*)
shows (*t* = *s* ∧ *path* = [])
⟨*proof*⟩

lemma *valid-footprint-field-lookup*:
assumes *wf*: *wf-desc t*
assumes *wf-size-desc*: *wf-size-desc t*
assumes *valid-t*: *valid-footprint d a t*
assumes *fl*: *field-lookup t path1 0 = Some* (*s*, *n*)
shows *valid-footprint d* (*a* + *word-of-nat n*) *s*
⟨*proof*⟩

lemma *in-set-mapD*:*x* ∈ *set* (*map f xs*) ⟹ ∃ *y* ∈ *set xs*. *x* = *f y*
⟨*proof*⟩

lemma *findSomeI*: *x* ∈ *set xs* ⟹ *P x* ⟹ ∃ *x*. *find P xs* = *Some x*
⟨*proof*⟩

lemma *find-in-set-inj-distinct*:
inj-on P (*set* (*map fst xs*)) ⟹ (*x*, *v*) ∈ *set xs* ⟹ *P x* ⟹ *distinct* (*map fst xs*)
⟹
find (λ(*x*, -). *P x*) *xs* = *Some* (*x*, *v*)
⟨*proof*⟩

definition *inj-on-true P A* = (∀ *x y*. *x* ∈ *A* ⟶ *y* ∈ *A* ⟶ *P x* ⟶ *P y* ⟶ *x* = *y*)

lemma *inj-on P A* ⟹ *inj-on-true P A*
⟨*proof*⟩

lemma *inj-on-trueD*: *inj-on-true P A* ⟹ *x* ∈ *A* ⟹ *y* ∈ *A* ⟹ *P x* ⟹ *P y* ⟹ *x* = *y*
⟨*proof*⟩

lemma *inj-on-trueI*: (⋀*x y*. *x* ∈ *A* ⟹ *y* ∈ *A* ⟹ *P x* ⟹ *P y* ⟹ *x* = *y*) ⟹

inj-on-true P A
⟨*proof*⟩

lemma *inj-on-true-mono*: $A \subseteq B \Longrightarrow$ *inj-on-true P B* $\Longrightarrow$ *inj-on-true P A*
⟨*proof*⟩

lemma *find-in-set-inj-on-true-distinct*:
inj-on-true P (*set* (*map fst xs*)) $\Longrightarrow$ $(x, v) \in$ *set xs* $\Longrightarrow$ *P x* $\Longrightarrow$ *distinct* (*map fst xs*) $\Longrightarrow$
find ($\lambda(x, \text{-}).$ *P x*) *xs* = *Some* (x, v)
⟨*proof*⟩

lemma *find-in-set-inj-on-true-distinct′*:
inj-on-true P (*set xs*) $\Longrightarrow$ *x* ∈ *set xs* $\Longrightarrow$ *P x* $\Longrightarrow$ *distinct xs* $\Longrightarrow$
find ($\lambda x.$ *P x*) *xs* = *Some x*
⟨*proof*⟩

lemma *typ-tag-le-size-le*: $t <$ (*u*::*typ-uinfo*) $\Longrightarrow$ *size t* < *size u*
⟨*proof*⟩

lemma *c-null-guard-intvl-nat-conv*:
fixes *p*::′*a*::*mem-type ptr*
assumes *c-null-guard*: *c-null-guard p*
shows *ptr-span p* = {*x*. (*unat* (*ptr-val p*) ≤ *unat x* ∧ *unat x* < (*unat* (*ptr-val p*) + *size-of TYPE*(′*a*)))}
⟨*proof*⟩

lemma *c-null-guard-ptr-add*:
fixes *p*::′*a*::*mem-type ptr*
assumes *bound*: *unat* (*ptr-val p*) + *Suc n* ∗ *size-of TYPE*(′*a*) ≤ *addr-card*
assumes *not-null*: *ptr-val p* ≠ *0*
assumes *le*: *i* ≤ *n*
shows *c-null-guard* (*p* +$_p$ *int i*)
⟨*proof*⟩

lemma *ptr-add-disjoint-index*:
fixes *p*::′*a*::*mem-type ptr*
assumes *bound*: *unat* (*ptr-val p*) + *Suc n* ∗ *size-of TYPE*(′*a*) ≤ *addr-card*
assumes *le*: *i* < *n*
assumes *n*: *c-null-guard* (*p* +$_p$ *int n*)
assumes *i*: *c-null-guard* (*p* +$_p$ *int i*)
shows *ptr-span* (*p* +$_p$ *int n*) ∩ *ptr-span* (*p* +$_p$ *int i*) = {}
⟨*proof*⟩

lemma *ptr-add-disjoint-last*:
fixes *p*::′*a*::*mem-type ptr*
assumes *bound*: *unat* (*ptr-val p*) + *Suc n* ∗ *size-of TYPE*(′*a*) ≤ *addr-card*
assumes *not-null*: *ptr-val p* ≠ *0*

shows $\{ptr\text{-}val\ p..+n * size\text{-}of\ TYPE('a)\} \cap ptr\text{-}span\ (p +_p int\ n) = \{\}$
⟨*proof*⟩

lemma *h-val-fold*:
fixes $p::'a::mem\text{-}type\ ptr$
assumes *bound*: $unat\ (ptr\text{-}val\ p) + n * size\text{-}of\ TYPE('a) \leq addr\text{-}card$
assumes *not-null*: $ptr\text{-}val\ p \neq 0$
assumes *i-bound*: $i < n$
shows*h-val* $((fold\ (\lambda i.\ heap\text{-}update\ (p +_p int\ i)\ v)\ [0..<n])\ h)\ (p +_p int\ i) = v$
⟨*proof*⟩

lemma *h-val-fold-disjoint*:
fixes $p::\ 'a::mem\text{-}type\ ptr$
fixes $q::\ 'b::mem\text{-}type\ ptr$
assumes *disj*: $\{ptr\text{-}val\ p..+n * size\text{-}of\ TYPE('a)\} \cap ptr\text{-}span\ q = \{\}$
shows $h\text{-}val\ (fold\ (\lambda i.\ heap\text{-}update\ (p +_p int\ i)\ (v\ i))\ [0..<n]\ h)\ q =$
$h\text{-}val\ h\ q$
⟨*proof*⟩

lemma *h-val-fold-zero-disjoint*:
fixes $p::\ 'a::mem\text{-}type\ ptr$
fixes $q::\ 'b::mem\text{-}type\ ptr$
assumes *disj*: $\{ptr\text{-}val\ p..+n * size\text{-}of\ TYPE('a)\} \cap ptr\text{-}span\ q = \{\}$
shows $h\text{-}val\ (fold\ (\lambda i.\ heap\text{-}update\ (p +_p int\ i)\ c\text{-}type\text{-}class.zero)\ [0..<n]\ h)\ q =$
$h\text{-}val\ h\ q$
⟨*proof*⟩

lemma *fold-heap-update-commute*:
fixes $p::'a::\ mem\text{-}type\ ptr$
fixes $q::'b::\ mem\text{-}type\ ptr$
assumes *disjoint*: $\{ptr\text{-}val\ p\ ..+\ n * size\text{-}of\ TYPE('a)\} \cap ptr\text{-}span\ q = \{\}$
shows $fold\ (\lambda i.\ heap\text{-}update\ (p +_p int\ i)\ (v\ i))\ [0..<n]\ (heap\text{-}update\ q\ x\ h) =$
$(heap\text{-}update\ q\ x)\ (fold\ (\lambda i.\ heap\text{-}update\ (p +_p int\ i)\ (v\ i))\ [0..<n]\ h)$
⟨*proof*⟩

lemma *fold-heap-update-padding-commute*:
fixes $p::'a::\ mem\text{-}type\ ptr$
fixes $q::'b::\ mem\text{-}type\ ptr$
assumes *disjoint*: $\{ptr\text{-}val\ p\ ..+\ n * size\text{-}of\ TYPE('a)\} \cap ptr\text{-}span\ q = \{\}$
assumes *lbs*: $\bigwedge i.\ i < n \Longrightarrow length\ (bs\ i) = size\text{-}of\ TYPE('a)$
assumes *lbs'*: $length\ bs' = size\text{-}of\ TYPE('b)$
shows $fold\ (\lambda i.\ heap\text{-}update\text{-}padding\ (p +_p int\ i)\ (v\ i)\ (bs\ i))\ [0..<n]\ (heap\text{-}update\text{-}padding$
$q\ x\ bs'\ h) =$
$(heap\text{-}update\text{-}padding\ q\ x\ bs')\ (fold\ (\lambda i.\ heap\text{-}update\text{-}padding\ (p +_p int\ i)$
$(v\ i)\ (bs\ i))\ [0..<n]\ h)$
⟨*proof*⟩

lemma *fold-heap-update-padding-heap-update-commute*:
fixes $p::'a::\ mem\text{-}type\ ptr$

fixes $q::'b::$ *mem-type ptr*
assumes *disjoint*: $\{ptr\text{-}val\ p\ ..+\ n * size\text{-}of\ TYPE('a)\} \cap ptr\text{-}span\ q = \{\}$
assumes *lbs*: $\bigwedge i.\ i < n \Longrightarrow length\ (bs\ i) = size\text{-}of\ TYPE('a)$
shows *fold* $(\lambda i.\ heap\text{-}update\text{-}padding\ (p +_p int\ i)\ (v\ i)\ (bs\ i))\ [0..<n]\ (heap\text{-}update\ q\ x\ h) =$
$(heap\text{-}update\ q\ x)\ (fold\ (\lambda i.\ heap\text{-}update\text{-}padding\ (p +_p int\ i)\ (v\ i)\ (bs\ i))\ [0..<n]\ h)$
⟨*proof*⟩

lemma *fold-heap-update-collapse*:
fixes $p::'a::xmem\text{-}type\ ptr$
assumes *bound*: $unat\ (ptr\text{-}val\ p) + n * size\text{-}of\ TYPE('a) \leq addr\text{-}card$
assumes *not-null*: $ptr\text{-}val\ p \neq 0$
shows $fold\ (\lambda i.\ heap\text{-}update\ (p +_p int\ i)\ (v\ i))\ [0..<n]$
$((fold\ (\lambda i.\ heap\text{-}update\ (p +_p int\ i)\ (w\ i))\ [0..<n])\ h) =$
$fold\ (\lambda i.\ heap\text{-}update\ (p +_p int\ i)\ (v\ i))\ [0..<n]\ h$
⟨*proof*⟩

lemma *fold-heap-update-padding-collapse*:
fixes $p::'a::xmem\text{-}type\ ptr$
assumes *bound*: $unat\ (ptr\text{-}val\ p) + n * size\text{-}of\ TYPE('a) \leq addr\text{-}card$
assumes *not-null*: $ptr\text{-}val\ p \neq 0$
assumes *lbs*: $\bigwedge i.\ i < n \Longrightarrow length\ (bs\ i) = size\text{-}of\ TYPE('a)$
assumes *lbs'*: $\bigwedge i.\ i < n \Longrightarrow length\ (bs'\ i) = size\text{-}of\ TYPE('a)$
shows $fold\ (\lambda i.\ heap\text{-}update\text{-}padding\ (p +_p int\ i)\ (v\ i)\ (bs\ i))\ [0..<n]$
$((fold\ (\lambda i.\ heap\text{-}update\text{-}padding\ (p +_p int\ i)\ (w\ i)\ (bs'\ i))\ [0..<n])\ h) =$
$fold\ (\lambda i.\ heap\text{-}update\text{-}padding\ (p +_p int\ i)\ (v\ i)\ (bs\ i))\ [0..<n]\ h$
⟨*proof*⟩

lemma *fold-heap-update-padding-heap-update-collapse*:
fixes $p::'a::xmem\text{-}type\ ptr$
assumes *bound*: $unat\ (ptr\text{-}val\ p) + n * size\text{-}of\ TYPE('a) \leq addr\text{-}card$
assumes *not-null*: $ptr\text{-}val\ p \neq 0$
assumes *lbs*: $\bigwedge i.\ i < n \Longrightarrow length\ (bs\ i) = size\text{-}of\ TYPE('a)$
shows $fold\ (\lambda i.\ heap\text{-}update\text{-}padding\ (p +_p int\ i)\ (v\ i)\ (bs\ i))\ [0..<n]$
$((fold\ (\lambda i.\ heap\text{-}update\ (p +_p int\ i)\ (w\ i)\ )\ [0..<n])\ h) =$
$fold\ (\lambda i.\ heap\text{-}update\text{-}padding\ (p +_p int\ i)\ (v\ i)\ (bs\ i))\ [0..<n]\ h$
⟨*proof*⟩

21.1 Open Types

named-theorems $\mathcal{T}$*-def* **and**
ptr-valid-definition **and** *fold-ptr-valid* **and** *ptr-valid* **and** *ptr-valid-u-recursion* **and**
addressable-ptr-valid-field **and** *plift-simps* **and** *stack-the-default-plift* **and**
plift-heap-update-padding-heap-update-conv **and**
plift-heap-update-list-stack-byte

lemma *list-ex-array-fields*[*simp*]:

list-ex P (array-fields m) $\longleftrightarrow$ ($\exists$ *n*<*m. P* [*replicate n CHR ''1''*])
⟨*proof*⟩

lemma *list-all-field-lookup-array-fields*[*simp*]:
CARD('*n*) = *m* $\Longrightarrow$
list-all
(λ*f*. $\exists$ *a b. field-lookup* (*typ-uinfo-t TYPE*('*a*::*c-type*['*n*::*finite*])) *f 0* = *Some* (*a*, *b*))
(*array-fields m*)
⟨*proof*⟩

lemma *list-ex-subst-def*:
c ≡ *xs* $\Longrightarrow$ *list-ex P c* = *list-ex P xs*
⟨*proof*⟩

lemma *ex-field-lookup-append*:
wf-desc t $\Longrightarrow$
field-lookup t p1 n1 = *Some* (*u*, *m1*) $\Longrightarrow$
field-lookup u p2 n2 = *Some* (*v*, *m2*) $\Longrightarrow$
$\exists$ *m. field-lookup t* (*p1* @ *p2*) *n3* = *Some* (*v*, *m*)
⟨*proof*⟩

named-theorems *addressable-field-exec*

locale *open-types* =
fixes $\mathcal{T}$:: (*typ-uinfo* × *qualified-field-name list*) *list* — toplevel addressable fields of a type
assumes *mem-type-u*: $\bigwedge$*t fs*. (*t*, *fs*) ∈ *set* $\mathcal{T}$ $\Longrightarrow$ *mem-type-u t*
assumes *wf-*$\mathcal{T}'$:
list-all (λ(*t*, *fs*).
[] ∉ *set fs* ∧
list-all (λ*f*. *field-lookup t f 0* ≠ *None*) *fs* ∧
distinct-prop disj-fn fs) $\mathcal{T}$
assumes *distinct-*$\mathcal{T}$: *distinct* (*map fst* $\mathcal{T}$)
begin

inductive *addressable-field* :: *typ-uinfo* ⇒ *qualified-field-name* ⇒ *typ-uinfo* ⇒ *bool* **where**
addressable-field-refl[*intro*!, *simp*]: $\bigwedge$*t*. *addressable-field t* [] *t*
| *addressable-field-step*: $\bigwedge$*t fs p p' u v n*.
map-of $\mathcal{T}$ *t* = *Some fs* $\Longrightarrow$ *p* ∈ *set fs* $\Longrightarrow$ *field-lookup t p 0* = *Some* (*u*, *n*) $\Longrightarrow$
addressable-field u p' v $\Longrightarrow$ *addressable-field t* (*p* @ *p'*) *v*

inductive *ptr-valid-u* :: *typ-uinfo* ⇒ *heap-typ-desc* ⇒ *addr* ⇒ *bool* **for** *t d* **where**
$\bigwedge$*r p a. root-ptr-valid-u r d a* $\Longrightarrow$ *addressable-field r p t* $\Longrightarrow$
ptr-valid-u t d (*a* + *of-nat* (*field-offset-untyped r p*))

definition *ptr-valid* :: *heap-typ-desc* ⇒ '*a*::*mem-type ptr* ⇒ *bool* **where**
[*ptr-valid-definition*]:

ptr-valid d p ≡ ptr-valid-u (typ-uinfo-t TYPE('a)) d (ptr-val p)

lemma *addressable-field-single*:
map-of 𝒯 t = Some fs ⟹ p ∈ set fs ⟹ field-lookup t p 0 = Some (v, n) ⟹
addressable-field t p v
⟨*proof*⟩

lemma *addressable-field-append*:
assumes ∗: *addressable-field t p1 u addressable-field u p2 v*
shows *addressable-field t (p1 @ p2) v*
⟨*proof*⟩

lemma *addressable-field-rev-induct*[*consumes 1*, *case-names refl step*]:
assumes ∗: *addressable-field t p u*
assumes *1*: ⋀*t*. *P t [] t*
assumes *2*: ⋀*t p′ u fs f v n*.
addressable-field t p′ u ⟹ map-of 𝒯 u = Some fs ⟹ f ∈ set fs ⟹
field-lookup u f 0 = Some (v, n) ⟹ P t p′ u ⟹ P t (p′ @ f) v
shows *P t p u*
⟨*proof*⟩

lemma *addressable-field-rev-cases*[*consumes 1*, *case-names refl step*]:
assumes ∗: *addressable-field t p u*
assumes *1*: *t = u ⟹ p = [] ⟹ P*
assumes *2*: ⋀*p′ fs f v n*.
addressable-field t p′ v ⟹ map-of 𝒯 v = Some fs ⟹ f ∈ set fs ⟹
field-lookup v f 0 = Some (u, n) ⟹ p = p′ @ f ⟹ P
shows *P*
⟨*proof*⟩

lemma *wf-𝒯*: *map-of 𝒯 t = Some fs ⟹ list-all* (λ*f*. *field-lookup t f 0 ≠ None*) *fs*
⟨*proof*⟩

lemma *𝒯-not-nil*: *map-of 𝒯 t = Some fs ⟹ [] ∉ set fs*
⟨*proof*⟩

lemma *addressable-field-refl-iff*[*simp*]:
addressable-field t [] u ⟷ t = u
⟨*proof*⟩

lemma *addressable-field-domain*:
addressable-field t p u ⟹ t ∈ fst ‘ set 𝒯 ∨ (t = u ∧ p = [])
⟨*proof*⟩

lemma *addressable-fieldD-mem-type-u*:
addressable-field t p u ⟹ mem-type-u t ∨ (t = u ∧ p = [])
⟨*proof*⟩

lemma *addressable-fieldD-field-lookup*:
assumes *p*: *addressable-field t p u* **shows** ∃ *n*. *field-lookup t p 0* = *Some* (*u*, *n*)
⟨*proof*⟩

lemma *addressable-field-same-iff*[*simp*]:
addressable-field t p t ⟷ *p* = []
⟨*proof*⟩

lemma *addressable-fieldD-field-lookup'*:
addressable-field t p u ⟹ *field-lookup t p 0* = *Some* (*u*, *field-offset-untyped t p*)
⟨*proof*⟩

lemma *addressable-fieldD-field-ti*:
assumes *p*: *addressable-field* (*typ-uinfo-t TYPE*('*a*::*c-type*)) *p t*
shows ∃ *u*. *field-ti TYPE*('*a*) *p* = *Some u* ∧ *export-uinfo u* = *t*
⟨*proof*⟩

lemma *addressable-fieldD-field-names*:
addressable-field t p u ⟹ *p* ∈ *set* (*field-names-u t u*)
⟨*proof*⟩

lemma *mem-type-u-addressable-field*:
assumes ∗: *addressable-field t fs u*
shows *mem-type-u u* ⟹ *mem-type-u t*
⟨*proof*⟩

lemma *wf-desc-addressable-field*:
assumes ∗: *addressable-field t fs u*
shows *wf-desc t* ⟹ *wf-desc u*
⟨*proof*⟩

lemma *wf-size-desc-addressable-field*:
assumes ∗: *addressable-field t fs u*
shows *wf-size-desc t* ⟹ *wf-size-desc u*
⟨*proof*⟩

lemma *field-offset-append-of-addressable-field*:
assumes *p*: *addressable-field t p u* **and** *q*: *addressable-field u q v*
shows *field-offset-untyped t* (*p* @ *q*) = *field-offset-untyped t p* + *field-offset-untyped u q*
⟨*proof*⟩

lemma *intvl-subset-of-addressable-field*:
assumes *p*: *addressable-field t p u*
shows {*a* + *of-nat* (*field-offset-untyped t p*) ..+ *size-td u*} ⊆ {*a* ..+ *size-td t*}
⟨*proof*⟩

lemma *root-ptr-valid-u-ptr-valid-u*:
assumes *a*: *root-ptr-valid-u t d a* **shows** *ptr-valid-u t d a*

⟨*proof*⟩

lemma *root-ptr-valid-ptr-valid*: *root-ptr-valid d p* ⟹ *ptr-valid d p*
⟨*proof*⟩

lemma *fold-ptr-valid'*[*fold-ptr-valid*]:
ptr-valid-u (*typ-uinfo-t TYPE*('*a*::*mem-type*)) *d x* = *ptr-valid d* (*PTR*('*a*) *x*)
⟨*proof*⟩

lemma *ptr-valid-u-trans*:
assumes ∗: *ptr-valid-u t d a*
shows *addressable-field t p u* ⟹ *ptr-valid-u u d* (*a* + *of-nat* (*field-offset-untyped t p*))
⟨*proof*⟩

lemma *ptr-valid-u-step*:
assumes ∗: *map-of* 𝒯 *t* = *Some F f* ∈ *set F field-lookup t f 0* = *Some* (*u*, *n*)
and *a*: *ptr-valid-u t d a*
shows *ptr-valid-u u d* (*a* + *of-nat* (*field-offset-untyped t f*))
⟨*proof*⟩

lemma *ptr-valid-u-recursion'*[*ptr-valid-u-recursion*]:
ptr-valid-u t d a ⟷
root-ptr-valid-u t d a ∨
list-ex (λ(*u*, *fs*). *list-ex* (λ*f*. ∃ *a' n*. *field-lookup u f 0* = *Some* (*t*, *n*) ∧
ptr-valid-u u d a' ∧ *a* = *a'* + *of-nat* (*field-offset-untyped u f*)) *fs*) 𝒯
⟨*proof*⟩

lemma 𝒯-*distinct*: *map-of* 𝒯 *t* = *Some fs* ⟹ *distinct-prop disj-fn fs*
⟨*proof*⟩

lemma *addressable-field-unique*:
assumes *t*: *addressable-field r p t* **shows** *addressable-field r p t'* ⟹ *t* = *t'*
⟨*proof*⟩

lemma *addressable-fields-split*:
assumes *r*: *addressable-field r p t addressable-field r* (*p* @ *p'*) *u*
shows *addressable-field t p' u*
⟨*proof*⟩

lemma *addressable-field-antisymm*:
addressable-field t p u ⟹ *addressable-field u q t* ⟹ *u* = *t*
⟨*proof*⟩

lemma *addressable-field-sub-typ*: *addressable-field t path u* ⟹ *u* ≤ *t*
⟨*proof*⟩

lemma *sub-typ-of-field-lookup*:
field-lookup (*typ-uinfo-t TYPE*('*a*)) *p 0* = *Some* (*typ-uinfo-t TYPE*('*b*), *n*) ⟹

$TYPE('b::mem\text{-}type) \leq_\tau TYPE('a::mem\text{-}type)$
⟨*proof*⟩

lemma *addressable-fields-imp-sub-typ*:
addressable-field (*typ-uinfo-t* $TYPE('a)$) *p* (*typ-uinfo-t* $TYPE('b)$) $\Longrightarrow$
$TYPE('b::mem\text{-}type) \leq_\tau TYPE('a::mem\text{-}type)$
⟨*proof*⟩

lemma *ptr-valid-u-cases*:
assumes *t*: *mem-type-u t*
assumes *t-a*: *ptr-valid-u t d a*
assumes *u-b*: *ptr-valid-u u d b*
shows *disjnt* {*a*..+*size-td t*} {*b*..+*size-td u*} ∨
(*t* = *u* ∧ *a* = *b*) ∨
(∃ *p*. *addressable-field t p u* ∧ *t* ≠ *u* ∧ *b* = *a* + *word-of-nat* (*field-offset-untyped t p*)) ∨
(∃ *p*. *addressable-field u p t* ∧ *t* ≠ *u* ∧ *a* = *b* + *word-of-nat* (*field-offset-untyped u p*))
(**is** *?disj* ∨ *?addr*)
⟨*proof*⟩

lemma *ptr-valid-cases'*:
fixes *p* :: *'a::mem-type ptr*
fixes *q* :: *'b::mem-type ptr*
assumes *p*: *ptr-valid d p*
assumes *q*: *ptr-valid d q*
shows *disjnt* (*ptr-span p*) (*ptr-span q*) ∨
(*typ-uinfo-t* $TYPE('a)$ = *typ-uinfo-t* $TYPE('b)$ ∧ *ptr-val p* = *ptr-val q*) ∨
(∃ *f*. *addressable-field* (*typ-uinfo-t* $TYPE('a)$) *f* (*typ-uinfo-t* $TYPE('b)$) ∧
typ-uinfo-t $TYPE('a)$ ≠ *typ-uinfo-t* $TYPE('b)$ ∧
q = $PTR('b)$ &(*p* → *f*)) ∨
(∃ *f*. *addressable-field* (*typ-uinfo-t* $TYPE('b)$) *f* (*typ-uinfo-t* $TYPE('a)$) ∧
typ-uinfo-t $TYPE('a)$ ≠ *typ-uinfo-t* $TYPE('b)$ ∧
p = $PTR('a)$ &(*q* → *f*))
⟨*proof*⟩

lemma *ptr-valid-u-cases-weak*:
assumes *mem-type-u t*
assumes *ptr-valid-u t d a*
assumes *ptr-valid-u u d b*
shows *disjnt* {*a*..+*size-td t*} {*b*..+*size-td u*} ∨
(∃ *p*. *addressable-field t p u* ∧ *b* = *a* + *word-of-nat* (*field-offset-untyped t p*)) ∨
(∃ *p*. *addressable-field u p t* ∧ *a* = *b* + *word-of-nat* (*field-offset-untyped u p*))
⟨*proof*⟩

lemma *ptr-valid-cases-weak*:
fixes *p* :: *'a::mem-type ptr*
fixes *q* :: *'b::mem-type ptr*
assumes *p*: *ptr-valid d p*

assumes *q*: *ptr-valid d q*
shows *disjnt* (*ptr-span p*) (*ptr-span q*) ∨
(∃*f*. *addressable-field* (*typ-uinfo-t TYPE*('*a*)) *f* (*typ-uinfo-t TYPE*('*b*)) ∧
q = *PTR*('*b*) &(*p* → *f*)) ∨
(∃*f*. *addressable-field* (*typ-uinfo-t TYPE*('*b*)) *f* (*typ-uinfo-t TYPE*('*a*)) ∧
p = *PTR*('*a*) &(*q* → *f*))
⟨*proof*⟩

lemma *ptr-valid-u-cvalid-u*:
assumes *t*: *mem-type-u t*
assumes *ptr-valid*: *ptr-valid-u t d a*
shows *cvalid-u t d a*
⟨*proof*⟩

lemma *ptr-valid-h-t-valid*: *ptr-valid d p* ⟹ *d* ⊨$_t$ (*p*::'*a*::*mem-type ptr*)
⟨*proof*⟩

sublocale *ptr-valid?*: *wf-ptr-valid ptr-valid*
⟨*proof*⟩

lemma *h-val-heap-update'* [*plift-simps*]:
fixes *p*::'*a*::*mem-type ptr*
fixes *q*::'*a*::*mem-type ptr*
assumes *ptr-valid-p*: *ptr-valid d p*
assumes *ptr-valid-q*: *ptr-valid d q*
shows *h-val* (*heap-update p v h*) *q* = ((*h-val h*)(*p*:=*v*)) *q*
⟨*proof*⟩

lemma *h-val-heap-update-padding'* [*plift-simps*]:
fixes *p*::'*a*::*mem-type ptr*
fixes *q*::'*a*::*mem-type ptr*
assumes *ptr-valid-p*: *ptr-valid d p*
assumes *ptr-valid-q*: *ptr-valid d q*
assumes *lbs*: *length bs* = *size-of TYPE*('*a*)
shows *h-val* (*heap-update-padding p v bs h*) *q* = ((*h-val h*)(*p*:=*v*)) *q*
⟨*proof*⟩

lemmas *ptr-valid-plift-Some-hval* [*plift-simps*]

theorem *disjoint-type-plift*:
fixes *p*::'*a*::*mem-type ptr*
assumes *unrelated1*: ¬ *TYPE*('*a*) ≤$_\tau$ *TYPE*('*b*::*mem-type*)
assumes *unrelated2*: ¬ *TYPE*('*b*) ≤$_\tau$ *TYPE*('*a*)
assumes *ptr-valid-p*: *ptr-valid* (*hrs-htd h*) *p*
shows *plift* ((*hrs-mem-update* (*heap-update p v*) *h*)) = (*plift h*::'*b ptr* ⇒ '*b option*)
⟨*proof*⟩

theorem *disjoint-type-update-h-val*:

fixes p::$'a$::*mem-type ptr*
fixes q::$'b$::*mem-type ptr*
assumes *unrelated1*: $\neg$ *TYPE*($'a$) $\leq_\tau$ *TYPE*($'b$)
assumes *unrelated2*: $\neg$ *TYPE*($'b$) $\leq_\tau$ *TYPE*($'a$)
assumes *typed-p*: *hrs-htd h* $\models_t$ *p*
assumes *typed-q*: *hrs-htd h* $\models_t$ *q*
shows *h-val* (*hrs-mem* (*hrs-mem-update* (*heap-update p v*) *h*)) *q* = *h-val* (*hrs-mem h*) *q*
⟨*proof*⟩

lemma *intvl-Suc-0*: {*x* ..+ *Suc 0*} = {*x*}
⟨*proof*⟩

lemma *ptr-span-subset-of-addressable-field*:
fixes *q*:: $'a$::*mem-type ptr*
shows *addressable-field t p* (*typ-uinfo-t TYPE*($'a$)) ⟹
ptr-val q = *x* + *word-of-nat* (*field-offset-untyped t p*) ⟹
ptr-span q ⊆ {*x*..+*size-td t*}
⟨*proof*⟩

lemma *addressable-field-wf-descD*:
assumes *path*: *addressable-field t path u*
shows *t* = *u* ∨ (*wf-desc t* ∧ *wf-desc u*)
⟨*proof*⟩

lemma *addressable-field-wf-size-descD*:
assumes *path*: *addressable-field t path u*
shows *t* = *u* ∨ (*wf-size-desc t* ∧ *wf-size-desc u*)
⟨*proof*⟩

lemma *root-ptr-valid-u-ptr-valid-u-cases*:
fixes *p q* :: *addr*
assumes *root-valid-p*: *root-ptr-valid-u t d p*
assumes *valid-q*: *ptr-valid-u s d q*
shows {*p* ..+ *size-td t*} ∩ {*q* ..+ *size-td s*} = {} ∨
(∃ *path*. *addressable-field t path s* ∧ *q* = *p* + *of-nat* (*field-offset-untyped t path*))
⟨*proof*⟩

lemma *root-ptr-valid-u-ptr-valid-cases*:
fixes *p* :: *addr* **and** *q* :: $'b$::*mem-type ptr*
assumes *root-valid-p*: *root-ptr-valid-u t d p*
assumes *valid-q*: *ptr-valid d q*
shows {*p* ..+ *size-td t*} ∩ *ptr-span q* = {} ∨
(∃ *path*. *addressable-field t path* (*typ-uinfo-t TYPE*($'b$)) ∧
ptr-val q = *p* + *of-nat* (*field-offset-untyped t path*))
⟨*proof*⟩

lemma *root-ptr-valid-ptr-valid-cases*:
fixes *p*:: $'a$::*mem-type ptr*

fixes *q*:: *'b*::*mem-type ptr*
assumes *root-valid-p*: *root-ptr-valid d p*
assumes *valid-q*: *ptr-valid d q*
shows *ptr-span p* ∩ *ptr-span q* = {} ∨
(∃ *path*. *addressable-field* (*typ-uinfo-t TYPE*(*'a*)) *path* (*typ-uinfo-t TYPE*(*'b*))
∧
q = *PTR* (*'b*) &(*p*→*path*))
⟨*proof*⟩

lemma *stack-allocs-ptr-valid-cases1*:
fixes *p*:: *'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *alloc*: (*p*, *d'*) ∈ *stack-allocs n* 𝒮 *TYPE*(*'a*) *d*
assumes *q-not-stack-byte*: *typ-uinfo-t TYPE*(*'b*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *valid-q*: *ptr-valid d' q*
shows ((∃ *i path*. *i* < *n* ∧
addressable-field (*typ-uinfo-t TYPE*(*'a*)) *path* (*typ-uinfo-t TYPE*(*'b*)) ∧
q = *PTR* (*'b*) &((*p* +$_p$ *int i*)→*path*)) ∧ ¬ *ptr-valid d q*) ∨
(*ptr-valid d q* ∧ {*ptr-val p* ..+ *n* ∗ *size-of TYPE*(*'a*)} ∩ *ptr-span q* = {})
⟨*proof*⟩

lemma *stack-allocs-ptr-valid-cases1'* [*consumes 3*, *case-names addressable-field disjoint*]:
fixes *p*:: *'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *alloc*: (*p*, *d'*) ∈ *stack-allocs n* 𝒮 *TYPE*(*'a*) *d*
assumes *q-not-stack-byte*: *typ-uinfo-t TYPE*(*'b*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *valid-q*: *ptr-valid d' q*
assumes *addressable-field*: ⋀*i path*. *i* < *n* ⟹
addressable-field (*typ-uinfo-t TYPE*(*'a*)) *path* (*typ-uinfo-t TYPE*(*'b*)) ⟹
q = *PTR* (*'b*) &((*p* +$_p$ *int i*)→*path*) ⟹ ¬ *ptr-valid d q* ⟹ *P*
assumes *disjoint*: *ptr-valid d q* ⟹ {*ptr-val p* ..+ *n* ∗ *size-of TYPE*(*'a*)} ∩ *ptr-span q* = {} ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *stack-allocs-ptr-valid-cases2*:
fixes *p*:: *'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *alloc*: (*p*, *d'*) ∈ *stack-allocs n* 𝒮 *TYPE*(*'a*) *d*
assumes *q-not-stack-byte*: *typ-uinfo-t TYPE*(*'b*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *valid-q*: *ptr-valid d q*
shows *ptr-valid d' q*
⟨*proof*⟩

lemma *stack-allocs-ptr-valid-cases3*:
fixes *p*:: *'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *alloc*: (*p*, *d'*) ∈ *stack-allocs n* 𝒮 *TYPE*(*'a*) *d*

assumes *q-not-stack-byte*: *typ-uinfo-t TYPE('b)* $\neq$ *typ-uinfo-t TYPE(stack-byte)*
assumes *i*: $i < n$
assumes *path*: *addressable-field* (*typ-uinfo-t TYPE('a)*) *path* (*typ-uinfo-t TYPE('b)*)
assumes *q*: $q = PTR\ ('b)\ \&((p +_p int\ i) \rightarrow path)$
shows *ptr-valid d' q*
⟨*proof*⟩

theorem *stack-allocs-ptr-valid-cases*:
fixes *p*:: *'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *alloc*: $(p,\ d') \in$ *stack-allocs n* $\mathcal{S}$ *TYPE('a) d*
assumes *q-not-stack-byte*: *typ-uinfo-t TYPE('b)* $\neq$ *typ-uinfo-t TYPE(stack-byte)*
shows *ptr-valid d' q* $\longleftrightarrow$
$(((\exists\, i\ path.\ i < n \wedge$ *addressable-field* (*typ-uinfo-t TYPE('a)*) *path* (*typ-uinfo-t TYPE('b)*) $\wedge$
$q = PTR\ ('b)\ \&((p +_p int\ i) \rightarrow path)) \wedge \neg$ *ptr-valid d q*) $\vee$
(*ptr-valid d q* $\wedge$ {*ptr-val p ..+ n* $*$ *size-of TYPE('a)*} $\cap$ *ptr-span q* $= \{\}$))
⟨*proof*⟩

lemma *stack-releases-ptr-valid-cases1*:
fixes *p*:: *'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *p-not-stack-byte*: *typ-uinfo-t TYPE('a)* $\neq$ *typ-uinfo-t TYPE(stack-byte)*
assumes *q-not-stack-byte*: *typ-uinfo-t TYPE('b)* $\neq$ *typ-uinfo-t TYPE(stack-byte)*
assumes *valid-q*: *ptr-valid* (*stack-releases n p d*) *q*
shows {*ptr-val p ..+ n* $*$ *size-of TYPE('a*::*mem-type)*} $\cap$ *ptr-span q* $= \{\} \wedge$
ptr-valid d q
⟨*proof*⟩

lemma *stack-releases-ptr-valid-cases2*:
fixes *p*:: *'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *p-not-stack-byte*: *typ-uinfo-t TYPE('a)* $\neq$ *typ-uinfo-t TYPE(stack-byte)*
assumes *q-not-stack-byte*: *typ-uinfo-t TYPE('b)* $\neq$ *typ-uinfo-t TYPE(stack-byte)*
assumes *root-p*: $\bigwedge i.\ i < n \Longrightarrow$ *root-ptr-valid d* $(p +_p int\ i)$
assumes *disj*: {*ptr-val p ..+ n* $*$ *size-of TYPE('a*::*mem-type)*} $\cap$ *ptr-span q* $= \{\}$
assumes *valid-q*: *ptr-valid d q*
shows *ptr-valid* (*stack-releases n p d*) *q*
⟨*proof*⟩

lemma *stack-releases-ptr-valid-cases3*:
fixes *p*:: *'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *p-not-stack-byte*: *typ-uinfo-t TYPE('a)* $\neq$ *typ-uinfo-t TYPE(stack-byte)*
assumes *q-not-stack-byte*: *typ-uinfo-t TYPE('b)* $\neq$ *typ-uinfo-t TYPE(stack-byte)*
assumes *root-p*: $\bigwedge i.\ i < n \Longrightarrow$ *root-ptr-valid d* $(p +_p int\ i)$
assumes *valid-q*: *ptr-valid d q*

assumes *invalid-q*: ¬ *ptr-valid* (*stack-releases n p d*) *q*
shows ∃ *path i*.
addressable-field (*typ-uinfo-t TYPE*('*a*)) *path* (*typ-uinfo-t TYPE*('*b*)) ∧ *i* < *n*
∧
q = *PTR*('*b*) &(*p* +$_p$ *int i*→*path*)
⟨*proof*⟩

lemma *stack-releases-ptr-valid-cases*:
fixes *p*:: '*a*::*mem-type ptr*
fixes *q*:: '*b*::*mem-type ptr*
assumes *p-not-stack-byte*: *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *q-not-stack-byte*: *typ-uinfo-t TYPE*('*b*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *root-p*: ⋀*i*. *i* < *n* ⟹ *root-ptr-valid d* (*p* +$_p$ *int i*)
shows *ptr-valid* (*stack-releases n p d*) *q* ⟷
ptr-valid d q ∧ {*ptr-val p* ..+ *n* ∗ *size-of TYPE*('*a*::*mem-type*)} ∩ *ptr-span q*
= {}
⟨*proof*⟩

lemma *ptr-valid-same-type-neq-no-overlap-conv*:
fixes *p*:: '*a*::*mem-type ptr*
fixes *q*:: '*a*::*mem-type ptr*
assumes *valid-p*: *ptr-valid d p*
assumes *valid-q*: *ptr-valid d q*
shows *p* ≠ *q* ⟷ *ptr-span p* ∩ *ptr-span q* = {}
⟨*proof*⟩

theorem *stack-allocs-the-default-plift* [*stack-the-default-plift*]:
fixes *p*:: '*a*::*xmem-type ptr*
fixes *q*:: '*b*::*mem-type ptr*
assumes *alloc*: (*p*, *d'*) ∈ *stack-allocs n* 𝒮 *TYPE*('*a*) (*hrs-htd h*)
assumes *q-not-stack-byte*: *typ-uinfo-t TYPE*('*b*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
shows *the-default c-type-class.zero*
(*plift* (*hrs-mem-update* (*fold* (λ*i*. *heap-update* (*p* +$_p$ *int i*) *c-type-class.zero*)
[*0*..<*n*]) (*hrs-htd-update* (λ-. *d'*) *h*)) *q*) =
the-default c-type-class.zero (*plift h q*)
⟨*proof*⟩

theorem *stack-releases-the-default-plift* [*stack-the-default-plift*]:
fixes *p*:: '*a*::*xmem-type ptr*
fixes *q*:: '*b*::*mem-type ptr*
assumes *roots*: ⋀*i*. *i* < *n* ⟹ *root-ptr-valid* (*hrs-htd h*) (*p* +$_p$ *int i*)
assumes *p-not-stack-byte*: *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *q-not-stack-byte*: *typ-uinfo-t TYPE*('*b*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
shows *the-default c-type-class.zero*
(*plift*
(*hrs-mem-update* (*fold* (λ*i*. *heap-update* (*p* +$_p$ *int i*) *c-type-class.zero*)
[*0*..<*n*]) *h*)
q) =
the-default c-type-class.zero (*plift* (*hrs-htd-update* (*stack-releases n p*) *h*) *q*)

⟨*proof*⟩

lemma *h-val-heap-update-padding-heap-update-conv*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*::*'b*::*mem-type ptr*
assumes *valid-p*: *ptr-valid d p*
assumes *valid-q*: *ptr-valid d q*
assumes *lbs*: *length bs* = *size-of TYPE*(*'a*)
shows *h-val* ((*heap-update-padding p v bs h*)) *q* = *h-val* (*heap-update p v h*) *q*
⟨*proof*⟩

lemma *plift-heap-update-padding-heap-update-conv*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*::*'b*::*mem-type ptr*
assumes *valid-p*: *ptr-valid* (*hrs-htd h*) *p*
assumes *lbs*: *length bs* = *size-of TYPE*(*'a*)
shows *plift* (*hrs-mem-update* (*heap-update-padding p v bs*) *h*) *q* = *plift* (*hrs-mem-update* (*heap-update p v*) *h*) *q*
⟨*proof*⟩

theorem *plift-heap-update-padding-heap-update-pointless-conv* [*plift-heap-update-padding-heap-update-conv*]:
fixes *p*::*'a*::*mem-type ptr*
assumes *valid-p*: *ptr-valid* (*hrs-htd h*) *p*
assumes *lbs*: *length bs* = *size-of TYPE*(*'a*)
shows *plift* (*hrs-mem-update* (*heap-update-padding p v bs*) *h*) = *plift* (*hrs-mem-update* (*heap-update p v*) *h*)
⟨*proof*⟩

lemma *ptr-valid-stack-byte-disjoint*:
fixes *q*::*'a*::{*mem-type*, *stack-type*} *ptr*
fixes *p*::*stack-byte ptr*
assumes *root-ptr-valid h p*
assumes *ptr-valid h q*
shows *ptr-span p* ∩ *ptr-span q* = {}
⟨*proof*⟩

lemma *h-val-heap-update-list-stack-byte*:
fixes *q*::*'a*::{*mem-type*, *stack-type*} *ptr*
assumes *valid-p*: *root-ptr-valid d* (*p*::*stack-byte ptr*)
assumes *valid-q*: *ptr-valid d q*
assumes *lbs*: *length bs* = *size-of TYPE*(*stack-byte*)
shows *h-val* (*heap-update-list* (*ptr-val p*) *bs h*) *q* = *h-val h q*
⟨*proof*⟩

lemma *h-val-heap-update-list-stack-byte'*:
fixes *q*::*'a*::{*mem-type*, *stack-type*} *ptr*
fixes *p*::*stack-byte ptr*
assumes *valid-p*: ⋀*i*. *i* < *length bs* ⟹ *root-ptr-valid d* (*p* +$_p$ *int i*)

assumes *valid-q*: *ptr-valid d q*
shows *h-val* (*heap-update-list* (*ptr-val p*) *bs h*) *q* = *h-val h q*
⟨*proof*⟩

lemma *plift-heap-update-list-stack-byte*:
fixes *q*::'*a*::{*mem-type*, *stack-type*} *ptr*
fixes *p*::*stack-byte ptr*
assumes *valid-p*: ⋀*i*. *i* < *length bs* ⟹ *root-ptr-valid* (*hrs-htd h*) (*p* +$_p$ *int i*)
shows *plift* (*hrs-mem-update* (*heap-update-list* (*ptr-val p*) *bs*) *h*) *q* = *plift h q*
⟨*proof*⟩

lemma *plift-heap-update-list-stack-byte-pointless*[*plift-heap-update-list-stack-byte*]:
fixes *p*::*stack-byte ptr*
assumes *valid-p*: ⋀*i*. *i* < *length bs* ⟹ *root-ptr-valid* (*hrs-htd h*) (*p* +$_p$ *int i*)
shows *plift* (*hrs-mem-update* (*heap-update-list* (*ptr-val p*) *bs*) *h*) =
(*plift h*::'*a*::{*mem-type*, *stack-type*} *ptr* ⇒ '*a option*)
⟨*proof*⟩

lemma *plift-eq-plift*:
hrs-htd h = *hrs-htd h'* ⟹
(⋀*p*::'*a ptr*. *ptr-valid* (*hrs-htd h'*) *p* ⟹
h-val (*hrs-mem h*) *p* = *h-val* (*hrs-mem h'*) *p*) ⟹
(*plift h*::'*a*::*mem-type ptr* ⇒ '*a option*) = *plift h'*
⟨*proof*⟩

definition *addressable-fields* :: '*a*::*mem-type itself* ⇒ (*field-name list* × '*a xtyp-info*) *list* **where**
addressable-fields TYPE('*a*) = (*case map-of* $\mathcal{T}$ (*typ-uinfo-t TYPE*('*a*)) *of*
None ⇒ []
| *Some fs* ⇒ *map* (λ*f*. (*f*, *the* (*field-ti TYPE*('*a*) *f*))) *fs*)

abbreviation *merge-addressable-fields* :: '*a*::*mem-type* ⇒ '*a* ⇒ '*a* **where**
merge-addressable-fields ≡ *merge-ti-list* (*map snd* (*addressable-fields TYPE*('*a*)))

definition *read-dedicated-heap* :: *heap-raw-state* ⇒ '*a*::*mem-type typ-heap-g* **where**
read-dedicated-heap h p = *merge-addressable-fields ZERO*('*a*) (*the-default ZERO*('*a*) (*plift h p*))

definition *write-dedicated-heap* :: '*a*::*mem-type ptr* ⇒ '*a* ⇒ *heap-raw-state* ⇒ *heap-raw-state* **where**
write-dedicated-heap p v = *hrs-mem-update* (*heap-upd p* (λ*old*. *merge-addressable-fields old v*))

lemma *mem-addressable-fields*:
assumes *F*: *map-of* $\mathcal{T}$ (*typ-uinfo-t TYPE*('*a*::*mem-type*)) = *Some F*
shows *f* ∈ *set F* ⟹ *field-ti TYPE*('*a*) *f* = *Some u* ⟹ (*f*, *u*) ∈ *set* (*addressable-fields TYPE*('*a*))
⟨*proof*⟩

lemma *mem-snd-addressable-fields*:
assumes *F*: *map-of* $\mathcal{T}$ (*typ-uinfo-t TYPE*('*a*::*mem-type*)) = *Some F f* ∈ *set F*
field-ti TYPE('*a*) *f* = *Some u*
shows *u* ∈ *snd* ' *set* (*addressable-fields TYPE*('*a*))
⟨*proof*⟩

lemma *list-all-field-ti-addressable-fields*:
list-all (λ(*f*, *u*). *field-ti TYPE*('*a*) *f* = *Some u*) (*addressable-fields TYPE*('*a*::*mem-type*))
⟨*proof*⟩

lemma *distinct-disj-fn-addressable-fields*:
distinct-prop disj-fn (*map fst* (*addressable-fields TYPE*('*a*::*mem-type*)))
⟨*proof*⟩

lemma *list-all-field-ti-snd-addressable-fields*:
list-all (λ*u*. ∃*f*. *field-ti TYPE*('*a*) *f* = *Some u*) (*map snd* (*addressable-fields TYPE*('*a*::*mem-type*)))
⟨*proof*⟩

sublocale *merge-addressable-fields*: *is-scene merge-addressable-fields* :: '*a*::*xmem-type* ⇒ '*a* ⇒ '*a*
⟨*proof*⟩

lemma *ptr-valid-u-cases-same-type*:
assumes *t*: *mem-type-u t* **and** *a*: *ptr-valid-u t d a* **and** *b*: *ptr-valid-u t d b*
shows *a* = *b* ∨ *disjnt* {*a*..+*size-td t*} {*b*..+*size-td t*}
⟨*proof*⟩

lemma *read-dedicated-heap-heap-update-list-stack-byte-pointless*[*simp*]:
fixes *p*::*stack-byte ptr*
assumes *valid-p*: ⋀*i*. *i* < *length bs* ⟹ *root-ptr-valid* (*hrs-htd h*) (*p* +$_p$ *int i*)
shows *read-dedicated-heap* (*hrs-mem-update* (*heap-update-list* (*ptr-val p*) *bs*) *h*)
=
(*read-dedicated-heap h*::'*a*::{*mem-type*, *stack-type*} *ptr* ⇒ '*a*)
⟨*proof*⟩

lemma *read-dedicated-heap-write-dedicated-heap-ne*[*simp*]:
fixes *p* :: '*a*::*xmem-type ptr* **and** *q* :: '*b*::*xmem-type ptr*
assumes *ne*: *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*('*b*)
assumes *p*: *ptr-valid* (*hrs-htd h*) *p*
shows *read-dedicated-heap* (*write-dedicated-heap p x h*) *q* = *read-dedicated-heap h q*
⟨*proof*⟩

lemma *read-dedicated-heap-write-dedicated-heap*[*simp*]:
ptr-valid (*hrs-htd h*) (*p*::'*a*::*xmem-type ptr*) ⟹
read-dedicated-heap (*write-dedicated-heap p x h*) =
upd-fun p (λ*old*. *merge-addressable-fields old x*) (*read-dedicated-heap h*)
⟨*proof*⟩

lemma *hrs-mem-update-heap-update*:
fixes $p :: 'a::xmem\text{-}type\ ptr$
shows *hrs-mem-update* (*heap-update* $p\ x$) $h =$
fold ($\lambda(f,\ u).$ *hrs-mem-update* (*heap-upd-list* (*size-td* u) &($p\rightarrow f$) (*access-ti* u x)))
(*addressable-fields* $TYPE('a)$)
(*write-dedicated-heap* $p\ x\ h$)
⟨*proof*⟩

lemma *hrs-mem-update-heap-update'*:
fixes $p :: 'a::xmem\text{-}type\ ptr$
shows *hrs-mem-update* (*heap-update* $p\ x$) =
fold ($\lambda(f,\ u).$ *hrs-mem-update* (*heap-upd-list* (*size-td* u) &($p\rightarrow f$) (*access-ti* u x)))
(*addressable-fields* $TYPE('a)$) ∘
write-dedicated-heap $p\ x$
⟨*proof*⟩

lemma *hrs-htd-write-dedicated-heap*[*simp*]:
hrs-htd (*write-dedicated-heap* $p\ x\ h$) = *hrs-htd* h
⟨*proof*⟩

lemma *partial-pointer-lense-merge-addressable-fields*:
fixes $r :: 'h \Rightarrow 'a::xmem\text{-}type\ ptr \Rightarrow 'a$
assumes *lense* $r\ w$
shows
partial-pointer-lense ($\lambda x\ y.$ *merge-addressable-fields* $y\ x$)
($\lambda h\ p\ x\ .$ *merge-addressable-fields* x ($r\ h\ p$)) ($\lambda p\ x.\ w$ (*upd-fun* p ($\lambda old.$ *merge-addressable-fields old* x)))
⟨*proof*⟩

lemma *pointer-lense-of-partials-cover*:
fixes $rs :: ('h \Rightarrow 'a::xmem\text{-}type\ ptr \Rightarrow 'a \Rightarrow 'a)\ list$
assumes *g-u*: *lense* $g\ u$
assumes $*$:
length ms = *length rs length ms* = *length ws*
list-all ($\lambda(m,\ r,\ w).$ *partial-pointer-lense* $m\ r\ w$) (*zip ms* (*zip rs ws*))
and $**$:
$\bigwedge a\ b\ c.$ *distinct-prop* ($\lambda m1\ m2.\ m1\ a$ ($m2\ b\ c$) = $m2\ b$ ($m1\ a\ c$)) *ms*
$\bigwedge p\ a\ q\ b\ h.$ *distinct-prop* ($\lambda w1\ w2.\ p = q \vee$ *disjnt* (*ptr-span* p) (*ptr-span* q) ⟶
$w1\ p\ a$ ($w2\ q\ b\ h$) = $w2\ q\ b$ ($w1\ p\ a\ h$)) ((λ$p\ x.\ u$ (*upd-fun* p (λ*old.* *merge-addressable-fields old* x))) # *ws*)
assumes *ms-cover*: $\bigwedge a\ b.$ *fold* ($\lambda m.\ m\ a$) *ms* b = *merge-addressable-fields* $a\ b$
assumes R: $\bigwedge h\ p.\ R\ h\ p$ = *fold* ($\lambda r.\ r\ h\ p$) *rs* ($g\ h\ p$)
assumes W: $\bigwedge p\ f\ h.\ W\ p\ f\ h =$
fold ($\lambda w.\ w\ p$ (f ($R\ h\ p$))) *ws* (u (*upd-fun* p (λ*old.* *merge-addressable-fields old* (f ($R\ h\ p$)))) h)
shows *pointer-lense* $R\ W$

⟨*proof*⟩

lemma *cover-read-dedicated-heap*[*simp*]:
merge-addressable-fields (*read-dedicated-heap h p*) = *merge-addressable-fields ZERO*('*a*::*xmem-type*)
⟨*proof*⟩

lemma *read-dedicated-heap-fun-upd-cover-zero-eq-upd-fun*[*simp*]:
((*read-dedicated-heap h*)(*p* := *merge-addressable-fields ZERO*('*a*::*xmem-type*) *x*))
q =
upd-fun p (λ*old*. *merge-addressable-fields old x*) (*read-dedicated-heap h*) *q*
⟨*proof*⟩

theorem *stack-allocs-read-dedicated-heap* [*stack-the-default-plift*]:
fixes *p*:: '*a*::*xmem-type ptr*
assumes *alloc*: (*p*, *d'*) ∈ *stack-allocs n* $\mathcal{S}$ *TYPE*('*a*) (*hrs-htd h*)
assumes *q-not-stack-byte*: *typ-uinfo-t TYPE*('*b*::*xmem-type*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
shows *read-dedicated-heap* (*hrs-mem-update* (*fold* (λ*i*. *heap-update* (*p* +$_p$ *int i*)
c-type-class.zero) [*0*..<*n*]) (*hrs-htd-update* (λ-. *d'*) *h*)) =
(*read-dedicated-heap h* :: '*b ptr* ⇒ -)
⟨*proof*⟩

theorem *stack-releases-read-dedicated-heap* [*stack-the-default-plift*]:
fixes *p*:: '*a*::*xmem-type ptr*
assumes *roots*: ⋀*i*. *i* < *n* ⟹ *root-ptr-valid* (*hrs-htd h*) (*p* +$_p$ *int i*)
assumes *p-not-stack-byte*: *typ-uinfo-t TYPE*('*a*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
assumes *q-not-stack-byte*: *typ-uinfo-t TYPE*('*b*) ≠ *typ-uinfo-t TYPE*(*stack-byte*)
shows *read-dedicated-heap* (*hrs-mem-update* (*fold* (λ*i*. *heap-update* (*p* +$_p$ *int i*)
c-type-class.zero) [*0*..<*n*]) *h*) =
(*read-dedicated-heap* (*hrs-htd-update* (*stack-releases n p*) *h*) :: '*b*::*xmem-type ptr*
⇒ -)
⟨*proof*⟩

lemma *ptr-valid-addressable-field-field-lvalue*[*addressable-field-exec*]:
fixes *p*::'*a*::*mem-type ptr*
assumes *ptr-valid h p*
assumes *addressable-field* (*typ-uinfo-t* (*TYPE*('*a*))) *fs* (*typ-uinfo-t* (*TYPE*('*b*::*mem-type*)))
shows *ptr-valid h* (*PTR*('*b*) (&(*p*→*fs*)))
⟨*proof*⟩

lemma *addressable-field-exec*[*addressable-field-exec*]:
shows
addressable-field t ps v =
(*case ps of*
[] => *t* = *v*
| - ⇒ (∃*fs p*. *map-of* $\mathcal{T}$ *t* = *Some fs* ∧
List.find (λ*p*. ∃*u n*. *field-lookup t p 0* = *Some* (*u*, *n*) ∧ *p* @ *drop* (*length*
p) *ps* = *ps* ∧
addressable-field u (*drop* (*length p*) *ps*) *v*) *fs* = *Some p*))
⟨*proof*⟩

⟨*ML*⟩

end

lemma *fold-field-lvalue*:
x + *word-of-nat* (*field-offset-untyped* (*typ-uinfo-t TYPE*('*a*::*c-type*)) *f*) = &(*PTR*('*a*) *x*→*f*)
⟨*proof*⟩

lemma *field-lvalue-same-root-ptr-conv*:
&(*p*::'*a*:: *c-type ptr*→*f*) = &(*q*::'*a*:: *c-type ptr*→*f*) ⟷ *p* = *q*
⟨*proof*⟩

lemma *ptr-exhaust-eq*: **fixes** *p*::'*a*::*c-type ptr* **shows** *PTR*('*a*) (*ptr-val p*) = *p*
⟨*proof*⟩

lemma *fold-exists-ptr*: (∃ *x*. *P* (*PTR*('*a*::*c-type*) *x*)) ⟷ (∃ *q*. *P q*)
⟨*proof*⟩

21.1.1 Syntax *PTR-VALID*('*a*)

context *open-types*
begin

We want to provide syntax that makes the type of *ptr-valid* visible on the term level for interpretations of the *open-types*. This is a bit tricky as Isabelle does not (yet) provide local syntax and local print / parse translations. We implement it by a combination of theory level syntax and translations and a local declarations for the interpretations.

end
syntax
-*ptr-valid* :: *type* ⇒ *logic* (‹(‹*indent*=*1 notation*=‹*mixfix PTR-VALID*››*PTR'-VALID*/(*1* '(-')))›)

⟨*ML*⟩

This theory level setup actually provides the local syntax when working within a locale like *open-types*. The *print-translation* is triggered on constant (abbreviation) *local.ptr-valid*

context *open-types*
begin
term *PTR-VALID*(*32 word*) — Works in both directions by theory level setup above.
⟨*ML*⟩

This declaration provides the setup for global interpretations. The print translation is triggered on constant (abbreviation) <*instance*−*name*>.*ptr-valid* introduced by the iterpretation. Keep in mind, that when printing a term

on the internal *PTR-VALID*('*a*) first the notations / abbreviatons are applied. These introduce the syntactic constant *PTR-VALID*('*a*) on which the translation then triggers.

end

21.1.2 Syntax *IS-VALID*('*a*)

context *open-types*
begin

Building on top of *PTR-VALID* we provide syntax that also takes the *heap-typing* of lifted globals into accound: *IS-VALID*(*32 word*) *s p* is translated to *PTR-VALID*('*a*) (*heap-typing s*) *p* Note that *heap-typing* is a field of the lifted globals record that is defined later.

end

syntax
-is-valid :: *type* ⇒ *logic* ⇒ *logic*
(‹(‹*indent=1 notation=‹mixfix IS-VALID››IS'-VALID/(1'(-'))*) -› [*0*, *1000*] *1000*)

⟨*ML*⟩

end

theory *AbstractArrays*
imports
TypHeapLib
WordSetup
begin

primrec
array-addrs :: ('*a*::*mem-type*) *ptr* ⇒ *nat* ⇒ '*a ptr list*
where
array-addrs - 0 = []
| *array-addrs p* (*Suc n*) = *p* # (*array-addrs* (*p* +$_p$ *1*) *n*)

declare *array-addrs.simps*(*2*) [*simp del*]

lemma *hd-in-array-addrs* [*simp*]:
(*x* ∈ *set* (*array-addrs x n*)) = (*n* > *0*)
⟨*proof*⟩

lemma *array-addrs-1* [*simp*]:
array-addrs p (*Suc 0*) = [*p*]

array-addrs p 1 = [p]
⟨*proof*⟩

lemma *array-addrs-ptr-aligned*:
⟦ *x ∈ set (array-addrs p n); ptr-aligned p* ⟧ ⟹ *ptr-aligned x*
⟨*proof*⟩

lemma *set-array-addrs-unfold-last*:
shows *set (array-addrs a (Suc n)) = set (array-addrs a n) ∪ {(a :: ('a::mem-type) ptr) +$_p$ int n}*
(**is** *?LHS a n = ?RHS a n*)
⟨*proof*⟩

lemma *set-array-addrs*:
set (array-addrs (p :: ('a::mem-type) ptr) n)
= {x. ∃ k. x = p +$_p$ int k ∧ k < n }
⟨*proof*⟩

end

theory *HeapLift*
imports
In-Out-Parameters
Split-Heap
AbstractArrays
begin

21.2 Refinement Lemmas

lemma *ucast-ucast-id*:
LENGTH('a) ≤ LENGTH('b) ⟹ ucast (ucast (x::'a::len word)::'b::len word) = x
⟨*proof*⟩

lemma *lense-ucast-signed*:
lense (unsigned :: 'a::len word ⇒ 'a signed word) (λv x. unsigned (v (unsigned x)))
⟨*proof*⟩

lemma *pointer-lense-ucast-signed*:
fixes *r :: 'h ⇒ 'a::len8 word ptr ⇒ 'a word*
assumes *pointer-lense r w*
shows *pointer-lense*
(λh p. UCAST('a → 'a signed) (r h (PTR-COERCE('a signed word → 'a word)

$p)))$
$(\lambda p\ m.\ w\ (PTR\text{-}COERCE('a\ signed\ word \to 'a\ word)\ p)$
$(\lambda w.\ UCAST('a\ signed \to 'a)\ (m\ (UCAST('a \to 'a\ signed)\ w))))$
⟨*proof*⟩

lemma (**in** *xmem-type*) *length-to-bytes*:
length (*to-bytes* (*v*::'*a*) *bs*) = *size-of TYPE*('*a*)
⟨*proof*⟩

lemma (**in** *xmem-type*) *heap-update-padding-eq*:
length bs = *size-of TYPE*('*a*) ⟹
heap-update-padding p v bs h = *heap-update p v* (*heap-update-list* (*ptr-val p*) *bs h*)
⟨*proof*⟩

lemma (**in** *xmem-type*) *heap-update-padding-eq'*:
length bs = *size-of TYPE*('*a*) ⟹
heap-update-padding p v bs = *heap-update p v* ∘ *heap-update-list* (*ptr-val p*) *bs*
⟨*proof*⟩

lemma *split-disj-asm*: *P* (*x* ∨ *y*) = (¬ (*x* ∧ ¬ *P x* ∨ ¬ *x* ∧ ¬ *P y*))
⟨*proof*⟩

lemma *comp-commute-of-fold*:
assumes *x*: *x* = *fold f xs*
assumes *xs*: *list-all* (λ*x*. *f x o a* = *a o f x*) *xs*
shows *x o a* = *a o x*
⟨*proof*⟩

definition *padding-closed-under-all-fields* **where**
padding-closed-under-all-fields t ⟷
(∀ *s f n bs bs'*. *field-lookup t f 0* = *Some* (*s*, *n*) ⟶
eq-upto-padding t bs bs' ⟶ *eq-upto-padding s* (*take* (*size-td s*) (*drop n bs*)) (*take* (*size-td s*) (*drop n bs'*)))

lemma *padding-closed-under-all-fields-typ-uinfo-t*:
padding-closed-under-all-fields (*typ-uinfo-t TYPE*('*a*::*xmem-type*))
⟨*proof*⟩

lemma (**in** *open-types*) *plift-heap-update-list-eq-upto-padding*:
assumes *t*: *mem-type-u t* **and** *t'*: *padding-closed-under-all-fields t*
assumes *a*: *ptr-valid-u t* (*hrs-htd h*) *a*
assumes *bs-bs'*: *eq-upto-padding t bs bs'*
shows *plift* (*hrs-mem-update* (*heap-update-list a bs*) *h*) =
(*plift* (*hrs-mem-update* (*heap-update-list a bs'*) *h*)::'*a*::*xmem-type ptr* ⇒ '*a option*)
⟨*proof*⟩

lemma (**in** *open-types*) *read-dedicated-heap-heap-update-list-eq-upto-padding*[*simp*]:

assumes *t*: *mem-type-u t* **and** *t'*: *padding-closed-under-all-fields t*
assumes *a*: *ptr-valid-u t* (*hrs-htd h*) *a*
assumes *bs-bs'*: *eq-upto-padding t bs bs'*
shows *read-dedicated-heap* (*hrs-mem-update* (*heap-update-list a bs*) *h*) =
(*read-dedicated-heap* (*hrs-mem-update* (*heap-update-list a bs'*) *h*)::'*a*::*xmem-type ptr* ⇒ '*a*) ⟷ *True*
⟨*proof*⟩

definition *L2Tcorres st A C* = *corresXF st* (λ*r* -. *r*) (λ*r* -. *r*) (λ-. *True*) *A C*

lemma *L2Tcorres-id*:
L2Tcorres id C C
⟨*proof*⟩

lemma *L2Tcorres-fail*:
L2Tcorres st L2-fail X
⟨*proof*⟩

lemma *admissible-nondet-ord-L2Tcorres* [*corres-admissible*]:
ccpo.admissible Inf (≥) (λ*A*. *L2Tcorres st A C*)
⟨*proof*⟩

lemma *L2Tcorres-top* [*corres-top*]: *L2Tcorres st* ⊤ *C*
⟨*proof*⟩

definition *abs-guard st A C* ≡ ∀ *s*. *A* (*st s*) ⟶ *C s*
definition *abs-expr st P A C* ≡ ∀ *s*. *P* (*st s*) ⟶ *C s* = *A* (*st s*)
definition *abs-modifies st P A C* ≡ ∀ *s*. *P* (*st s*) ⟶ *st* (*C s*) = *A* (*st s*)

definition *struct-rewrite-guard A C* ≡ ∀ *s*. *A s* ⟶ *C s*
definition *struct-rewrite-expr P A C* ≡ ∀ *s*. *P s* ⟶ *C s* = *A s*
definition *struct-rewrite-modifies P A C* ≡ ∀ *s*. *P s* ⟶ *C s* = *A s*

named-theorems *heap-abs*

named-theorems *heap-abs-fo*

named-theorems *derived-heap-defs* **and**
valid-array-defs **and**
heap-upd-cong **and**
valid-same-typ-descs

lemma *deepen-heap-upd-cong*: *f* = *f'* ⟹ *upd f s* = *upd f' s*
⟨*proof*⟩

lemma *deepen-heap-map-cong*: $f = f' \Longrightarrow upd\ f\ p\ s = upd\ f'\ p\ s$
⟨*proof*⟩

lemma *abs-expr-fun-app2* [*heap-abs-fo*]:
⟦ *abs-expr st P f f'*;
abs-expr st Q g g' ⟧ ⟹
abs-expr st (λ*s*. *P s* ∧ *Q s*) (λ*s a*. *f s a* (*g s a*)) (λ*s a*. *f' s a* $ *g' s a*)
⟨*proof*⟩

lemma *abs-expr-fun-app* [*heap-abs-fo*]:
⟦ *abs-expr st Y x x'*; *abs-expr st X f f'* ⟧ ⟹
abs-expr st (λ*s*. *X s* ∧ *Y s*) (λ*s*. *f s* (*x s*)) (λ*s*. *f' s* $ *x' s*)
⟨*proof*⟩

lemma *abs-expr-Pair* [*heap-abs*]:
abs-expr st X f1 f1' ⟹ *abs-expr st Y f2 f2'* ⟹
abs-expr st (λ*s*. *X s* ∧ *Y s*) (λ*s*. (*f1 s*, *f2 s*)) (λ*s*. (*f1' s*, *f2' s*))
⟨*proof*⟩

lemma *abs-expr-constant* [*heap-abs*]:
abs-expr st (λ-. *True*) (λ*s*. *a*) (λ*s*. *a*)
⟨*proof*⟩

lemma *abs-guard-expr* [*heap-abs*]:
abs-expr st P a' a ⟹ *abs-guard st* (λ*s*. *P s* ∧ *a' s*) *a*
⟨*proof*⟩

lemma *abs-guard-constant* [*heap-abs*]:
abs-guard st (λ-. *P*) (λ-. *P*)
⟨*proof*⟩

lemma *abs-guard-conj* [*heap-abs*]:
⟦ *abs-guard st G G'*; *abs-guard st H H'* ⟧
⟹ *abs-guard st* (λ*s*. *G s* ∧ *H s*) (λ*s*. *G' s* ∧ *H' s*)
⟨*proof*⟩

lemma *L2Tcorres-modify* [*heap-abs*]:
⟦ *struct-rewrite-modifies P b c*; *abs-guard st P' P*;
abs-modifies st Q a b ⟧ ⟹
L2Tcorres st (*L2-seq* (*L2-guard* (λ*s*. *P' s* ∧ *Q s*)) (λ-. (*L2-modify a*)))
(*L2-modify c*)
⟨*proof*⟩

lemma *L2Tcorres-gets* [*heap-abs*]:
⟦ *struct-rewrite-expr P b c*; *abs-guard st P' P*;

abs-expr st Q a b ⟧ ⟹
L2Tcorres st (*L2-seq* (*L2-guard* (λ*s*. *P′ s* ∧ *Q s*)) (λ-. *L2-gets a n*)) (*L2-gets c n*)
⟨*proof*⟩

lemma *L2Tcorres-gets-const* [*heap-abs*]:
L2Tcorres st (*L2-gets* (λ-. *a*) *n*) (*L2-gets* (λ-. *a*) *n*)
⟨*proof*⟩

lemma *L2Tcorres-guard* [*heap-abs*]:
⟦ *struct-rewrite-guard b c*; *abs-guard st a b* ⟧ ⟹
L2Tcorres st (*L2-guard a*) (*L2-guard c*)
⟨*proof*⟩

lemma *L2Tcorres-while* [*heap-abs*]:
assumes *body-corres* [*simplified THIN-def*,*rule-format*]:
PROP THIN (⋀*x*. *L2Tcorres st* (*B′ x*) (*B x*))
and *cond-rewrite* [*simplified THIN-def*,*rule-format*]:
PROP THIN (⋀*r*. *struct-rewrite-expr* (*G r*) (*C′ r*) (*C r*))
and *guard-abs*[*simplified THIN-def*,*rule-format*]:
PROP THIN (⋀*r*. *abs-guard st* (*G′ r*) (*G r*))
and *guard-impl-cond*[*simplified THIN-def*,*rule-format*]:
PROP THIN (⋀*r*. *abs-expr st* (*H r*) (*C″ r*) (*C′ r*))
shows *L2Tcorres st* (*L2-guarded-while* (λ*i s*. *G′ i s* ∧ *H i s*) *C″ B′ i n*) (*L2-while C B i n*)
⟨*proof*⟩

named-theorems *abs-spec*

definition *abs-spec st P* (*A* :: (*′a* × *′a*) *set*) (*C* :: (*′c* × *′c*) *set*)
≡ (∀ *s t*. *P* (*st s*) ⟶ (((*s*, *t*) ∈ *C*) ⟶ ((*st s*, *st t*) ∈ *A*)))
∧ (∀ *s*. *P* (*st s*) ⟶ (∃ *x*. (*st s*, *x*) ∈ *A*) ⟶ (∃ *x*. (*s*, *x*) ∈ *C*))

lemma *L2Tcorres-spec* [*heap-abs*]:
⟦ *abs-spec st P A C* ⟧
⟹ *L2Tcorres st* (*L2-seq* (*L2-guard P*) (λ-. (*L2-spec A*))) (*L2-spec C*)
⟨*proof*⟩

definition *abs-assume st P* (*A* :: *′a* ⇒ (*′b* × *′a*) *set*) (*C* :: *′c* ⇒ (*′b* × *′c*) *set*)
≡ (∀ *s t r*. *P* (*st s*) ⟶ (((*r*, *t*) ∈ *C s*) ⟶ ((*r*, *st t*) ∈ *A* (*st s*))))

lemma *refines-assume-result-and-state′*:
refines (*assume-result-and-state P*) (*assume-result-and-state Q*) *s t R*
if *sim-set* (λ(*v*, *s*) (*w*, *t*). *R* (*Result v*, *s*) (*Result w*, *t*)) (*P s*) (*Q t*)
⟨*proof*⟩

lemma *L2Tcorres-assume* [*heap-abs*]:
⟦ *abs-assume st P A C* ⟧
⟹ *L2Tcorres st* (*L2-seq* (*L2-guard P*) (λ-. (*L2-assume A*))) (*L2-assume C*)
⟨*proof*⟩

lemma *abs-spec-constant* [*heap-abs*]:
abs-spec st (λ-. *True*) {(*a*, *b*). *C*} {(*a*, *b*). *C*}
⟨*proof*⟩

lemma *L2Tcorres-condition* [*heap-abs*]:
⟦*PROP THIN* (*Trueprop* (*L2Tcorres st L L′*));
PROP THIN (*Trueprop* (*L2Tcorres st R R′*));
PROP THIN (*Trueprop* (*struct-rewrite-expr P C′ C*));
PROP THIN (*Trueprop* (*abs-guard st P′ P*));
PROP THIN (*Trueprop* (*abs-expr st Q C″ C′*))⟧ ⟹
L2Tcorres st (*L2-seq* (*L2-guard* (λ*s*. *P′ s* ∧ *Q s*)) (λ-. *L2-condition C″ L R*))
(*L2-condition C L′ R′*)
⟨*proof*⟩

lemma *L2Tcorres-seq* [*heap-abs*]:
⟦*PROP THIN* (*Trueprop* (*L2Tcorres st L′ L*)); *PROP THIN* (⋀*r*. *L2Tcorres st*
(*R′ r*) (*R r*)) ⟧
⟹ *L2Tcorres st* (*L2-seq L′ R′*) (*L2-seq L R*)
⟨*proof*⟩

lemma *L2Tcorres-guarded-simple* [*heap-abs*]:
assumes *b-c*: *struct-rewrite-guard b c*
assumes *a-b*: *abs-guard st a b*
assumes *f-g*: ⋀*s s′*. *c s* ⟹ *s′* = *st s* ⟹ *L2Tcorres st f g*
shows *L2Tcorres st* (*L2-guarded a f*) (*L2-guarded c g*)
⟨*proof*⟩

lemma *L2Tcorres-catch* [*heap-abs*]:
⟦*PROP THIN* (*Trueprop* (*L2Tcorres st L L′*));
PROP THIN (⋀*r*. *L2Tcorres st* (*R r*) (*R′ r*))
⟧ ⟹ *L2Tcorres st* (*L2-catch L R*) (*L2-catch L′ R′*)
⟨*proof*⟩

lemma *corresXF-return-same*:
corresXF st (λ*r* -. *r*) (λ*r* -. *r*) (λ-. *True*) (*return e*) (*return e*)
⟨*proof*⟩

lemma *corresXF-yield-same*:
corresXF st (λ*r* -. *r*) (λ*r* -. *r*) (λ-. *True*) (*yield e*) (*yield e*)
⟨*proof*⟩

lemma *L2-try-catch*: *L2-try L = L2-catch L* (λ*e*. *yield* (*to-xval e*))
⟨*proof*⟩

lemma *L2Tcorres-try* [*heap-abs*]:
⟦ *L2Tcorres st L L'* ⟧ ⟹ *L2Tcorres st* (*L2-try L*) (*L2-try L'*)
⟨*proof*⟩

lemma *L2Tcorres-unknown* [*heap-abs*]:
L2Tcorres st (*L2-unknown ns*) (*L2-unknown ns*)
⟨*proof*⟩

lemma *L2Tcorres-throw* [*heap-abs*]:
L2Tcorres st (*L2-throw x n*) (*L2-throw x n*)
⟨*proof*⟩

lemma *L2Tcorres-split* [*heap-abs*]:
⟦ ⋀*x y*. *L2Tcorres st* (*P x y*) (*P' x y*) ⟧ ⟹
L2Tcorres st (*case a of* (*x*, *y*) ⇒ *P x y*) (*case a of* (*x*, *y*) ⇒ *P' x y*)
⟨*proof*⟩

lemma *L2Tcorres-seq-unused-result* [*heap-abs*]:
⟦*PROP THIN* (*Trueprop* (*L2Tcorres st L L'*)); *PROP THIN* (*Trueprop* (*L2Tcorres st R R'*)) ⟧
⟹ *L2Tcorres st* (*L2-seq L* (λ-. *R*)) (*L2-seq L'* (λ-. *R'*))
⟨*proof*⟩

lemma *abs-expr-split* [*heap-abs*]:
⟦ ⋀*a b*. *abs-expr st* (*P a b*) (*A a b*) (*C a b*) ⟧
⟹ *abs-expr st* (*case r of* (*a*, *b*) ⇒ *P a b*)
(*case r of* (*a*, *b*) ⇒ *A a b*) (*case r of* (*a*, *b*) ⇒ *C a b*)
⟨*proof*⟩

lemma *abs-guard-split* [*heap-abs*]:
⟦ ⋀*a b*. *abs-guard st* (*A a b*) (*C a b*) ⟧
⟹ *abs-guard st* (*case r of* (*a*, *b*) ⇒ *A a b*) (*case r of* (*a*, *b*) ⇒ *C a b*)
⟨*proof*⟩

lemma *L2Tcorres-abstract-fail* [*heap-abs*]:
L2Tcorres st L2-fail L2-fail
⟨*proof*⟩

lemma *abs-expr-id* [*heap-abs*]:
abs-expr id (λ-. *True*) *A A*
⟨*proof*⟩

lemma *abs-expr-lambda-null* [*heap-abs*]:
abs-expr st P A C ⟹ *abs-expr st P* (λ*s r*. *A s*) (λ*s r*. *C s*)
⟨*proof*⟩

lemma *abs-modify-id* [*heap-abs*]:
abs-modifies id (λ-. *True*) *A A*
⟨*proof*⟩

lemma *corresXF-exec-concrete* [*intro?*]:
corresXF id ret-xf ex-xf P A C ⟹ *corresXF st ret-xf ex-xf P* (*exec-concrete st A*) *C*
⟨*proof*⟩

lemma *L2Tcorres-exec-concrete* [*heap-abs*]:
L2Tcorres id A C ⟹ *L2Tcorres st* (*exec-concrete st* (*L2-call A emb ns*)) (*L2-call C emb ns*)
⟨*proof*⟩

lemma *L2Tcorres-exec-concrete-simpl* [*heap-abs*]:
L2Tcorres id A C ⟹ *L2Tcorres st* (*exec-concrete st* (*L2-call-L1 arg-xf gs ret-xf A*)) (*L2-call-L1 arg-xf gs ret-xf C*)
⟨*proof*⟩

lemma *corresXF-exec-abstract* [*intro?*]:
corresXF st ret-xf ex-xf P A C ⟹ *corresXF id ret-xf ex-xf P* (*exec-abstract st A*) *C*
⟨*proof*⟩

lemma *L2Tcorres-exec-abstract* [*heap-abs*]:
L2Tcorres st A C ⟹ *L2Tcorres id* (*exec-abstract st* (*L2-call A emb ns*)) (*L2-call C emb ns*)
⟨*proof*⟩

lemma *L2Tcorres-call* [*heap-abs*]:
L2Tcorres st A C ⟹ *L2Tcorres st* (*L2-call A emb ns*) (*L2-call C emb ns*)
⟨*proof*⟩

named-theorems
valid-implies-c-guard **and**
read-commutes **and**
write-commutes **and**
field-write-commutes **and**
write-valid-preservation **and**
lift-heap-update-padding-heap-update-conv

locale *valid-implies-cguard* =
fixes *st*::'*s* ⇒ '*t*
fixes *v*::'*t* ⇒ '*a*::*c-type ptr* ⇒ *bool*

assumes *valid-implies-c-guard*[*valid-implies-c-guard*]: $v\ (st\ s)\ p \Longrightarrow$ *c-guard* p

locale *read-simulation* =
fixes $st :: 's \Rightarrow 't$
fixes $v :: 't \Rightarrow 'a$::*c-type ptr* $\Rightarrow$ *bool*
fixes $r :: 't \Rightarrow 'a\ ptr \Rightarrow 'a$
fixes *t-hrs*$::'s \Rightarrow$ *heap-raw-state*
assumes *read-commutes*[*read-commutes*]: $v\ (st\ s)\ p \Longrightarrow$
$r\ (st\ s)\ p =$ *h-val* (*hrs-mem* (*t-hrs* s)) p

locale *write-simulation* =
heap-raw-state t-hrs t-hrs-upd
for
t-hrs :: ($'s \Rightarrow$ *heap-raw-state*) **and**
t-hrs-upd::(*heap-raw-state* $\Rightarrow$ *heap-raw-state*) $\Rightarrow\ 's \Rightarrow\ 's +$
fixes $st :: 's \Rightarrow 't$
fixes $v :: 't \Rightarrow 'a$::*mem-type ptr* $\Rightarrow$ *bool*
fixes $w :: 'a\ ptr \Rightarrow ('a \Rightarrow 'a)\ \Rightarrow 't \Rightarrow 't$

assumes *write-padding-commutes*[*write-commutes*]: $v\ (st\ s)\ p \Longrightarrow$ *length* $bs =$
size-of $TYPE('a) \Longrightarrow$
st (*t-hrs-upd* (*hrs-mem-update* (*heap-update-padding* $p\ x\ bs$)) s) =
$w\ p\ (\lambda\text{-}.\ x)\ \ (st\ s)$

begin
lemma *write-commutes*[*write-commutes*]:
assumes *valid*: $v\ (st\ s)\ p$
shows st (*t-hrs-upd* (*hrs-mem-update* (*heap-update* $p\ x$)) s) =
$w\ p\ (\lambda\text{-}.\ x)\ (st\ s)$
⟨*proof*⟩

lemma *lift-heap-update-padding-heap-update-conv*[*lift-heap-update-padding-heap-update-conv*]:
$v\ (st\ s)\ p \Longrightarrow$ *length* $bs =$ *size-of* $TYPE('a) \Longrightarrow$
st (*t-hrs-upd* (*hrs-mem-update* (*heap-update-padding* $p\ x\ bs$)) s) =
st (*t-hrs-upd* (*hrs-mem-update* (*heap-update* $p\ x$)) s)
⟨*proof*⟩

lemma *write-commutes-atomic*: $\forall\ s\ p\ x.\ v\ (st\ s)\ p \longrightarrow$
st (*t-hrs-upd* (*hrs-mem-update* (*heap-update* $p\ x$)) s) =
$w\ p\ (\lambda\text{-}.\ x)\ (st\ s)$
⟨*proof*⟩

end

locale *write-preserves-valid* =
fixes $v :: 't \Rightarrow 'a$::*c-type ptr* $\Rightarrow$ *bool*
fixes $w :: 'a\ ptr \Rightarrow ('b \Rightarrow 'b) \Rightarrow 't \Rightarrow 't$
assumes *valid-preserved*: $v\ (w\ p'\ f\ s)\ p = v\ s\ p$

begin
lemma *valid-preserved-pointless*[*simp*]: *v* (*w p' f s*) = *v s*
⟨*proof*⟩
end

locale *valid-only-typ-desc-dependent* =
fixes *t-hrs* :: ('*s* ⇒ *heap-raw-state*)
fixes *st* ::'*s* ⇒ '*t*
fixes *v* ::'*t* ⇒ '*a*::*c-type ptr* ⇒ *bool*
assumes *valid-same-typ-desc* [*valid-same-typ-descs*]: *hrs-htd* (*t-hrs s*) = *hrs-htd* (*t-hrs t*) ⟹ *v* (*st s*) *p* = *v* (*st t*) *p*

locale *heap-typing-simulation* =
open-types $\mathcal{T}$ + *t-hrs*: *heap-raw-state t-hrs t-hrs-upd* + *heap-typing-state heap-typing heap-typing-upd*
for
$\mathcal{T}$ **and**
t-hrs :: ('*s* ⇒ *heap-raw-state*) **and**
t-hrs-upd :: (*heap-raw-state* ⇒ *heap-raw-state*) ⇒ ('*s* ⇒ '*s*) **and**
heap-typing :: '*t* ⇒ *heap-typ-desc* **and**
heap-typing-upd :: (*heap-typ-desc* ⇒ *heap-typ-desc*) ⇒ '*t* ⇒ '*t* +
fixes *st* ::'*s* ⇒ '*t*
assumes *heap-typing-commutes*[*simp*]: *heap-typing* (*st s*) = *hrs-htd* (*t-hrs s*)
assumes *lift-heap-update-list-stack-byte-independent*:
(⋀*i*. *i* < *length bs* ⟹ *root-ptr-valid* (*hrs-htd* (*t-hrs s*)) ((*p*::*stack-byte ptr*) +$_p$ *int i*)) ⟹
st (*t-hrs-upd* (*hrs-mem-update* (*heap-update-list* (*ptr-val p*) *bs*)) *s*) = *st s*
assumes *st-eq-upto-padding*:
mem-type-u t ⟹ *padding-closed-under-all-fields t* ⟹
ptr-valid-u t (*hrs-htd* (*t-hrs s*)) *a* ⟹ *eq-upto-padding t bs bs'* ⟹
st (*t-hrs-upd* (*hrs-mem-update* (*heap-update-list a bs*)) *s*) =
st (*t-hrs-upd* (*hrs-mem-update* (*heap-update-list a bs'*)) *s*)
begin

lemma *heap-typing-upd-commutes*: *heap-typing* (*heap-typing-upd f* (*st s*)) = *hrs-htd* (*t-hrs* (*t-hrs-upd* (*hrs-htd-update f*) *s*))
⟨*proof*⟩

lemma *write-simulation-alt*:
assumes *v*: ⋀*s p*. *v* (*st s*) *p* ⟹ *ptr-valid* (*hrs-htd* (*t-hrs s*)) *p*
assumes ∗: ⋀*s* (*p*::'*a*::*xmem-type ptr*) *x*. *v* (*st s*) *p* ⟹
st (*t-hrs-upd* (*hrs-mem-update* (*heap-update p x*)) *s*) = *w p* (λ-. *x*) (*st s*)
shows *write-simulation t-hrs t-hrs-upd st v w*
⟨*proof*⟩

end

locale *typ-heap-simulation* =

heap-raw-state t-hrs t-hrs-update +
read-simulation st v r t-hrs +
write-simulation t-hrs t-hrs-update st v w +
write-preserves-valid v w +
valid-implies-cguard st v +
valid-only-typ-desc-dependent t-hrs st v +
pointer-lense r w
for
st:: *'s* ⇒ *'t* **and**
r:: *'t* ⇒ (*'a*::*xmem-type*) *ptr* ⇒ *'a* **and**
w:: *'a ptr* ⇒ (*'a* ⇒ *'a*) ⇒ *'t* ⇒ *'t* **and**
v:: *'t* ⇒ (*'a*::*xmem-type*) *ptr* ⇒ *bool* **and**
t-hrs :: *'s* ⇒ *heap-raw-state* **and**
t-hrs-update:: (*heap-raw-state* ⇒ *heap-raw-state*) ⇒ *'s* ⇒ *'s*
begin

lemma *write-valid-preservation* [*write-valid-preservation*]:
shows *v* (*st* (*t-hrs-update* (*hrs-mem-update* (*heap-update q x*)) *s*)) *p* = *v* (*st s*) *p*
⟨*proof*⟩

lemma *write-padding-valid-preservation* [*write-valid-preservation*]:
shows *v* (*st* (*t-hrs-update* (*hrs-mem-update* (*heap-update-padding q x bs*)) *s*)) *p* = *v* (*st s*) *p*
⟨*proof*⟩

end

locale *stack-simulation* =
heap-typing-simulation $\mathcal{T}$ *t-hrs t-hrs-update heap-typing heap-typing-upd st* +
typ-heap-typing r w heap-typing heap-typing-upd $\mathcal{S}$
for
$\mathcal{T}$ **and**
st:: *'s* ⇒ *'t* **and**
r:: *'t* ⇒ (*'a*::*xmem-type*) *ptr* ⇒ *'a* **and**
w:: *'a ptr* ⇒ (*'a* ⇒ *'a*) ⇒ *'t* ⇒ *'t* **and**
t-hrs :: *'s* ⇒ *heap-raw-state* **and**
t-hrs-update:: (*heap-raw-state* ⇒ *heap-raw-state*) ⇒ *'s* ⇒ *'s* **and**
heap-typing :: *'t* ⇒ *heap-typ-desc* **and**
heap-typing-upd :: (*heap-typ-desc* ⇒ *heap-typ-desc*) ⇒ *'t* ⇒ *'t* **and**
$\mathcal{S}$:: *addr set* +
assumes *sim-stack-alloc*:
⋀*p d vs bs s n*.
(*p*, *d*) ∈ *stack-allocs n* $\mathcal{S}$ *TYPE*(*'a*) (*hrs-htd* (*t-hrs s*)) ⟹ *length vs* = *n* ⟹ *length bs* = *n* ∗ *size-of TYPE* (*'a*) ⟹
st (*t-hrs-update* (*hrs-mem-update* (*fold* (λ*i*. *heap-update-padding* (*p* +$_p$ *int i*) (*vs*!*i*) (*take* (*size-of TYPE*(*'a*)) (*drop* (*i* ∗ *size-of TYPE*(*'a*)) *bs*))) [*0*..<*n*]) ∘ *hrs-htd-update* (λ-. *d*)) *s*) =

(*fold* (λ*i*. *w* (*p* +$_p$ *int i*) (λ-. (*vs* ! *i*))) [*0*..<*n*]) (*heap-typing-upd* (λ-. *d*) (*st s*))

assumes *sim-stack-release*: ⋀*p s n*. (⋀*i*. *i* < *n* ⟹ *root-ptr-valid* (*hrs-htd* (*t-hrs s*)) (*p* +$_p$ *int i*)) ⟹
length bs = *n* ∗ *size-of TYPE*('*a*) ⟹
st (*t-hrs-update* (*hrs-mem-update* (*heap-update-list* (*ptr-val p*) *bs*) ∘ *hrs-htd-update* (*stack-releases n p*)) *s*) =
((*heap-typing-upd* (*stack-releases n p*) (*fold* (λ*i*. *w* (*p* +$_p$ *int i*) (λ-. *c-type-class.zero*)) [*0*..<*n*] (*st s*))))

assumes *stack-byte-zero*: ⋀*p d i*. (*p*, *d*) ∈ *stack-allocs n* 𝒮 *TYPE*('*a*) (*hrs-htd* (*t-hrs s*)) ⟹ *i* < *n* ⟹ *r* (*st s*) (*p* +$_p$ *int i*) = *ZERO*('*a*)

lemma (**in** *typ-heap-simulation*) *L2Tcorres-IO-modify-paddingE* [*heap-abs*]:
assumes *abs-expr st P a c*
shows *L2Tcorres st* (*L2-seq* (*L2-guard* (λ*t*. *v t p* ∧ *P t*)) (λ-. (*L2-modify* (λ*s*. *w p* (λ-. *a s*) *s*))))
(*IO-modify-heap-paddingE* (*p*::'*a ptr*) *c*)
⟨*proof*⟩

locale *typ-heap-typing-stack-simulation* =
typ-heap-simulation st r w v t-hrs t-hrs-update +
stack-simulation 𝒯 *st r w t-hrs t-hrs-update heap-typing heap-typing-upd* 𝒮
for
𝒯 **and**
st:: '*s* ⇒ '*t* **and**
r:: '*t* ⇒ ('*a*::*xmem-type*) *ptr* ⇒ '*a* **and**
w:: '*a ptr* ⇒ ('*a* ⇒ '*a*) ⇒ '*t* ⇒ '*t* **and**
v:: '*t* ⇒ ('*a*::*xmem-type*) *ptr* ⇒ *bool* **and**
t-hrs :: '*s* ⇒ *heap-raw-state* **and**
t-hrs-update:: (*heap-raw-state* ⇒ *heap-raw-state*) ⇒ '*s* ⇒ '*s* **and**
heap-typing :: '*t* ⇒ *heap-typ-desc* **and**
heap-typing-upd :: (*heap-typ-desc* ⇒ *heap-typ-desc*) ⇒ '*t* ⇒ '*t* **and**
𝒮:: *addr set*
begin

sublocale *monolithic*: *stack-heap-raw-state t-hrs t-hrs-update* 𝒮
⟨*proof*⟩

definition *rel-split-heap* ≡ λs_c s_a. s_a = *st* s_c

lemma *rel-split-heap-stack-free-eq*:
rel-split-heap s_c s_a ⟹ *stack-free* (*hrs-htd* (*t-hrs* s_c)) = *stack-free* (*heap-typing* s_a)
⟨*proof*⟩

definition *rel-stack-free-eq* **where**
rel-stack-free-eq s_c s_a $\equiv$ *stack-free* (*hrs-htd* (*t-hrs* s_c)) = *stack-free* (*heap-typing* s_a)

lemma *rel-prod-rel-split-heap-conv*:
rel-prod (=) *rel-split-heap* = (λ(v, t) (r, s).
s = *st* t $\wedge$ (*case* v *of Exn* x $\Rightarrow$ r = *Exn* x | *Result* x $\Rightarrow$ r = *Result* x))
⟨*proof*⟩

lemma *L2Tcorres-refines*:
L2Tcorres st f_a f_c $\Longrightarrow$ *refines* f_c f_a s (*st* s) (*rel-prod* (=) *rel-split-heap*)
⟨*proof*⟩

lemma *refines-L2Tcorres*:
assumes f: $\bigwedge s$. *refines* f_c f_a s (*st* s) (*rel-prod* (=) *rel-split-heap*)
shows *L2Tcorres st* f_a f_c
⟨*proof*⟩

lemma *L2Tcorres-refines-conv*:
L2Tcorres st f_a f_c $\longleftrightarrow$ ($\forall$ s. *refines* f_c f_a s (*st* s) (*rel-prod* (=) *rel-split-heap*))
⟨*proof*⟩

lemma *sim-guard-with-fresh-stack-ptr*:
fixes f_c:: *'a ptr* $\Rightarrow$ (*'b*, *'c*, *'s*) *exn-monad*
assumes *init*: $init_a$ (*st* s) = $init_c$ s
assumes f: $\bigwedge s$ p::*'a ptr*. *refines* (f_c p) (f_a p) s (*st* s) (*rel-prod* (=) *rel-split-heap*)
shows *refines*
(*monolithic.with-fresh-stack-ptr* n $init_c$ f_c)
(*guard-with-fresh-stack-ptr* n $init_a$ f_a) s (*st* s)
(*rel-prod* (=) *rel-split-heap*)
⟨*proof*⟩

lemma *sim-with-fresh-stack-ptr*:
fixes f_c:: *'a ptr* $\Rightarrow$ (*'b*, *'c*, *'s*) *exn-monad*
assumes *init*: $init_a$ (*st* s) = $init_c$ s
assumes f: $\bigwedge s$ p::*'a ptr*. *refines* (f_c p) (f_a p) s (*st* s) (*rel-prod* (=) *rel-split-heap*)
assumes *typing-unchanged*: $\bigwedge s$ p::*'a ptr*. (f_c p) $\cdot$ s *?*{|λr t. *typing.unchanged-typing-on* $\mathcal{S}$ s t|}
shows *refines*
(*monolithic.with-fresh-stack-ptr* n $init_c$ f_c)
(*with-fresh-stack-ptr* n $init_a$ f_a) s (*st* s)
(*rel-prod* (=) *rel-split-heap*)
⟨*proof*⟩

lemma *sim-assume-with-fresh-stack-ptr*:
fixes f_c:: *'a ptr* $\Rightarrow$ (*'b*, *'c*, *'s*) *exn-monad*
assumes *init*: $init_a$ (*st* s) = $init_c$ s
assumes f: $\bigwedge s$ p::*'a ptr*. *refines* (f_c p) (f_a p) s (*st* s) (*rel-prod* (=) *rel-split-heap*)

assumes *typing-unchanged*: $\bigwedge s\ p::'a\ ptr.\ (f_c\ p) \cdot s\ ?\{\!|\lambda r\ t.\ typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ t|\!\}$
shows *refines*
(*monolithic.with-fresh-stack-ptr n* $init_c$ f_c)
(*assume-with-fresh-stack-ptr n* $init_a$ f_a) *s* (*st s*)
(*rel-prod* (=) *rel-split-heap*)
⟨*proof*⟩

lemma *L2Tcorres-guard-with-fresh-stack-ptr* [*heap-abs*]:
assumes *rew*: *struct-rewrite-expr P* $init_c'$ $init_c$
assumes *grd*: *abs-guard st P' P*
assumes *expr*: *abs-expr st Q* $init_a$ $init_c'$
assumes *f*[*simplified THIN-def*, *rule-format*]: *PROP THIN* ($\bigwedge p::'a\ ptr.$ *L2Tcorres st* (f_a *p*) (f_c *p*))
shows *L2Tcorres st* (*L2-seq* (*L2-guard* ($\lambda s.$ *P' s* ∧ *Q s*))
(λ-. (*guard-with-fresh-stack-ptr n* $init_a$ (*L2-VARS* f_a *nm*))))
(*monolithic.with-fresh-stack-ptr n* $init_c$ (*L2-VARS* f_c *nm*))
⟨*proof*⟩

lemma *L2Tcorres-with-fresh-stack-ptr*:
assumes *typing-unchanged*: $\bigwedge s\ p::'a\ ptr.\ (f_c\ p) \cdot s\ ?\{\!|\lambda r\ t.\ typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ t|\!\}$
assumes *rew*: *struct-rewrite-expr P* $init_c'$ $init_c$
assumes *grd*: *abs-guard st P' P*
assumes *expr*: *abs-expr st Q* $init_a$ $init_c'$
assumes *f*[*simplified THIN-def*, *rule-format*]: *PROP THIN* ($\bigwedge p::'a\ ptr.$ *L2Tcorres st* (f_a *p*) (f_c *p*))
shows *L2Tcorres st* (*L2-seq* (*L2-guard* ($\lambda s.$ *P' s* ∧ *Q s*))
(λ-. (*with-fresh-stack-ptr n* $init_a$ (*L2-VARS* f_a *nm*))))
(*monolithic.with-fresh-stack-ptr n* $init_c$ (*L2-VARS* f_c *nm*))
⟨*proof*⟩

lemma *L2Tcorres-assume-with-fresh-stack-ptr*[*heap-abs*]:
assumes *typing-unchanged*: $\bigwedge s\ p::'a\ ptr.\ (f_c\ p) \cdot s\ ?\{\!|\lambda r\ t.\ typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ t|\!\}$
assumes *rew*: *struct-rewrite-expr P* $init_c'$ $init_c$
assumes *grd*: *abs-guard st P' P*
assumes *expr*: *abs-expr st Q* $init_a$ $init_c'$
assumes *f*[*simplified THIN-def*, *rule-format*]: *PROP THIN* ($\bigwedge p::'a\ ptr.$ *L2Tcorres st* (f_a *p*) (f_c *p*))
shows *L2Tcorres st* (*L2-seq* (*L2-guard* ($\lambda s.$ *P' s* ∧ *Q s*))
(λ-. (*assume-with-fresh-stack-ptr n* $init_a$ (*L2-VARS* f_a *nm*))))
(*monolithic.with-fresh-stack-ptr n* $init_c$ (*L2-VARS* f_c *nm*))
⟨*proof*⟩

lemma *unchanged-typing-commutes*: *typing.unchanged-typing-on S s t* ⟹ *un-*

changed-typing-on S (st s) (st t)
⟨*proof*⟩
end

named-theorems *read-stack-byte-ZERO-base*
and *read-stack-byte-ZERO-step*
and *read-stack-byte-ZERO-step-subst*

lemma (**in** *open-types*) *ptr-span-with-stack-byte-type-implies-ptr-invalid*:
fixes *p* :: (*'a* :: {*mem-type*, *stack-type*}) *ptr*
assumes ∗: ∀ *a* ∈ *ptr-span p*. *root-ptr-valid d* (*PTR* (*stack-byte*) *a*)
shows ¬ *ptr-valid-u* (*typ-uinfo-t TYPE*(*'a*)) *d* (*ptr-val p*)
⟨*proof*⟩

lemma (**in** *open-types*)
ptr-span-with-stack-byte-type-implies-ZERO[*read-stack-byte-ZERO-base*]:
fixes *p* :: (*'a* :: {*mem-type*, *stack-type*}) *ptr*
assumes ∀ *a* ∈ *ptr-span p*. *root-ptr-valid* (*hrs-htd d*) (*PTR* (*stack-byte*) *a*)
shows *the-default* (*ZERO*(*'a*)) (*plift d p*) = *ZERO*(*'a*)
⟨*proof*⟩

lemma *ptr-span-array-ptr-index-subset-ptr-span*:
fixes *p* :: ((*'a* :: {*array-outer-max-size*})[*'b* :: *array-max-count*]) *ptr*
assumes *i* < *CARD*(*'b*)
shows *ptr-span* (*array-ptr-index p c i*) ⊆ *ptr-span p*
⟨*proof*⟩

lemma *read-stack-byte-ZERO-array-intro*[*read-stack-byte-ZERO-step*]:
fixes *q* :: (*'a* :: {*array-outer-max-size*}[*'b* :: *array-max-count*]) *ptr*
assumes *ptr-span-has-stack-byte-type*:
∀ *a*∈*ptr-span q*. *root-ptr-valid d* (*PTR*(*stack-byte*) *a*)
assumes *subtype-reads-ZERO*:
⋀*p* :: *'a ptr*. ∀ *a*∈*ptr-span p*. *root-ptr-valid d* (*PTR*(*stack-byte*) *a*) ⟹ *r s p* = *ZERO*(*'a*)
shows (*ARRAY i*. *r s* (*array-ptr-index q c i*)) = *ZERO*(*'a*[*'b*])
⟨*proof*⟩

lemma *read-stack-byte-ZERO-array-2dim-intro*[*read-stack-byte-ZERO-step*]:
fixes *q* :: (*'a* :: {*array-inner-max-size*}[*'b* :: *array-max-count*][*'c* :: *array-max-count*]) *ptr*
assumes *ptr-span-has-stack-byte-type*:
∀ *a*∈*ptr-span q*. *root-ptr-valid d* (*PTR*(*stack-byte*) *a*)
assumes *subtype-reads-ZERO*:
⋀*p* :: *'a ptr*. ∀ *a*∈*ptr-span p*. *root-ptr-valid d* (*PTR*(*stack-byte*) *a*) ⟹ *r s p* = *ZERO*(*'a*)
shows (*ARRAY i j*. *r s* (*array-ptr-index* (*array-ptr-index q c i*) *c j*)) = *ZERO*(*'a*[*'b*][*'c*])
⟨*proof*⟩

lemma *read-stack-byte-ZERO-field-intro*[*read-stack-byte-ZERO-step*]:
fixes $q :: 'a :: mem\text{-}type\ ptr$
assumes *ptr-span-has-stack-byte-type*:
$\forall\, a{\in}ptr\text{-}span\ q.\ root\text{-}ptr\text{-}valid\ d\ (PTR(stack\text{-}byte)\ a)$
assumes *subtype-reads-ZERO*:
$\bigwedge p :: 'b :: mem\text{-}type\ ptr.\ \forall\, a{\in}ptr\text{-}span\ p.\ root\text{-}ptr\text{-}valid\ d\ (PTR(stack\text{-}byte)\ a)$
$\Longrightarrow r\ s\ p = ZERO('b)$
assumes *subtype-lookup*:
$field\text{-}lookup\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ f\ 0 = Some\ (typ\text{-}uinfo\text{-}t\ TYPE('b),\ n)$
shows $r\ s\ (PTR('b)\ (\&(q{\rightarrow}f))) = ZERO('b)$
⟨*proof*⟩

context *open-types*
begin

lemma *ptr-span-with-stack-byte-type-implies-read-dedicated-heap-ZERO*[*simp*]:
$\forall\, a{\in}ptr\text{-}span\ p.\ root\text{-}ptr\text{-}valid\ (hrs\text{-}htd\ s)\ (PTR(stack\text{-}byte)\ a) \Longrightarrow$
$read\text{-}dedicated\text{-}heap\ s\ p = ZERO('a{::}\{stack\text{-}type,\ xmem\text{-}type\})$
⟨*proof*⟩

lemma *write-simulationI*:
fixes $R :: 's \Rightarrow 'a{::}xmem\text{-}type\ ptr \Rightarrow 'a$
assumes *fs*: $map\text{-}of\ \mathcal{T}\ (typ\text{-}uinfo\text{-}t\ TYPE('a)) = Some\ fs$
assumes *heap-typing-simulation* $\mathcal{T}$ *t-hrs t-hrs-update heap-typing heap-typing-update l*
and *l-w*: $list\text{-}all2\ (\lambda f\ w.\ \forall\, t\ u\ n\ h\ (p{::}'a\ ptr)\ x.$
$field\text{-}ti\ TYPE('a)\ f = Some\ t \longrightarrow$
$field\text{-}lookup\ (typ\text{-}uinfo\text{-}t\ TYPE('a))\ f\ 0 = Some\ (u,\ n) \longrightarrow$
$ptr\text{-}valid\text{-}u\ u\ (hrs\text{-}htd\ (t\text{-}hrs\ h))\ \&(p{\rightarrow}f) \longrightarrow$
$l\ (t\text{-}hrs\text{-}update\ (hrs\text{-}mem\text{-}update\ (heap\text{-}upd\text{-}list\ (size\text{-}td\ u)\ \&(p{\rightarrow}f)\ (access\text{-}ti\ t\ x)))\ h)$
$= w\ p\ x\ (l\ h))\ fs\ ws$
and *l-u*: $\bigwedge(p{::}'a\ ptr)\ (x{::}'a)\ (s{::}'b).$
$ptr\text{-}valid\ (hrs\text{-}htd\ (t\text{-}hrs\ s))\ p \Longrightarrow$
$l\ (t\text{-}hrs\text{-}update\ (write\text{-}dedicated\text{-}heap\ p\ x)\ s) = u\ (upd\text{-}fun\ p\ (\lambda old.\ merge\text{-}addressable\text{-}fields\ old\ x))\ (l\ s)$
assumes *V*:
$\bigwedge h\ p.\ V\ (l\ h)\ p \longleftrightarrow ptr\text{-}valid\ (hrs\text{-}htd\ (t\text{-}hrs\ h))\ p$
assumes *W*:
$\bigwedge p\ f\ h.\ W\ p\ f\ h =$
$fold\ (\lambda w.\ w\ p\ (f\ (R\ h\ p)))\ ws\ (u\ (upd\text{-}fun\ p\ (\lambda old.\ merge\text{-}addressable\text{-}fields\ old\ (f\ (R\ h\ p))))\ h)$
shows *write-simulation t-hrs t-hrs-update l V W*
⟨*proof*⟩

end

locale *stack-simulation-heap-typing* =
typ-heap-simulation st r w λt p. open-types.ptr-valid 𝒯 (heap-typing t) p t-hrs t-hrs-update +
heap-typing-simulation 𝒯 t-hrs t-hrs-update heap-typing heap-typing-upd st +
typ-heap-typing r w heap-typing heap-typing-upd 𝒮
for
st:: *'s* ⇒ *'t* **and**
r:: *'t* ⇒ (*'a*::{*xmem-type*, *stack-type*}) *ptr* ⇒ *'a* **and**
w:: *'a ptr* ⇒ (*'a* ⇒ *'a*) ⇒ *'t* ⇒ *'t* **and**
t-hrs :: *'s* ⇒ *heap-raw-state* **and**
t-hrs-update:: (*heap-raw-state* ⇒ *heap-raw-state*) ⇒ *'s* ⇒ *'s* **and**
heap-typing :: *'t* ⇒ *heap-typ-desc* **and**
heap-typing-upd :: (*heap-typ-desc* ⇒ *heap-typ-desc*) ⇒ *'t* ⇒ *'t* **and**
𝒮:: *addr set* **and**
𝒯:: (*typ-uinfo* ∗ *qualified-field-name list*) *list* +

assumes *sim-stack-alloc-heap-typing*:
⋀*p d s n*.
(*p*, *d*) ∈ *stack-allocs n 𝒮 TYPE*(*'a*) (*hrs-htd* (*t-hrs s*)) ⟹
st (*t-hrs-update* (*hrs-mem-update* (*fold* (λ*i*. *heap-update* (*p* +$_p$ *int i*) *c-type-class.zero*) [*0*..<*n*]) ∘ *hrs-htd-update* (λ-. *d*)) *s*) =
(*heap-typing-upd* (λ-. *d*) (*st s*))

assumes *sim-stack-release-heap-typing*:
⋀(*p*::*'a ptr*) *s n*. (⋀*i*. *i* < *n* ⟹ *root-ptr-valid* (*hrs-htd* (*t-hrs s*)) (*p* +$_p$ *int i*)) ⟹
st (*t-hrs-update* (*hrs-htd-update* (*stack-releases n p*)) *s*) =
heap-typing-upd (*stack-releases n p*)
(*st* (*t-hrs-update* (*hrs-mem-update* (*fold* (λ*i*. *heap-update* (*p* +$_p$ *int i*) *c-type-class.zero*) [*0*..<*n*])) *s*))

assumes *sim-stack-stack-byte-zero*[*read-stack-byte-ZERO-step*]:
⋀*p s*. ∀ *a*∈*ptr-span p*. *root-ptr-valid* (*hrs-htd* (*t-hrs s*)) (*PTR*(*stack-byte*) *a*) ⟹
r (*st s*) *p* = *ZERO*(*'a*)

begin

lemma *fold-heap-update-simulation*:
assumes *valid*: ⋀*i*. *i* < *n* ⟹ *ptr-valid* (*heap-typing* (*st s*)) (*p* +$_p$ *int i*)
shows *st* (*t-hrs-update* (*hrs-mem-update* (*fold* (λ*i*. *heap-update* (*p* +$_p$ *int i*) (*vs i*)) [*0*..<*n*])) *s*) =
fold (λ*i*. *w* (*p* +$_p$ *int i*) (λ-. *vs i*)) [*0*..<*n*] (*st s*)
⟨*proof*⟩

lemma *fold-heap-update-padding-simulation*:
assumes *valid*: ⋀*i*. *i* < *n* ⟹ *ptr-valid* (*heap-typing* (*st s*)) (*p* +$_p$ *int i*)
assumes *lbs*: *length bs* = *n* ∗ *size-of TYPE*(*'a*)
shows *st* (*t-hrs-update* (*hrs-mem-update* (*fold* (λ*i*. *heap-update-padding* (*p* +$_p$ *int i*) (*vs i*) (*take* (*size-of TYPE*(*'a*)) (*drop* (*i* ∗ *size-of TYPE*(*'a*)) *bs*))) [*0*..<*n*])) *s*) =

fold (λ*i*. *w* (*p* +$_p$ *int i*) (λ-. *vs i*)) [*0*..<*n*] (*st s*)
⟨*proof*⟩

lemma *sim-stack-alloc′*:
assumes *alloc*: (*p*, *d*) ∈ *stack-allocs n* $\mathcal{S}$ *TYPE*('*a*) (*hrs-htd* (*t-hrs s*))
assumes *len*: *length vs* = *n*
assumes *lbs*: *length bs* = *n* ∗ *size-of TYPE*('*a*)
shows *st* (*t-hrs-update* (*hrs-mem-update* (*fold* (λ*i*. *heap-update-padding* (*p* +$_p$ *int i*) (*vs*!*i*) (*take* (*size-of TYPE*('*a*)) (*drop* (*i* ∗ *size-of TYPE*('*a*)) *bs*))) [*0*..<*n*]) ∘ *hrs-htd-update* (λ-. *d*)) *s*) =
(*fold* (λ*i*. *w* (*p* +$_p$ *int i*) (λ-. (*vs* ! *i*))) [*0*..<*n*]) (*heap-typing-upd* (λ-. *d*) (*st s*))
⟨*proof*⟩

lemma *sim-stack-release′*:
fixes *p* :: '*a ptr*
assumes *roots*: ⋀*i*. *i* < *n* ⟹ *root-ptr-valid* (*hrs-htd* (*t-hrs s*)) (*p* +$_p$ *int i*)
shows *st* (*t-hrs-update* (*hrs-htd-update* (*stack-releases n p*)) *s*) =
((*heap-typing-upd* (*stack-releases n p*) ((*fold* (λ*i*. *w* (*p* +$_p$ *int i*) (λ-. *c-type-class.zero*)) [*0*..<*n*]) (*st s*))))
⟨*proof*⟩

lemma *sim-stack-release″*:
fixes *p* :: '*a ptr*
assumes *roots*: ⋀*i*. *i* < *n* ⟹ *root-ptr-valid* (*hrs-htd* (*t-hrs s*)) (*p* +$_p$ *int i*)
assumes *lbs*: *length bs* = *n* ∗ *size-of TYPE*('*a*)
shows *st* (*t-hrs-update* (*hrs-mem-update* (*heap-update-list* (*ptr-val p*) *bs*) *o hrs-htd-update* (*stack-releases n p*)) *s*) =
((*heap-typing-upd* (*stack-releases n p*) ((*fold* (λ*i*. *w* (*p* +$_p$ *int i*) (λ-. *c-type-class.zero*)) [*0*..<*n*]) (*st s*))))
⟨*proof*⟩

lemma *stack-byte-zero′*:
assumes (*p*, *d*) ∈ *stack-allocs n* $\mathcal{S}$ *TYPE*('*a*) (*hrs-htd* (*t-hrs s*))
assumes *i* < *n*
shows *r* (*st s*) (*p* +$_p$ *int i*) = *ZERO*('*a*)
⟨*proof*⟩

sublocale *stack-simulation*
⟨*proof*⟩

sublocale *typ-heap-typing-stack-simulation* $\mathcal{T}$ *st r w* λ*t p*. *open-types.ptr-valid* $\mathcal{T}$ (*heap-typing t*) *p t-hrs t-hrs-update heap-typing heap-typing-upd* $\mathcal{S}$
⟨*proof*⟩

end

definition
valid-struct-field
:: *string list*
⇒ (('*p*::*xmem-type*) ⇒ ('*f*::*xmem-type*))
⇒ (('*f* ⇒ '*f*) ⇒ ('*p* ⇒ '*p*))
⇒ ('*s* ⇒ *heap-raw-state*)
⇒ ((*heap-raw-state* ⇒ *heap-raw-state*) ⇒ '*s* ⇒ '*s*)
⇒ *bool*
where
valid-struct-field field-name field-getter field-setter t-hrs t-hrs-update ≡
(*lense field-getter field-setter*
∧ *field-ti TYPE*('*p*) *field-name* =
Some (*adjust-ti* (*typ-info-t TYPE*('*f*)) *field-getter* (*field-setter* ∘ (λ*x* -. *x*)))
∧ (∀ *p* :: '*p ptr*. *c-guard p* ⟶ *c-guard* (*Ptr* &(*p*→*field-name*) :: '*f ptr*))
∧ *lense t-hrs t-hrs-update*)

lemma *typ-heap-simulation-get-hvalD*:
⟦ *typ-heap-simulation st r w v*
t-hrs t-hrs-update; *v* (*st s*) *p* ⟧ ⟹
h-val (*hrs-mem* (*t-hrs s*)) *p* = *r* (*st s*) *p*
⟨*proof*⟩

lemma *valid-struct-fieldI* [*intro*]:
fixes *field-getter* :: ('*a*::*xmem-type*) ⇒ ('*f*::*xmem-type*)
shows ⟦
⋀*s f*. *f* (*field-getter s*) = (*field-getter s*) ⟹ *field-setter f s* = *s*;
⋀*s f f'*. *f* (*field-getter s*) = *f'* (*field-getter s*) ⟹ *field-setter f s* = *field-setter f' s*;
⋀*s f*. *field-getter* (*field-setter f s*) = *f* (*field-getter s*);
⋀*s f g*. *field-setter f* (*field-setter g s*) = *field-setter* (*f* ∘ *g*) *s*;
field-ti TYPE('*a*) *field-name* =
Some (*adjust-ti* (*typ-info-t TYPE*('*f*)) *field-getter* (*field-setter* ∘ (λ*x* -. *x*)));
⋀(*p*::'*a ptr*). *c-guard p* ⟹ *c-guard* (*Ptr* &(*p*→*field-name*) :: '*f ptr*);
⋀*s x*. *t-hrs* (*t-hrs-update x s*) = *x* (*t-hrs s*);
⋀*s f*. *f* (*t-hrs s*) = *t-hrs s* ⟹ *t-hrs-update f s* = *s*;
⋀*s f f'*. *f* (*t-hrs s*) = *f'* (*t-hrs s*) ⟹ *t-hrs-update f s* = *t-hrs-update f' s*;
⋀*s f g*. *t-hrs-update f* (*t-hrs-update g s*) = *t-hrs-update* (λ*x*. *f* (*g x*)) *s*
⟧ ⟹
valid-struct-field field-name field-getter field-setter t-hrs t-hrs-update
⟨*proof*⟩

lemma *typ-heap-simulation-t-hrs-updateD*:
⟦ *typ-heap-simulation st r w v*
t-hrs t-hrs-update; *v* (*st s*) *p* ⟧ ⟹
st (*t-hrs-update* (*hrs-mem-update* (*heap-update p v'*)) *s*) =
w p (λ*x*. *v'*) (*st s*)
⟨*proof*⟩

lemma *heap-abs-expr-guard* [*heap-abs*]:
⟦ *typ-heap-simulation st getter setter vgetter t-hrs t-hrs-update*;
abs-expr st P x' x ⟧ ⟹
abs-guard st (λ*s*. *P s* ∧ *vgetter s* (*x' s*)) (λ*s*. (*c-guard* (*x s* :: ('*a*::*xmem-type*) *ptr*)))
⟨*proof*⟩

lemma *heap-abs-expr-h-val* [*heap-abs*]:
⟦ *typ-heap-simulation st r w v t-hrs t-hrs-update*;
abs-expr st P x' x ⟧ ⟹
abs-expr st
(λ*s*. *P s* ∧ *v s* (*x' s*))
(λ*s*. (*r s* (*x' s*)))
(λ*s*. (*h-val* (*hrs-mem* (*t-hrs s*))) (*x s*))
⟨*proof*⟩

lemma *heap-abs-modifies-heap-update--unused*:
⟦ *typ-heap-simulation st r w v t-hrs t-hrs-update*;
abs-expr st Pb b' b;
abs-expr st Pc c' c ⟧ ⟹
abs-modifies st (λ*s*. *Pb s* ∧ *Pc s* ∧ *v s* (*b' s*))
(λ*s*. *w* (*b' s*) (λ*x*. (*c' s*)) *s*)
(λ*s*. *t-hrs-update* (*hrs-mem-update* (*heap-update* (*b s* :: ('*a*::*xmem-type*) *ptr*) (*c s*))) *s*)
⟨*proof*⟩

definition *heap-lift--h-val* ≡ *h-val*

lemma *heap-abs-modifies-heap-update* [*heap-abs*]:
⟦ *typ-heap-simulation st r w v t-hrs t-hrs-update*;
abs-expr st Pb b' b;
⋀*v*. *abs-expr st Pc* (*c' v*) (*c v*) ⟧ ⟹
abs-modifies st (λ*s*. *Pb s* ∧ *Pc s* ∧ *v s* (*b' s*))
(λ*s*. *w* (*b' s*) (λ-. (*c'* (*r s* (*b' s*)) *s*)) *s*)
(λ*s*. *t-hrs-update* (*hrs-mem-update*
(*heap-update* (*b s* :: ('*a*::*xmem-type*) *ptr*)
(*c* (*heap-lift--h-val* (*hrs-mem* (*t-hrs s*)) (*b s*)) *s*))) *s*)
⟨*proof*⟩

lemma *struct-rewrite-guard-expr* [*heap-abs*]:
struct-rewrite-expr P a' a ⟹ *struct-rewrite-guard* (λ*s*. *P s* ∧ *a' s*) *a*
⟨*proof*⟩

lemma *struct-rewrite-guard-constant* [*heap-abs*]:
struct-rewrite-guard (λ-. *P*) (λ-. *P*)
⟨*proof*⟩

lemma *struct-rewrite-guard-conj* [*heap-abs*]:
⟦ *struct-rewrite-guard b' b*; *struct-rewrite-guard a' a* ⟧ ⟹
struct-rewrite-guard (λ*s*. *a' s* ∧ *b' s*) (λ*s*. *a s* ∧ *b s*)
⟨*proof*⟩

lemma *struct-rewrite-guard-split* [*heap-abs*]:
⟦ ⋀*a b*. *struct-rewrite-guard* (*A a b*) (*C a b*) ⟧
⟹ *struct-rewrite-guard* (*case r of* (*a*, *b*) ⇒ *A a b*) (*case r of* (*a*, *b*) ⇒ *C a b*)
⟨*proof*⟩

lemma *struct-rewrite-guard-c-guard-field* [*heap-abs*]:
⟦ *valid-struct-field field-name* (*field-getter* :: (*'a* :: *xmem-type*) ⇒ (*'f* :: *xmem-type*)) *field-setter t-hrs t-hrs-update*;
struct-rewrite-expr P p' p;
struct-rewrite-guard Q (λ*s*. *c-guard* (*p' s*)) ⟧ ⟹
struct-rewrite-guard (λ*s*. *P s* ∧ *Q s*)
(λ*s*. *c-guard* (*Ptr* (*field-lvalue* (*p s* :: *'a ptr*) *field-name*) :: *'f ptr*))
⟨*proof*⟩

lemma *align-of-array*: *align-of TYPE*((*'a* :: *array-outer-max-size*)[*'b'* :: *array-max-count*]) = *align-of TYPE*(*'a*)
⟨*proof*⟩

lemma *c-guard-array*:
⟦ *0* ≤ *k*; *nat k* < *CARD*(*'b*); *c-guard* (*p* :: ((*'a*::*array-outer-max-size*)[*'b*::*array-max-count*]) *ptr*) ⟧
⟹ *c-guard* (*ptr-coerce p* +*p k* :: *'a ptr*)
⟨*proof*⟩

lemma *struct-rewrite-guard-c-guard-Array-field* [*heap-abs*]:
⟦ *valid-struct-field field-name* (*field-getter* :: (*'a* :: *xmem-type*) ⇒ (*'f*::*array-outer-max-size* [*'n*::*array-max-count*])) *field-setter t-hrs t-hrs-update*;
struct-rewrite-expr P p' p;
struct-rewrite-guard Q (λ*s*. *c-guard* (*p' s*)) ⟧ ⟹

struct-rewrite-guard ($\lambda s.\ P\ s \wedge Q\ s \wedge 0 \leq k \wedge nat\ k < CARD('n)$)
($\lambda s.$ *c-guard* (*ptr-coerce* (*Ptr* (*field-lvalue* ($p\ s :: 'a\ ptr$) *field-name*) :: (('f['n]) *ptr*)) $+_p\ k :: 'f\ ptr$))
⟨*proof*⟩

lemma *struct-rewrite-expr-id*:
struct-rewrite-expr (λ-. *True*) *A A*
⟨*proof*⟩

lemma *struct-rewrite-expr-fun-app2* [*heap-abs-fo*]:
⟦ *struct-rewrite-expr P f f'*;
struct-rewrite-expr Q g g' ⟧ ⟹
struct-rewrite-expr ($\lambda s.\ P\ s \wedge Q\ s$) ($\lambda s\ a.\ f\ s\ a\ (g\ s\ a)$) ($\lambda s\ a.\ f'\ s\ a\ \$\ g'\ s\ a$)
⟨*proof*⟩

lemma *struct-rewrite-expr-fun-app* [*heap-abs-fo*]:
⟦ *struct-rewrite-expr Y x x'*; *struct-rewrite-expr X f f'* ⟧ ⟹
struct-rewrite-expr ($\lambda s.\ X\ s \wedge Y\ s$) ($\lambda s.\ f\ s\ (x\ s)$) ($\lambda s.\ f'\ s\ \$\ x'\ s$)
⟨*proof*⟩

lemma *struct-rewrite-expr-constant* [*heap-abs*]:
struct-rewrite-expr (λ-. *True*) (λ-. *a*) (λ-. *a*)
⟨*proof*⟩

lemma *struct-rewrite-expr-lambda-null* [*heap-abs*]:
struct-rewrite-expr P A C ⟹ *struct-rewrite-expr P* (λs -. *A s*) (λs -. *C s*)
⟨*proof*⟩

lemma *struct-rewrite-expr-split* [*heap-abs*]:
⟦ $\bigwedge a\ b.$ *struct-rewrite-expr* (*P a b*) (*A a b*) (*C a b*) ⟧
⟹ *struct-rewrite-expr* (*case r of* (*a*, *b*) ⇒ *P a b*)
(*case r of* (*a*, *b*) ⇒ *A a b*) (*case r of* (*a*, *b*) ⇒ *C a b*)
⟨*proof*⟩

lemma *struct-rewrite-expr-basecase-h-val* [*heap-abs*]:
struct-rewrite-expr (λ-. *True*) ($\lambda s.$ *h-val* (*h s*) (*p s*)) ($\lambda s.$ *h-val* (*h s*) (*p s*))
⟨*proof*⟩

lemma *struct-rewrite-expr-indirect-h-val* [*heap-abs*]:
struct-rewrite-expr P a c ⟹
struct-rewrite-expr P ($\lambda s.$ *h-val* (*h s*) (*a s*)) ($\lambda s.$ *h-val* (*h s*) (*c s*))
⟨*proof*⟩

lemma *struct-rewrite-expr-field* [*heap-abs*]:

⟦ *valid-struct-field field-name* (*field-getter* :: (′*a* :: *xmem-type*) ⇒ (′*f* :: *xmem-type*)) *field-setter t-hrs t-hrs-update*;
struct-rewrite-expr P p′ p;
struct-rewrite-expr Q a (λ*s*. *h-val* (*hrs-mem* (*t-hrs s*)) (*p′ s*)) ⟧
⟹ *struct-rewrite-expr* (λ*s*. *P s* ∧ *Q s*) (λ*s*. *field-getter* (*a s*))
(λ*s*. *h-val* (*hrs-mem* (*t-hrs s*)) (*Ptr* (*field-lvalue* (*p s*) *field-name*)))
⟨*proof*⟩

lemma *abs-expr-field* [*heap-abs*]:
⟦ *valid-struct-field field-name* (*field-getter* :: (′*a* :: *xmem-type*) ⇒ (′*f* :: *xmem-type*)) *field-setter t-hrs t-hrs-update*;
abs-expr st P a c⟧
⟹ *abs-expr st P* (λ*s*. *field-getter* (*a s*)) (λ*s*. *field-getter* (*c s*))
⟨*proof*⟩

lemma *struct-rewrite-expr-Array-field* [*heap-abs*]:
⟦ *valid-struct-field field-name*
(*field-getter* :: (′*a* :: *xmem-type*) ⇒ ′*f*::*array-outer-max-size* [′*n*::*array-max-count*])
field-setter t-hrs t-hrs-update;
struct-rewrite-expr P p′ p;
struct-rewrite-expr Q a (λ*s*. *h-val* (*hrs-mem* (*t-hrs s*)) (*p′ s*)) ⟧
⟹ *struct-rewrite-expr* (λ*s*. *P s* ∧ *Q s* ∧ *k* ≥ *0* ∧ *nat k* < *CARD*(′*n*))
(λ*s*. *index* (*field-getter* (*a s*)) (*nat k*))
(λ*s*. *h-val* (*hrs-mem* (*t-hrs s*))
(*ptr-coerce* (*Ptr* (*field-lvalue* (*p s*) *field-name*) :: (′*f*[′*n*]) *ptr*) +$_p$ *k*))
⟨*proof*⟩
declare *struct-rewrite-expr-Array-field* [*unfolded ptr-coerce.simps*, *heap-abs*]

lemma *struct-rewrite-modifies-id* [*heap-abs*]:
struct-rewrite-modifies (λ*-*. *True*) *A A*
⟨*proof*⟩

lemma *struct-rewrite-modifies-basecase* [*heap-abs*]:
⟦ *typ-heap-simulation st* (*getter* :: ′*s* ⇒ ′*a ptr* ⇒ (′*a*::*xmem-type*)) *setter vgetter t-hrs t-hrs-update*;
struct-rewrite-expr P p′ p;
struct-rewrite-expr Q v′ v ⟧ ⟹
struct-rewrite-modifies (λ*s*. *P s* ∧ *Q s*)
(λ*s*. *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p′ s*) (*v′ s* :: ′*a*))) *s*)
(λ*s*. *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p s*) (*v s* :: ′*a*))) *s*)
⟨*proof*⟩

lemma *heap-update-field-unpacked*:

⟦ *field-ti TYPE('a::mem-type) f = Some (t :: 'a xtyp-info)*;
c-guard (p :: 'a::mem-type ptr);
export-uinfo t = export-uinfo (typ-info-t TYPE('b::mem-type)) ⟧ ⟹
heap-update (Ptr &(p→f) :: 'b ptr) v hp =
heap-update p (update-ti t (to-bytes-p v) (h-val hp p)) hp
⟨*proof*⟩

lemma *heap-update-Array-element-unpacked*:
n < CARD('b::array-max-count) ⟹
heap-update (ptr-coerce p' +$_p$ int n) w hp =
heap-update (p'::('a::array-outer-max-size['b::array-max-count]) ptr)
(Arrays.update (h-val hp p') n w) hp
⟨*proof*⟩

lemma *read-write-valid-hrs-mem*:
lense hrs-mem hrs-mem-update
⟨*proof*⟩

definition *heap-lift--wrap-h-val* ≡ (=)

lemma *heap-lift-wrap-h-val* [*heap-abs*]:
heap-lift--wrap-h-val (heap-lift--h-val s p) (h-val s p)
⟨*proof*⟩

lemma *heap-lift-wrap-h-val-skip* [*heap-abs*]:
heap-lift--wrap-h-val (h-val s (Ptr (field-lvalue p f))) (h-val s (Ptr (field-lvalue p f)))
⟨*proof*⟩

lemma *heap-lift-wrap-h-val-skip-array* [*heap-abs*]:
heap-lift--wrap-h-val (h-val s (ptr-coerce p +$_p$ k))
(h-val s (ptr-coerce p +$_p$ k))
⟨*proof*⟩

lemma *struct-rewrite-modifies-field--unused*:
⟦ *valid-struct-field field-name (field-getter :: ('a::xmem-type) ⇒ ('f::xmem-type)) field-setter t-hrs t-hrs-update*;
struct-rewrite-expr P p' p;
struct-rewrite-expr Q f' f;
struct-rewrite-modifies R
(λs. t-hrs-update (hrs-mem-update (heap-update (p'' s)
(u s (field-setter (λ-. f' s))))) s)

(λs. t-hrs-update (hrs-mem-update (heap-update (p′ s)
(field-setter (λ-. f′ s) (h-val (hrs-mem (t-hrs s)) (p′ s))))) s);
struct-rewrite-guard S (λs. c-guard (p′ s)) ⟧ ⟹
struct-rewrite-modifies (λs. P s ∧ Q s ∧ R s ∧ S s)
(λs. t-hrs-update (hrs-mem-update (heap-update (p″ s)
(u s (field-setter (λ-. f′ s))))) s)
(λs. t-hrs-update (hrs-mem-update (heap-update (Ptr (field-lvalue (p s) field-name))
(f s))) s)
⟨proof⟩

lemma *struct-rewrite-modifies-Array-field--unused*:
⟦ valid-struct-field field-name (field-getter :: ('a::xmem-type) ⇒ (('f::array-outer-max-size)['n::array-max-coun
field-setter t-hrs t-hrs-update;
struct-rewrite-expr P p′ p;
struct-rewrite-expr Q f′ f;
struct-rewrite-modifies R
(λs. t-hrs-update (hrs-mem-update (heap-update (p″ s)
(u s (field-setter (λa. Arrays.update a (nat k) (f′ s)))))) s)
(λs. t-hrs-update (hrs-mem-update (heap-update (p′ s)
(field-setter (λa. Arrays.update a (nat k) (f′ s))
(h-val (hrs-mem (t-hrs s)) (p′ s))))) s);
struct-rewrite-guard S (λs. c-guard (p′ s)) ⟧ ⟹
struct-rewrite-modifies (λs. P s ∧ Q s ∧ R s ∧ S s ∧ 0 ≤ k ∧ nat k < CARD('n))
(λs. t-hrs-update (hrs-mem-update (heap-update (p″ s)
(u s (field-setter (λa. Arrays.update a (nat k) (f′ s)))))) s)
(λs. t-hrs-update (hrs-mem-update (heap-update
(ptr-coerce (Ptr (field-lvalue (p s) field-name) :: ('f['n]) ptr) +$_p$ k) (f s)))
s)
⟨proof⟩

lemma *struct-rewrite-modifies-field* [*heap-abs*]:
⟦ valid-struct-field field-name (field-getter :: ('a::xmem-type) ⇒ ('f::xmem-type))
field-setter t-hrs t-hrs-update;
struct-rewrite-expr P p′ p;
struct-rewrite-expr Q f′ f;
⋀s. heap-lift--wrap-h-val (h-val-p′ s) (h-val (hrs-mem (t-hrs s)) (p′ s));
struct-rewrite-modifies R
(λs. t-hrs-update (hrs-mem-update (heap-update (p″ s)
(u s (field-setter (f′ s))))) s)
(λs. t-hrs-update (hrs-mem-update (heap-update (p′ s)
(field-setter (f′ s) (h-val-p′ s)))) s);
struct-rewrite-guard S (λs. c-guard (p′ s)) ⟧ ⟹
struct-rewrite-modifies (λs. P s ∧ Q s ∧ R s ∧ S s)
(λs. t-hrs-update (hrs-mem-update (heap-update (p″ s)
(u s (field-setter (f′ s))))) s)
(λs. t-hrs-update (hrs-mem-update (heap-update (Ptr (field-lvalue (p s) field-name))
(f s (h-val (hrs-mem (t-hrs s)) (Ptr (field-lvalue (p s) field-name))))))

s)
⟨*proof*⟩

lemma *struct-rewrite-modifies-Array-field* [*heap-abs*]:
⟦ *valid-struct-field field-name* (*field-getter* :: ('*a*::*xmem-type*) ⇒ (('*f*::*array-outer-max-size*)['*n*::*array-max-coun*
field-setter t-hrs t-hrs-update;
struct-rewrite-expr P p' p;
struct-rewrite-expr Q f' f;
⋀*s*. *heap-lift--wrap-h-val* (*h-val-p'* *s*) (*h-val* (*hrs-mem* (*t-hrs s*)) (*p'* *s*));
struct-rewrite-modifies R
(λ*s*. *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p''* *s*)
(*u s* (*field-setter* (λ*a*. *Arrays.update a* (*nat k*) (*f'* *s* (*index a* (*nat
k)))))))) *s*)
(λ*s*. *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p'* *s*)
(*field-setter* (λ*a*. *Arrays.update a* (*nat k*) (*f'* *s* (*index a* (*nat k*))))
(*h-val-p'* *s*)))) *s*);
struct-rewrite-guard S (λ*s*. *c-guard* (*p'* *s*)) ⟧ ⟹
struct-rewrite-modifies (λ*s*. *P s* ∧ *Q s* ∧ *R s* ∧ *S s* ∧ *0* ≤ *k* ∧ *nat k* < *CARD*('*n*))
(λ*s*. *t-hrs-update* (*hrs-mem-update* (*heap-update* (*p''* *s*)
(*u s* (*field-setter* (λ*a*. *Arrays.update a* (*nat k*) (*f'* *s* (*index a* (*nat k*))))))))
s)
(λ*s*. *t-hrs-update* (*hrs-mem-update* (*heap-update*
(*ptr-coerce* (*Ptr* (*field-lvalue* (*p s*) *field-name*) :: ('*f*['*n*]) *ptr*) +$_p$ *k*)
(*f s* (*h-val* (*hrs-mem* (*t-hrs s*)) (*ptr-coerce* (*Ptr* (*field-lvalue* (*p s*)
field-name) :: ('*f*['*n*]) *ptr*) +$_p$ *k* :: '*f ptr*))))) *s*)
⟨*proof*⟩

definition
valid-globals-field ::
('*s* ⇒ '*t*)
⇒ ('*s* ⇒ '*a*)
⇒ (('*a* ⇒ '*a*) ⇒ '*s* ⇒ '*s*)
⇒ ('*t* ⇒ '*a*)
⇒ (('*a* ⇒ '*a*) ⇒ '*t* ⇒ '*t*)
⇒ *bool*
where
valid-globals-field st old-getter old-setter new-getter new-setter ≡
(∀ *s*. *new-getter* (*st s*) = *old-getter s*)
∧ (∀ *s v*. *new-setter v* (*st s*) = *st* (*old-setter v s*))

lemma *abs-expr-globals-getter* [*heap-abs*]:
⟦ *valid-globals-field st old-getter old-setter new-getter new-setter* ⟧
⟹ *abs-expr st* (λ*-*. *True*) *new-getter old-getter*
⟨*proof*⟩

lemma *abs-expr-globals-setter* [*heap-abs*]:
⟦ *valid-globals-field st old-getter old-setter new-getter new-setter*;
⋀*old*. *abs-expr st* (*P old*) (*v old*) (*v′ old*) ⟧
⟹ *abs-modifies st* (λ*s*. ∀ *old*. *P old s*) (λ*s*. *new-setter* (λ*old*. *v old s*) *s*) (λ*s*. *old-setter* (λ*old*. *v′ old s*) *s*)
⟨*proof*⟩

lemma *struct-rewrite-expr-globals-getter* [*heap-abs*]:
⟦ *valid-globals-field st old-getter old-setter new-getter new-setter* ⟧
⟹ *struct-rewrite-expr* (λ-. *True*) *old-getter old-getter*
⟨*proof*⟩

lemma *struct-rewrite-modifies-globals-setter* [*heap-abs*]:
⟦ *valid-globals-field st old-getter old-setter new-getter new-setter*;
⋀*old*. *struct-rewrite-expr* (*P old*) (*v′ old*) (*v old*) ⟧
⟹ *struct-rewrite-modifies* (λ*s*. ∀ *old*. *P old s*) (λ*s*. *old-setter* (λ*old*. *v′ old s*) *s*) (λ*s*. *old-setter* (λ*old*. *v old s*) *s*)
⟨*proof*⟩

lemma *abs-spec-may-not-modify-globals*[*heap-abs*]:
abs-spec st (λ-. *True*) {(*a*, *b*). *meq b a*} {(*a*, *b*). *meq b a*}
⟨*proof*⟩

lemma *abs-spec-modify-global*[*heap-abs*]:
valid-globals-field st old-getter old-setter new-getter new-setter ⟹
abs-spec st (λ-. *True*) {(*a*, *b*). *C a b*} {(*a*, *b*). *C′ a b*} ⟹
abs-spec st (λ-. *True*) {(*a*, *b*). *mex* (λ*x*. *C* (*new-setter* (λ-. *x*) *a*) *b*)} {(*a*, *b*). *mex* (λ*x*. *C′* (*old-setter* (λ-. *x*) *a*) *b*)}
⟨*proof*⟩

lemma *uint-scast*:
uint (*scast x* :: *′a word*) = *uint* (*x* :: *′a*::*len signed word*)
⟨*proof*⟩

lemma *to-bytes-signed-word*:
to-bytes (*x* :: *′a*::*len8 signed word*) *p* = *to-bytes* (*scast x* :: *′a word*) *p*
⟨*proof*⟩

lemma *from-bytes-signed-word*:
length p = *len-of TYPE*(*′a*) *div* 8 ⟹
(*from-bytes p* :: *′a*::*len8 signed word*) = *ucast* (*from-bytes p* :: *′a word*)
⟨*proof*⟩

lemma *hrs-mem-update-signed-word*:

hrs-mem-update (*heap-update* (*ptr-coerce p* :: *'a*::*len8 word ptr*) (*scast val* :: *'a*::*len8 word*))
= *hrs-mem-update* (*heap-update p* (*val* :: *'a*::*len8 signed word*))
⟨*proof*⟩

lemma *h-val-signed-word*:
(*h-val a p* :: *'a*::*len8 signed word*) = *ucast* (*h-val a* (*ptr-coerce p* :: *'a word ptr*))
⟨*proof*⟩

lemma *align-of-signed-word*:
align-of TYPE(*'a*::*len8 signed word*) = *align-of TYPE*(*'a word*)
⟨*proof*⟩

lemma *size-of-signed-word*:
size-of TYPE(*'a*::*len8 signed word*) = *size-of TYPE*(*'a word*)
⟨*proof*⟩

lemma *c-guard-ptr-coerce*:
⟦ *align-of TYPE*(*'a*) = *align-of TYPE*(*'b*);
size-of TYPE(*'a*) = *size-of TYPE*(*'b*) ⟧ ⟹
c-guard (*ptr-coerce p* :: (*'b*::*c-type*) *ptr*) = *c-guard* (*p* :: (*'a*::*c-type*) *ptr*)
⟨*proof*⟩

lemma *word-rsplit-signed*:
(*word-rsplit* (*ucast v'* :: (*'a*::*len*) *signed word*) :: *8 word list*) = *word-rsplit* (*v'* :: *'a word*)
⟨*proof*⟩

lemma *heap-update-signed-word*:
heap-update (*ptr-coerce p* :: *'a word ptr*) (*scast v*) = *heap-update* (*p* :: (*'a*::*len8*) *signed word ptr*) *v*
heap-update (*ptr-coerce p'* :: *'a signed word ptr*) (*ucast v'*) = *heap-update* (*p'* :: (*'a*::*len8*) *word ptr*) *v'*
⟨*proof*⟩

lemma *heap-update-padding-signed-word*:
heap-update-padding (*ptr-coerce p* :: *'a word ptr*) (*scast v*) *bs* = *heap-update-padding* (*p* :: (*'a*::*len8*) *signed word ptr*) *v bs*
heap-update-padding (*ptr-coerce p'* :: *'a signed word ptr*) (*ucast v'*) *bs* = *heap-update-padding* (*p'* :: (*'a*::*len8*) *word ptr*) *v' bs*
⟨*proof*⟩

lemma *typ-heap-simulation-c-guard*:
⟦ *typ-heap-simulation st r w v t-hrs t-hrs-update*;
v (*st s*) *p* ⟧ ⟹ *c-guard p*
⟨*proof*⟩

abbreviation (*input*)

scast-f :: ((*'a*::*len*) *signed word ptr* ⇒ *'a signed word*)
⇒ (*'a word ptr* ⇒ *'a word*)
where
scast-f f ≡ (λ*p*. *scast* (*f* (*ptr-coerce p*)))

abbreviation (*input*)
ucast-f :: ((*'a*::*len*) *word ptr* ⇒ *'a word*)
⇒ (*'a signed word ptr* ⇒ *'a signed word*)
where
ucast-f f ≡ (λ*p*. *ucast* (*f* (*ptr-coerce p*)))

abbreviation (*input*)
cast-f' :: (*'a ptr* ⇒ *'x*) ⇒ (*'b ptr* ⇒ *'x*)
where
cast-f' f ≡ (λ*p*. *f* (*ptr-coerce p*))

lemma *read-write-validE-weak*:
⟦ *lense r w*;
⟦ ⋀*f s*. *r* (*w f s*) = *f* (*r s*);
⋀*f s*. *f* (*r s*) = (*r s*) ⟹ *w f s* = *s* ⟧ ⟹ *R* ⟧
⟹ *R*
⟨*proof*⟩

lemma *lense-transcode*:
⟦ *lense r w*; ⋀*v*. *f'* (*f v*) = *v*; ⋀*v*. *f* (*f' v*) = *v* ⟧ ⟹
lense (λ*s*. *f'* (*r s*)) (λ*g s*. *w* (λ*old*. *f* (*g* (*f' old*))) *s*)
⟨*proof*⟩

lemma *typ-heap-simulation-signed-word*:
notes *align-of-signed-word* [*simp*]
size-of-signed-word [*simp*]
heap-update-signed-word [*simp*]
shows
⟦ *typ-heap-simulation st*
(*r* :: *'s* ⇒ (*'a*::*len8*) *word ptr* ⇒ *'a word*) *w*
v t-hrs t-hrs-update ⟧
⟹ *typ-heap-simulation st*
(λ*s p*. *ucast* (*r s* (*ptr-coerce p*)) :: *'a signed word*)
(λ*p f*. (*w* (*ptr-coerce p*) ((λ*x*. *scast* (*f* (*ucast x*))))))
(λ*s p*. *v s* (*ptr-coerce p*))
t-hrs t-hrs-update
⟨*proof*⟩

lemma *c-guard-ptr-ptr-coerce*:
⟦ *c-guard* (*a* :: (*'a*::*c-type*) *ptr ptr*); *ptr-val a* = *ptr-val b* ⟧ ⟹
c-guard (*b* :: (*'b*::*c-type*) *ptr ptr*)
⟨*proof*⟩

abbreviation (*input*)
ptr-coerce-f :: (*'a ptr ptr* ⇒ *'a ptr*) ⇒ (*'b ptr ptr* ⇒ *'b ptr*)
where
ptr-coerce-f f ≡ (λ*p*. *ptr-coerce* (*f* (*ptr-coerce p*)))

abbreviation (*input*)
ptr-coerce-range-f :: (*'a ptr* ⇒ *bool*) ⇒ (*'b ptr* ⇒ *bool*)
where
ptr-coerce-range-f f ≡ (λ*p*. (*f* (*ptr-coerce p*)))

lemma *typ-heap-simulation-ptr-coerce*:
⟦ *typ-heap-simulation st*
(*r* :: *'s* ⇒ (*'a*::*c-type*) *ptr ptr* ⇒ *'a ptr*) *w*
v t-hrs t-hrs-update ⟧
⟹ *typ-heap-simulation st*
(λ*s p*. *ptr-coerce* (*r s* (*ptr-coerce p*)) :: (*'b*::*c-type*) *ptr*)
(λ*p f*. (*w* (*ptr-coerce p*) ((λ*x*. *ptr-coerce* (*f* (*ptr-coerce x*))))))
(λ*s p*. *v s* (*ptr-coerce p*))
t-hrs t-hrs-update
⟨*proof*⟩

lemmas *signed-typ-heap-simulations* =
typ-heap-simulation-signed-word
typ-heap-simulation-ptr-coerce [**where** *'a*=(*'x*::*len8*) *word* **and** *'b*=(*'x*::*len8*) *signed word*]
typ-heap-simulation-ptr-coerce [**where** *'a*=(*'x*::*len8*) *word ptr* **and** *'b*=(*'x*::*len8*) *signed word ptr*]
typ-heap-simulation-ptr-coerce [**where** *'a*=(*'x*::*len8*) *word ptr ptr* **and** *'b*=(*'x*::*len8*) *signed word ptr ptr*]
typ-heap-simulation-ptr-coerce [**where** *'a*=(*'x*::*len8*) *word ptr ptr ptr* **and** *'b*=(*'x*::*len8*) *signed word ptr ptr ptr*]

lemma *ptr-coerce-eq*:
(*ptr-coerce x* = *ptr-coerce y*) = (*x* = *y*)
⟨*proof*⟩

lemma *signed-word-heap-opt* [*L2opt*]:
(*scast* (((λ*x*. *ucast* (*a* (*ptr-coerce x*))) (*p* := *v* :: *'a*::*len signed word*)) (*b* :: *'a signed word ptr*)))
= ((*a*(*ptr-coerce p* := (*scast v* :: *'a word*))) ((*ptr-coerce b*) :: *'a word ptr*))
⟨*proof*⟩

lemma *signed-word-heap-ptr-coerce-opt* [*L2opt*]:
(*ptr-coerce* (((λ*x*. *ptr-coerce* (*a* (*ptr-coerce x*))) (*p* := *v* :: *'a ptr*)) (*b* :: *'a ptr ptr*)))
= ((*a*(*ptr-coerce p* := (*ptr-coerce v* :: *'b ptr*))) ((*ptr-coerce b*) :: *'b ptr ptr*))
⟨*proof*⟩

declare *ptr-coerce-idem* [*L2opt*]
declare *scast-ucast-id* [*L2opt*]
declare *ucast-scast-id* [*L2opt*]

lemma *heap-abs-expr-c-guard-array* [*heap-abs*]:
⟦ *typ-heap-simulation st r w v t-hrs t-hrs-update*;
abs-expr st P x' (λ*s*. *ptr-coerce* (*x s*) :: *'a ptr*) ⟧ ⟹
abs-guard st
(λ*s*. *P s* ∧ (∀ *a* ∈ *set* (*array-addrs* (*x' s*) *CARD*(*'b*)). *v s a*))
(λ*s*. *c-guard* (*x s* :: (*'a*::*array-outer-max-size*, *'b*::*array-max-count*) *array ptr*))
⟨*proof*⟩

lemma *fold-over-st*:
⟦ *xs* = *ys*; *P s*;
⋀*s x*. *x* ∈ *set xs* ∧ *P s* ⟹ *P* (*g x s*) ∧ *f x* (*st s*) = *st* (*g x s*)
⟧ ⟹ *fold f xs* (*st s*) = *st* (*fold g ys s*)
⟨*proof*⟩

lemma *fold-lift-write*:
⟦ *xs* = *ys*; *lense r w*
⟧ ⟹ *fold* (λ*i*. *w* (*f i*)) *xs s* = *w* (*fold f ys*) *s*
⟨*proof*⟩

lemma *fold-heap-update-list-nmem-same*:
⟦ *n* ∗ *size-of TYPE*(*'a* :: *mem-type*) < *addr-card*;
n ∗ *size-of TYPE*(*'a*) ≤ *k*; *k* < *addr-card*;
⋀*i h*. *length* (*pad i h*) = *size-of TYPE*(*'a*) ⟧ ⟹
h (*ptr-val* (*p* :: *'a ptr*) + *of-nat k*) =
(*fold* (λ*i h*. *heap-update-list* (*ptr-val* (*p* +$_p$ *int i*))
(*to-bytes* (*val i h* :: *'a*) (*pad i h*)) *h*) [*0*..<*n*] *h*) (*ptr-val p* + *of-nat k*)
⟨*proof*⟩

lemma *heap-list-of-disjoint-fold-heap-update-list*:
⟦ *n* ∗ *size-of TYPE*(*'a* :: *mem-type*) < *addr-card*;
n ∗ *size-of TYPE*(*'a*) + *k* < *addr-card*;
⋀*i h*. *length* (*pad i h*) = *size-of TYPE*(*'a*) ⟧ ⟹
heap-list (*fold* (λ*i h*. *heap-update-list* (*ptr-val* ((*p* :: *'a ptr*) +$_p$ *int i*))
(*to-bytes* (*val i h* :: *'a*) (*pad i h*)) *h*) [*0*..<*n*] *h*)
k (*ptr-val* (*p* +$_p$ *int n*))
= *heap-list h k* (*ptr-val* (*p* +$_p$ *int n*))
⟨*proof*⟩

lemma *fold-heap-update-list*:
$n * size\text{-}of\ TYPE('a :: mem\text{-}type) < 2\hat{}addr\text{-}bitsize \Longrightarrow$
$fold\ (\lambda i\ h.\ heap\text{-}update\text{-}list\ (ptr\text{-}val\ ((p :: 'a\ ptr) +_p int\ i))$
$(to\text{-}bytes\ (val\ i :: 'a)$
$(heap\text{-}list\ h\ (size\text{-}of\ TYPE('a))\ (ptr\text{-}val\ (p +_p int\ i))))\ h)$
$[0..<n]\ h =$
$fold\ (\lambda i.\ heap\text{-}update\text{-}list\ (ptr\text{-}val\ (p +_p int\ i))$
$(to\text{-}bytes\ (val\ i)$
$(heap\text{-}list\ h\ (size\text{-}of\ TYPE('a))\ (ptr\text{-}val\ (p +_p int\ i)))))$
$[0..<n]\ h$
⟨*proof*⟩

lemma *access-ti-list-array-unpacked*:
$[\![\forall n.\ size\text{-}td\text{-}tuple\ (f\ n) = v3;\ length\ xs = v3 * n;$
$\forall m\ xs.\ length\ xs = v3 \wedge m < n \longrightarrow$
$access\text{-}ti\text{-}tuple\ (f\ m)\ (FCP\ g)\ xs = h\ m\ xs$
$]\!] \Longrightarrow$
$access\text{-}ti\text{-}list\ (map\ f\ [0\ ..<\ n])\ (FCP\ g)\ xs$
$= foldl\ (@)\ []\ (map\ (\lambda m.\ h\ m\ (take\ v3\ (drop\ (v3 * m)\ xs)))\ [0\ ..<\ n])$
⟨*proof*⟩

lemma *concat-nth-chunk*:
$[\![\forall x \in set\ xs.\ length\ (f\ x) = chunk;\ n < length\ xs\ ]\!]$
$\Longrightarrow take\ chunk\ (drop\ (n * chunk)\ (concat\ (map\ f\ xs))) = f\ (xs\ !\ n)$
⟨*proof*⟩

lemma *array-update-split*:
$[\![typ\text{-}heap\text{-}simulation\ st\ (r :: 's \Rightarrow ('a::array\text{-}outer\text{-}max\text{-}size)\ ptr \Rightarrow 'a)\ w$
$v\ t\text{-}hrs\ t\text{-}hrs\text{-}update;$
$\forall x \in set\ (array\text{-}addrs\ (ptr\text{-}coerce\ p)\ CARD('b::array\text{-}max\text{-}count)).$
$v\ (st\ s)\ x$
$]\!] \Longrightarrow st\ (t\text{-}hrs\text{-}update\ (hrs\text{-}mem\text{-}update\ (heap\text{-}update\ p\ (arr :: 'a['b]))))\ s) =$
$fold\ (\lambda i.\ w\ (ptr\text{-}coerce\ p +_p int\ i)\ (\lambda x.\ index\ arr\ i))$
$[0\ ..<\ CARD('b)]\ (st\ s)$
⟨*proof*⟩

lemma *fold-update-id*:
$[\![lense\ getter\ setter;$
$\forall i \in set\ xs.\ \forall j \in set\ xs.\ (i = j) = (ind\ i = ind\ j);$
$\forall i \in set\ xs.\ val\ i = getter\ s\ (ind\ i)$
$]\!] \Longrightarrow fold\ (\lambda i.\ setter\ (\lambda x.\ x(ind\ i := val\ i)))\ xs\ s = s$
⟨*proof*⟩

lemma *fold-update-id'*:
$[\![pointer\text{-}lense\ r\ w;$

$\forall i \in set\ xs.\ \forall j \in set\ xs.\ (i = j) = (ind\ i = ind\ j)$;
$\forall i \in set\ xs.\ val\ i = r\ s\ (ind\ i)$
$\rrbracket \Longrightarrow fold\ (\lambda i.\ w\ (ind\ i)\ (\lambda\text{-}.\ val\ i))\ xs\ s = s$
⟨*proof*⟩

lemma *array-count-index*:
$\llbracket\ i < CARD('b::array\text{-}max\text{-}count);\ j < CARD('b)\ \rrbracket$
$\Longrightarrow (i = j) =$
$((of\text{-}nat\ (i * size\text{-}of\ TYPE('a::array\text{-}outer\text{-}max\text{-}size)) :: addr)$
$= of\text{-}nat\ (j * size\text{-}of\ TYPE('a)))$
⟨*proof*⟩

lemma *le-outside-intvl*: $p < a \Longrightarrow 0 \notin \{a\ ..+b\} \Longrightarrow p \notin \{a\ ..+b\}$
⟨*proof*⟩

lemma *intvl-mult-split*:
$\{p\ ..+\ a * b\} = (\bigcup i{<}b.\ \{p + of\text{-}nat\ (i * a)\ ..+\ a\})$
⟨*proof*⟩

lemma *intvl-mul-disjnt*:
fixes $n\ i :: 'a::len\ word$
assumes *n*: $unat\ n < b$ **and** *i*: $unat\ i < b$ **and** *b*: $sz * b \leq 2\hat{}LENGTH('a)$
assumes *ni*: $n \neq i$
shows $\{n * word\text{-}of\text{-}nat\ sz..+sz\} \cap \{i * word\text{-}of\text{-}nat\ sz..+sz\} = \{\}$
⟨*proof*⟩

lemma *array-disj-helper*:
fixes $p :: ('a::mem\text{-}type['b])\ ptr$
assumes *ni*: $n < CARD('b)\ i < CARD('b)\ n \neq i$
assumes *valid*: $\forall x{\in}set\ (array\text{-}addrs\ (PTR\text{-}COERCE('a['b] \to 'a)\ p)\ CARD('b)).$
c-guard x
shows $\{ptr\text{-}val\ p + word\text{-}of\text{-}nat\ n * word\text{-}of\text{-}nat\ (size\text{-}of\ TYPE('a))..+size\text{-}of$
$TYPE('a)\} \cap$
$\{ptr\text{-}val\ p + word\text{-}of\text{-}nat\ i * word\text{-}of\text{-}nat\ (size\text{-}of\ TYPE('a))..+size\text{-}of\ TYPE('a)\}$
$= \{\}$
⟨*proof*⟩

theorem *heap-abs-array-update* [*heap-abs*]:
$\llbracket$ *typ-heap-simulation st* $(r :: 's \Rightarrow 'a\ ptr \Rightarrow 'a)\ w$
v t-hrs t-hrs-update;
abs-expr st Pb b' b;
abs-expr st Pn n' n;
abs-expr st Pv y' y
$\rrbracket \Longrightarrow$

abs-modifies st ($\lambda s.\ Pb\ s \wedge Pn\ s \wedge Pv\ s \wedge n'\ s < CARD('b) \wedge$
$(\forall\, ptr \in set\ (array\text{-}addrs\ (ptr\text{-}coerce\ (b'\ s))\ CARD('b)).\ (v\ s\ ptr)))$
($\lambda s.\ w\ (ptr\text{-}coerce\ (b'\ s) +_p int\ (n'\ s))\ (\lambda v.\ y'\ s)\ \ s$)
($\lambda s.\ t\text{-}hrs\text{-}update\ (hrs\text{-}mem\text{-}update\ ($
heap-update ($b\ s$) (*Arrays.update* ((*h-val* (*hrs-mem* (*t-hrs* s)) ($b\ s$))
:: ('*a*::*array-outer-max-size*)['*b*::*array-max-count*]) ($n\ s$) ($y\ s$)))) s)
⟨*proof*⟩

lemma *heap-abs-array-access*[*heap-abs*]:
⟦ *typ-heap-simulation st* (r :: '$s \Rightarrow$ ('*a*::*xmem-type*) *ptr* $\Rightarrow$ 'a) w
v *t-hrs t-hrs-update*;
abs-expr st Pb b' b;
abs-expr st Pn n' n
⟧ $\Longrightarrow$
abs-expr st ($\lambda s.\ Pb\ s \wedge Pn\ s \wedge n'\ s < CARD('b::finite) \wedge v\ s\ (ptr\text{-}coerce\ (b'\ s) +_p int\ (n'\ s))$)
($\lambda s.\ r\ s\ (ptr\text{-}coerce\ (b'\ s) +_p int\ (n'\ s))$)
($\lambda s.\ index\ (h\text{-}val\ (hrs\text{-}mem\ (t\text{-}hrs\ s))\ (b\ s :: ('a['b])\ ptr))\ (n\ s)$)
⟨*proof*⟩

lemma *the-fun-upd-lemma1*:
($\lambda x.\ the\ (f\ x))(p := v) = (\lambda x.\ the\ ((f\ (p := Some\ v))\ x))$
⟨*proof*⟩

lemma *the-fun-upd-lemma2*:
$\exists\, z.\ f\ p = Some\ z \Longrightarrow$
$(\lambda x.\ \exists\, z.\ (f\ (p := Some\ v))\ x = Some\ z) = (\lambda x.\ \exists\, z.\ f\ x = Some\ z)$
⟨*proof*⟩

lemma *the-fun-upd-lemma3*:
$((\lambda x.\ the\ (f\ x))(p := v))\ x = the\ ((f\ (p := Some\ v))\ x)$
⟨*proof*⟩

lemma *the-fun-upd-lemma4*:
$\exists\, z.\ f\ p = Some\ z \Longrightarrow$
$(\exists\, z.\ (f\ (p := Some\ v))\ x = Some\ z) = (\exists\, z.\ f\ x = Some\ z)$
⟨*proof*⟩

lemmas *the-fun-upd-lemmas* =
the-fun-upd-lemma1
the-fun-upd-lemma2
the-fun-upd-lemma3
the-fun-upd-lemma4

lemma *sword-update*:
$\bigwedge ptr :: ('a :: len)\ signed\ word\ ptr.$
$(\lambda(x :: 'a\ word\ ptr \Rightarrow 'a\ word)\ p :: 'a\ word\ ptr.$
if ptr-coerce p = ptr then scast $(n :: 'a\ signed\ word)$ *else x* $(ptr\text{-}coerce\ p))$
$=$
$(\lambda(old :: 'a\ word\ ptr \Rightarrow 'a\ word)\ x :: 'a\ word\ ptr.$
if x = ptr-coerce ptr then scast n else old x)
⟨*proof*⟩

Proof taken from $[\![?n < CARD(?'b);\ CARD(?'b) * size\text{-}of\ TYPE(?'a) < 2^{addr\text{-}bitsize}]\!] \Longrightarrow heap\text{-}update\ ?p\ (Arrays.update\ ?arr\ ?n\ ?v)\ ?hp = heap\text{-}update\ (array\text{-}ptr\text{-}index\ ?p\ False\ ?n)\ ?v\ (heap\text{-}update\ ?p\ ?arr\ ?hp)$

lemma (**in** *array-outer-max-size*) *array-index-unat-conv*:
assumes *x-bound*: $x < CARD('b :: array\text{-}max\text{-}count)$
assumes *k-bound*: $k < size\text{-}of\ TYPE('a)$
shows $unat\ (of\text{-}nat\ x * of\text{-}nat\ (size\text{-}of\ TYPE('a)) + (of\text{-}nat\ k :: addr))$
$= x * size\text{-}of\ TYPE('a) + k$
⟨*proof*⟩

lemma *ptr-span-array-index-disj*:
fixes $p :: ('a :: array\text{-}outer\text{-}max\text{-}size['b :: array\text{-}max\text{-}count])\ ptr$
assumes *n-bound*: $n < CARD\ ('b)$
assumes *i-bound*: $i < n$
shows *disjnt* $(ptr\text{-}span\ (array\text{-}ptr\text{-}index\ p\ False\ n))\ (ptr\text{-}span\ (array\text{-}ptr\text{-}index\ p\ False\ i))$
⟨*proof*⟩

lemma *ptr-span-array-ptr-index-disj*:
fixes $p :: ('a :: array\text{-}outer\text{-}max\text{-}size['b :: array\text{-}max\text{-}count])\ ptr$
fixes $q :: ('a['b :: array\text{-}max\text{-}count])\ ptr$
assumes *n-bound*: $n < CARD\ ('b)$
assumes *m-bound*: $m < CARD\ ('b)$
assumes *disj*:*disjnt* $(ptr\text{-}span\ p)\ (ptr\text{-}span\ q)$
shows *disjnt* $(ptr\text{-}span\ (array\text{-}ptr\text{-}index\ p\ False\ n))\ (ptr\text{-}span\ (array\text{-}ptr\text{-}index\ q\ False\ m))$
⟨*proof*⟩

named-theorems *select-conv* **and** *select-conv-independent* **and** *valid-conv* **and** *valid-array-conv* **and**
gen-update-commute **and** *gen-outer-update-commute* **and** *update-commute*

locale *pointer-array-lense* = *pointer-lense r w*
for $r :: 's \Rightarrow 'a :: array\text{-}outer\text{-}max\text{-}size\ ptr \Rightarrow 'a$
and $w :: 'a\ ptr \Rightarrow ('a \Rightarrow 'a) \Rightarrow 's \Rightarrow 's$
begin

definition *heap-array* ::'s ⇒ ('a['b::*array-max-count*]) *ptr* ⇒ 'a['b] **where**
heap-array s p = *FCP* (λ*i*. *r s* (*array-ptr-index p False i*))

lemmas [*read-stack-byte-ZERO-step-subst*] = *heap-array-def*

definition *heap-array-map* :: ('a['b]) *ptr* ⇒ ('a['b::*array-max-count*] ⇒ 'a['b]) ⇒ 's ⇒ 's **where**
heap-array-map p f s ≡
fold (λ*i*. *w* (*array-ptr-index p False i*) (λ-. (*f* (*heap-array s p*)).[*i*])) [*0*..<*CARD*('*b*)] *s*

lemma *element-heap-eq-heap-array-eq* [*select-conv*]: *r s* = *r s'* ⟹ *heap-array s* = *heap-array s'*
⟨*proof*⟩

lemma *fold-write-independent-field*:
fixes *p*::('a['b::*array-max-count*]) *ptr*
assumes *field-independent*: (⋀*p x s*. *f* (*w p* (λ-. *x*) *s*) = *f s*)
assumes *n-bound*: *n* ≤ *CARD*('*b*)
shows *f* (*fold* (λ*i*. *w* (*array-ptr-index p False i*) (λ-. *g* (*heap-array s p*).[*i*])) [*0*..<*n*] *t*) = *f t*
⟨*proof*⟩

lemma *heap-array-map-independent-field* [*select-conv-independent*]:
assumes *field-independent*: (⋀*p x s*. *f* (*w p* (λ-. *x*) *s*) = *f s*)
shows *f* (*heap-array-map p g s*) = *f s*
⟨*proof*⟩

lemma *fold-write-independent-field-upd-commute*:
fixes *p*::('a['b::*array-max-count*]) *ptr*
assumes *write-commute*: ⋀*p x s*. (*f-upd* (*w p* (λ-. *x*) *s*)) = *w p* (λ-. *x*) (*f-upd s*)
assumes *n-bound*: *n* ≤ *CARD*('*b*)
shows *f-upd* (*fold* (λ*i*. *w* (*array-ptr-index p False i*) (λ-. *g* (*heap-array s p*).[*i*])) [*0*..<*n*] *t*) =
fold (λ*i*. *w* (*array-ptr-index p False i*) (λ-. *g* (*heap-array s p*).[*i*])) [*0*..<*n*] (*f-upd t*)
⟨*proof*⟩

lemma *heap-array-map-independent-field-commute* [*gen-update-commute*]:
assumes *read-independent*: ⋀*s*. *r* (*f-upd s*) = *r s*
assumes *write-independent*: ⋀*p x s*. (*f-upd* (*w p* (λ-. *x*) *s*)) = *w p* (λ-. *x*) (*f-upd s*)
shows *f-upd* (*heap-array-map p g s*) = (*heap-array-map p g* (*f-upd s*))
⟨*proof*⟩

lemma *heap-array-index*[*simp*]: *i* < *CARD*('*b*) ⟹
heap-array s (*p*::('a['b::*array-max-count*]) *ptr*).[*i*] = *r s* (*array-ptr-index p False i*)

⟨*proof*⟩

lemma *read-write-same-fold-aux*:
fixes *p*::('*a*['*b*::*array-max-count*]) *ptr*
assumes *n-bound*: *n* ≤ *CARD* ('*b*)
assumes *i-bound*: *i* < *n*
shows *r* (*fold* (λ*i*. *w* (*array-ptr-index p False i*) (λ-. *f* (*heap-array s p*).[*i*])) [*0*..<*n*] *s*) (*array-ptr-index p False i*) = *f* (*heap-array s p*).[*i*]
⟨*proof*⟩

lemma *array-read-write-same*: *heap-array* (*heap-array-map p f s*) *p* = *f* (*heap-array s p*)
⟨*proof*⟩

lemma *read-write-other-fold-aux*:
fixes *p*::('*a*['*b*::*array-max-count*]) *ptr*
fixes *p'*::('*a*['*b*::*array-max-count*]) *ptr*
assumes *n-bound*: *n* ≤ *CARD* ('*b*)
assumes *i-bound*: *i* < *CARD* ('*b*)
assumes *disj*:*disjnt* (*ptr-span p*) (*ptr-span p'*)
shows *r* (*fold* (λ*i*. *w* (*array-ptr-index p False i*) (λ-. *f* (*heap-array s p*).[*i*])) [*0*..<*n*] *s*) (*array-ptr-index p' False i*) =
r s (*array-ptr-index p' False i*)
⟨*proof*⟩

lemma *array-read-write-other*:
fixes *p*::('*a*['*b*::*array-max-count*]) *ptr*
fixes *p'*::('*a*['*b*::*array-max-count*]) *ptr*
assumes *disj*:*disjnt* (*ptr-span p*) (*ptr-span p'*)
shows *heap-array* (*heap-array-map p f s*) *p'* = *heap-array s p'*
⟨*proof*⟩

lemma *write-cong-fold-aux*:
fixes *p*::('*a*['*b*::*array-max-count*]) *ptr*
assumes *f-f'*: *f* (*heap-array s p*) = *f'* (*heap-array s p*)
assumes *n-bound*: *n* ≤ *CARD*('*b*)
shows *fold* (λ*i*. *w* (*array-ptr-index p False i*) (λ-. *f* (*heap-array s p*).[*i*])) [*0*..<*n*] *s* =
fold (λ*i*. *w* (*array-ptr-index p False i*) (λ-. *f'* (*heap-array s p*).[*i*])) [*0*..<*n*] *s*
⟨*proof*⟩

lemma *array-write-cong*:
fixes *p*::('*a*['*b*::*array-max-count*]) *ptr*
assumes *eq*: *f* (*heap-array s p*) = *f'* (*heap-array s p*)
shows *heap-array-map p f s* = *heap-array-map p f' s*
⟨*proof*⟩

lemma *array-write-same-triv*[*simp*]:

fixes p::$('a['b::array\text{-}max\text{-}count])$ ptr
shows $heap\text{-}array\text{-}map\ p\ (\lambda\text{-}.\ f\ (heap\text{-}array\ s\ p))\ s = heap\text{-}array\text{-}map\ p\ f\ s$
⟨*proof*⟩

lemma *array-write-fold-same-aux*:
fixes p::$('a['b::array\text{-}max\text{-}count])$ ptr
assumes *f-id*: $f\ (heap\text{-}array\ s\ p) = heap\text{-}array\ s\ p$
assumes *n-bound*: $n \leq CARD('b)$
shows $fold\ (\lambda i.\ w\ (array\text{-}ptr\text{-}index\ p\ False\ i)\ (\lambda\text{-}.\ f\ (heap\text{-}array\ s\ p).[i]))\ [0..<n]$
$s = s$
⟨*proof*⟩

lemma *array-write-same*:
fixes p::$('a['b::array\text{-}max\text{-}count])$ ptr
assumes *f-id*: $f\ (heap\text{-}array\ s\ p) = heap\text{-}array\ s\ p$
shows $heap\text{-}array\text{-}map\ p\ f\ s = s$
⟨*proof*⟩

lemma *array-write-triv* [*simp*]:
fixes p::$('a['b::array\text{-}max\text{-}count])$ ptr
shows $heap\text{-}array\text{-}map\ p\ (\lambda\text{-}.\ heap\text{-}array\ s\ p)\ s = s$ **and** $heap\text{-}array\text{-}map\ p\ (\lambda x.$
$x)\ s = s$
⟨*proof*⟩

lemma *write-fold-other-commute*:
fixes p::$nat \Rightarrow 'a\ ptr$
and q:: $'a\ ptr$
assumes *disj*: $\bigwedge i.\ i < n \Longrightarrow disjnt\ (ptr\text{-}span\ q)\ (ptr\text{-}span\ (p\ i))$
shows
$w\ q\ f\ \ (fold\ (\lambda i.\ w\ (p\ i)\ (g\ i))\ [0..<n]\ s) =$
$\ fold\ (\lambda i.\ w\ (p\ i)\ (g\ i)\ )\ [0..<n]\ (w\ q\ f\ s)$
⟨*proof*⟩

lemma *write-fold-arr-commute*:
fixes p::$('a['b::array\text{-}max\text{-}count])$ ptr
and arr:: $'a['b::array\text{-}max\text{-}count]$
assumes *n-bound*: $n < CARD('b)$
shows
$w\ (array\text{-}ptr\text{-}index\ p\ False\ n)\ (\lambda\text{-}.\ arr.[n])$
$\ (fold\ (\lambda i.\ w\ (array\text{-}ptr\text{-}index\ p\ False\ i)\ (\lambda\text{-}.\ arr.[i]))$
$\ \ [0..<n]\ s) =$
$\ fold$
$\ \ (\lambda i.\ w\ (array\text{-}ptr\text{-}index\ p\ False\ i)\ (\lambda\text{-}.\ arr.[i]))$
$\ \ [0..<n]\ (\ w\ (array\text{-}ptr\text{-}index\ p\ False\ n)\ \ (\lambda\text{-}.\ arr.[n])\ s)$
⟨*proof*⟩

lemma *array-fold-fuse-aux*:
fixes *p*::('*a*['*b*::*array-max-count*]) *ptr*
fixes *f*::'*a*['*b*] ⇒ '*a*['*b*]
fixes *g*::'*a*['*b*] ⇒ '*a*['*b*]
assumes *n-bound*: *n* ≤ *CARD*('*b*)
shows *fold*
(λ*i*. *w* (*array-ptr-index p False i*) (λ-. *f* (*g* (*heap-array s p*)).[*i*]))
[*0*..<*n*]
(*fold*
(λ*i*. *w* (*array-ptr-index p False i*) (λ-. *g* (*heap-array s p*).[*i*]))
[*0*..<*n*] *t*) =
fold
(λ*i*. *w* (*array-ptr-index p False i*) (λ-. *f* (*g* (*heap-array s p*)).[*i*]))
[*0*..<*n*] *t*
⟨*proof*⟩

lemma *array-write-comp*:
fixes *p*::('*a*['*b*::*array-max-count*]) *ptr*
shows *heap-array-map p f* (*heap-array-map p g s*) = *heap-array-map p* (*f o g*) *s*
⟨*proof*⟩

lemma *fold-fold-other-commute*:
fixes *p*::*nat* ⇒ '*a ptr*
and *q*:: *nat* ⇒ '*a ptr*
assumes *disj*: ⋀*i j*. *i* < *n* ⟹ *j* < *m* ⟹ *disjnt* (*ptr-span* (*p i*)) (*ptr-span* (*q j*))
shows
fold (λ*i*. *w* (*p i*) (*f i*)) [*0*..<*n*] (*fold* (λ*i*. *w* (*q i*) (*g i*)) [*0*..<*m*] *s*) =
fold (λ*i*. *w* (*q i*) (*g i*)) [*0*..<*m*] (*fold* (λ*i*. *w* (*p i*) (*f i*)) [*0*..<*n*] *s*)
⟨*proof*⟩

lemma *array-write-other-commute*:
fixes *p*::('*a*['*b*::*array-max-count*]) *ptr*
fixes *q*::('*a*['*b*::*array-max-count*]) *ptr*
assumes *disj*: *disjnt* (*ptr-span p*) (*ptr-span q*)
shows *heap-array-map q g* (*heap-array-map p f s*) = *heap-array-map p f* (*heap-array-map q g s*)
⟨*proof*⟩

sublocale *array*: *pointer-lense heap-array heap-array-map*
⟨*proof*⟩

end

locale *pointer-two-dimensional-array-lense* = *pointer-array-lense r w*

for *r*:: *'s* ⇒ *'a*:: *array-inner-max-size ptr* ⇒ *'a*
and *w*:: *'a ptr* ⇒ (*'a* ⇒ *'a*) ⇒ *'s* ⇒ *'s*
begin
sublocale *outer*: *pointer-array-lense heap-array heap-array-map*
⟨*proof*⟩

lemmas *outer.heap-array-map-independent-field-commute* [*gen-outer-update-commute*]

end

locale *valid-array-base* =
fixes *vgetter*:: (*'t* ⇒ *'a*::*array-outer-max-size ptr* ⇒ *bool*)
begin

definition *valid-array* ::*'t* ⇒ (*'a*[*'b*::*array-max-count*]) *ptr* ⇒ *bool* **where**
[*valid-array-defs*]: *valid-array s p* = (∀ *i* < *CARD*(*'b*). *vgetter s* (*array-ptr-index p False i*))

lemma *element-vgetter-eq-valid-array-eq* [*valid-array-conv*]: *vgetter s* = *vgetter s'* ⟹ *valid-array s* = *valid-array s'*
⟨*proof*⟩

end

locale *valid-pointer-array-lense* = *pointer-array-lense r w* +
valid-array-base vgetter +
write-preserves-valid vgetter w
for *r*:: *'s* ⇒ *'a*:: *array-outer-max-size ptr* ⇒ *'a*
and *w*:: *'a ptr* ⇒ (*'a* ⇒ *'a*) ⇒ *'s* ⇒ *'s*
and *vgetter*:: (*'s* ⇒ *'a ptr* ⇒ *bool*)
begin

lemma *setter-fold-keeps-vgetter*:
fixes *p'*:: (*'a*[*'b*::*array-max-count*]) *ptr*
fixes *p*:: *'a ptr*
assumes *n-bound*: *n* ≤ *CARD*(*'b*)
shows *vgetter* (*fold* (λ*i*. *w* (*array-ptr-index p' False i*) (λ-. *f* (*heap-array s p'*).[*i*])) [*0*..<*n*] *s*) *p* = *vgetter s p*
⟨*proof*⟩

lemma *heap-array-map-keeps-vgetter*:
fixes *p*:: (*'a*[*'b*::*array-max-count*]) *ptr*
fixes *p'*:: (*'a*[*'b*]) *ptr*
shows *valid-array* (*heap-array-map p' f s*) *p* = *valid-array s p*
⟨*proof*⟩

sublocale *array*: *write-preserves-valid valid-array heap-array-map*
⟨*proof*⟩

lemma *heap-array-map-keeps-vgetter-element*[*simp*]:
fixes *p'*:: ('*a*['*b*::*array-max-count*]) *ptr*
shows *vgetter* (*heap-array-map p' f s*) = *vgetter s*
⟨*proof*⟩

lemma *getter-keeps-valid-array*[*simp*]:
fixes *p'*:: '*a ptr*
shows (*valid-array*::'*s* ⇒ ('*a*['*b*::*array-max-count*]) *ptr* ⇒ *bool*) (*w p' f s*) = *valid-array s*
⟨*proof*⟩

lemma *getter-keeps-valid-array-pointwise*[]:
fixes *p'*:: '*a ptr*
fixes *p* :: ('*a*['*b*::*array-max-count*]) *ptr*
shows (*valid-array*::'*s* ⇒ ('*a*['*b*::*array-max-count*]) *ptr* ⇒ *bool*) (*w p' f s*) *p* = *valid-array s p*
⟨*proof*⟩
end

locale *valid-pointer-two-dimensional-array-lense* = *pointer-two-dimensional-array-lense r w* +
valid-array-base vgetter +
write-preserves-valid vgetter w
for *r*:: '*s* ⇒ '*a*:: *array-inner-max-size ptr* ⇒ '*a*
and *w*:: '*a ptr* ⇒ ('*a* ⇒ '*a*) ⇒ '*s* ⇒ '*s*
and *vgetter*:: ('*s* ⇒ '*a ptr* ⇒ *bool*)
begin
sublocale *inner*: *valid-pointer-array-lense r w vgetter*
⟨*proof*⟩

sublocale *outer*: *valid-pointer-array-lense heap-array heap-array-map valid-array*
⟨*proof*⟩

end

locale *array-typ-heap-simulation* =
lense t-hrs t-hrs-update +
read-simulation st v r t-hrs +
write-simulation t-hrs t-hrs-update st v w +
write-preserves-valid v w +
valid-implies-cguard st v +
valid-only-typ-desc-dependent t-hrs st v +
pointer-array-lense r w +
valid-array-base v
for
st:: '*s* ⇒ '*t* **and**
r:: '*t* ⇒ '*a*::*array-outer-max-size ptr* ⇒ '*a* **and**
w:: '*a ptr* ⇒ ('*a* ⇒ '*a*) ⇒ '*t* ⇒ '*t* **and**

v:: *'t* ⇒ *'a ptr* ⇒ *bool* **and**
t-hrs :: *'s* ⇒ *heap-raw-state* **and**
t-hrs-update:: (*heap-raw-state* ⇒ *heap-raw-state*) ⇒ *'s* ⇒ *'s*
begin

sublocale *typ-heap-simulation*
⟨*proof*⟩

sublocale *valid-pointer-array-lense r w v*
⟨*proof*⟩

lemmas *typ-heap-simulation = typ-heap-simulation-axioms*

lemma *valid-array-implies-safe*: *valid-array* (*st s*) *p* ⟹ *c-guard p*
⟨*proof*⟩

lemma *heap-array-data-correct*:
assumes *valid-arr*: *valid-array* (*st s*) *p*
shows *heap-array* (*st s*) *p* = *h-val* (*hrs-mem* (*t-hrs s*)) *p*
⟨*proof*⟩

lemma *write-fold-commutes*:
fixes *p*:: (*'a*[*'b*::*array-max-count*]) *ptr*
assumes *n-bound*: *n* ≤ *CARD*(*'b*)
assumes *valid*: *valid-array* (*st s*) *p*
shows *st* (*t-hrs-update* (*hrs-mem-update* (*fold* (λ*i*. *heap-update* (*array-ptr-index p False i*) (*x*.[*i*])) [*0*..<*n*])) *s*) =
fold (λ*i*. *w* (*array-ptr-index p False i*) (λ-. *x*.[*i*])) [*0*..<*n*] (*st s*)
⟨*proof*⟩

lemma *heap-array-map-commutes*:
fixes *p*:: (*'a*[*'b*::*array-max-count*]) *ptr*
assumes *valid*: *valid-array* (*st s*) *p*
shows *st* (*t-hrs-update* (*hrs-mem-update* (*heap-update p x*)) *s*) = *heap-array-map p* (λ-. *x*) (*st s*)
⟨*proof*⟩

lemma *write-padding-fold-commutes*:
fixes *p*:: (*'a*[*'b*::*array-max-count*]) *ptr*
assumes *n-bound*: *n* ≤ *CARD*(*'b*)
assumes *valid*: *valid-array* (*st s*) *p*
assumes *lbs*: *length bs* = *CARD*(*'b*) ∗ *size-of TYPE*(*'a*)
shows *st* (*t-hrs-update*
(*hrs-mem-update*
(*fold*
(λ*i*. *heap-update-padding* (*array-ptr-index p False i*) (*x*.[*i*])
(*take* (*size-of TYPE*(*'a*)) (*drop* (*i* ∗ *size-of TYPE*(*'a*)) *bs*)))

[0..<n]))
s) =
fold (λi. w (array-ptr-index p False i) (λ-. x.[i])) [0..<n] (st s)
⟨proof⟩

lemma *heap-array-padding-map-commutes*:
fixes *p*:: ('a['b::*array-max-count*]) *ptr*
assumes *valid*: *valid-array* (*st s*) *p*
assumes *bound*: *length bs* = *size-of TYPE*('a['b])
shows *st* (*t-hrs-update* (*hrs-mem-update* (*heap-update-padding p x bs*)) *s*) = *heap-array-map p* (λ-. *x*) (*st s*)
⟨*proof*⟩

lemma *array-valid-same-typ-desc*:
fixes *p*:: ('a['b::*array-max-count*]) *ptr*
assumes *typ-eq*: *hrs-htd* (*t-hrs s*) = *hrs-htd* (*t-hrs t*)
shows *valid-array* (*st s*) *p* = *valid-array* (*st t*) *p*
⟨*proof*⟩

sublocale *array*: *typ-heap-simulation st heap-array heap-array-map valid-array t-hrs t-hrs-update*
⟨*proof*⟩

end

locale *two-dimensional-array-typ-heap-simulation* =
lense t-hrs t-hrs-update +
read-simulation st v r t-hrs +
write-simulation t-hrs t-hrs-update st v w +
write-preserves-valid v w +
valid-implies-cguard st v +
valid-only-typ-desc-dependent t-hrs st v +
pointer-two-dimensional-array-lense r w
for
st:: 's ⇒ 't **and**
r:: 't ⇒ 'a::*array-inner-max-size ptr* ⇒ 'a **and**
w:: 'a *ptr* ⇒ ('a ⇒ 'a) ⇒ 't ⇒ 't **and**
v:: 't ⇒ 'a *ptr* ⇒ *bool* **and**
t-hrs :: 's ⇒ *heap-raw-state* **and**
t-hrs-update:: (*heap-raw-state* ⇒ *heap-raw-state*) ⇒ 's ⇒ 's
begin

sublocale *inner*: *array-typ-heap-simulation st r w v t-hrs t-hrs-update*
⟨*proof*⟩

lemmas *inner-typ-heap-simulation* = *inner.typ-heap-simulation*

sublocale *outer*: *array-typ-heap-simulation st heap-array heap-array-map inner.valid-array t-hrs t-hrs-update*
⟨*proof*⟩

lemmas *outer-typ-heap-simulation* = *outer.typ-heap-simulation*

end

lemma *root-ptr-valid-field-lvalue*:
fixes *p*::*'a*::*mem-type ptr*
fixes *q*:: *'b*::*mem-type ptr*
assumes *root-p*: *root-ptr-valid d p*
assumes *root-q*: *root-ptr-valid d q*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'b*)) *f 0* = *Some* (*t*, *k*)
assumes *other*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*'b*)
shows *ptr-val p* = &(*q*→*f*) ⟷ *False*
⟨*proof*⟩

lemma *root-ptr-valid-field-lvalue'*:
fixes *q*:: *'b*::*mem-type ptr*
assumes *root-p*: *root-ptr-valid d* (*PTR* (*'a*::*mem-type*)(&(*q*→*f*)))
assumes *root-q*: *root-ptr-valid d q*
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'b*)) *f 0* = *Some* (*t*, *k*)
assumes *other*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*'b*)
shows *False*
⟨*proof*⟩

lemma *root-ptr-valid-array-ptr-index-dim1*:
fixes *q*:: (*'a*::*array-outer-max-size*[*'c*::*array-max-count*]) *ptr*
assumes *root-p*: *root-ptr-valid d* (*array-ptr-index q False i*)
assumes *root-q*: *root-ptr-valid d q*
assumes *other*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*'a*[*'c*])
assumes *i-bound*: *i* < *CARD*(*'c*)
shows *False*
⟨*proof*⟩

lemma *root-ptr-valid-field-lvalue-array-ptr-index-dim1*:
fixes *q*:: *'b*::*mem-type ptr*
assumes *root-p*: *root-ptr-valid d* (*array-ptr-index* (*PTR*(*'a*[*'c*]) &(*q*→*f*)) *False i*)
assumes *root-q*: *root-ptr-valid d q*
assumes *other*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*'b*)
assumes *i-bound*: *i* < *CARD*(*'c*)
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'b*)) *f 0* = *Some* (*t*, *k*)
assumes *t*: *export-uinfo t* = *typ-uinfo-t* (*TYPE*(*'a*::*array-outer-max-size*[*'c*::*array-max-count*]))
shows *False*
⟨*proof*⟩

lemma *root-ptr-valid-field-lvalue-array-ptr-index-dim1'*:
fixes *q*:: *'b*::*mem-type ptr*
assumes *root-p*: *root-ptr-valid d* (*array-ptr-index* (*PTR*((*'a*::*array-outer-max-size*[*'c*::*array-max-count*]))
&(*q*→*f*)) *False i*)
assumes *root-q*: *root-ptr-valid d q*
assumes *other*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*'b*)
assumes *i-bound*: *i* < *CARD*(*'c*)
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'b*)) *f 0* = *Some* (*adjust-ti* (*typ-info-t*
(*TYPE*(*'a*[*'c*]))) *acc upd*, *k*)
assumes *fg*: *fg-cons acc upd*
shows *False*
⟨*proof*⟩

lemma *root-ptr-valid-array-ptr-index-dim2*:
fixes *q*:: ((*'a*::*array-inner-max-size*[*'c*::*array-max-count*])[*'d*::*array-max-count*]) *ptr*
assumes *root-p*: *root-ptr-valid d* (*array-ptr-index* (*array-ptr-index q False i*) *False*
j)
assumes *root-q*: *root-ptr-valid d q*
assumes *other*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*'a*[*'c*][*'d*])
assumes *i-bound*: *i* < *CARD*(*'d*)
assumes *j-bound*: *j* < *CARD*(*'c*)
shows *False*
⟨*proof*⟩

lemma *root-ptr-valid-field-lvalue-array-ptr-index-dim2*:
fixes *q*:: *'b*::*mem-type ptr*
assumes *root-p*: *root-ptr-valid d* (*array-ptr-index* (*array-ptr-index* (*PTR*(*'a*[*'c*][*'d*])
&(*q*→*f*)) *False i*) *False j*)
assumes *root-q*: *root-ptr-valid d q*
assumes *other*: *typ-uinfo-t TYPE*(*'a*) ≠ *typ-uinfo-t TYPE*(*'b*)
assumes *i-bound*: *i* < *CARD*(*'d*)
assumes *j-bound*: *j* < *CARD*(*'c*)
assumes *fl*: *field-lookup* (*typ-info-t TYPE*(*'b*)) *f 0* = *Some* (*t*, *k*)
assumes *t*: *export-uinfo t* = *typ-uinfo-t* (*TYPE*((*'a*::*array-inner-max-size*[*'c*::*array-max-count*])[*'d*::*array-max*
shows *False*
⟨*proof*⟩

context *open-types*
begin

definition
typ-heap-simulation-of-field (*st*::*'s* ⇒ *'t*) *t-hrs t-hrs-update heap-typing-upd f' r'*
w' ⟷
(∀ *d p f*. *heap-typing-upd d o w' p f* = *w' p f o heap-typing-upd d*) ∧
(∀ *t u n*.
field-ti TYPE(*'a*::*mem-type*) *f'* = *Some t* ⟶
field-lookup (*typ-uinfo-t TYPE*(*'a*)) *f' 0* = *Some* (*u*, *n*) ⟶
partial-pointer-lense (*merge-ti t*) *r' w'* ∧

(∀ (p::'a ptr) s. (∀ a∈ptr-span p. root-ptr-valid (hrs-htd (t-hrs s)) (PTR(stack-byte) a)) ⟶
r' (st s) p ZERO('a) = ZERO('a)) ∧
(∀ (p::'a ptr) x h. ptr-valid-u u (hrs-htd (t-hrs h)) &(p→f') ⟶
st (t-hrs-update (hrs-mem-update (heap-upd-list (size-td u) &(p→f') (access-ti t x))) h) =
w' p x (st h)))

end

definition
pointer-writer-disjnt ::
('a::c-type ptr ⇒ 't upd) ⇒ ('b::c-type ptr ⇒ 't upd) ⇒ bool
where
pointer-writer-disjnt f g ⟷
(∀ p q. disjnt (ptr-span p) (ptr-span q) ⟶ f p ∘ g q = g q ∘ f p)

definition
pointer-writer-disjnt-eq ::
('a::c-type ptr ⇒ 'x ⇒ 't upd) ⇒ ('b::c-type ptr ⇒ 'y ⇒ 't upd) ⇒ bool
where
pointer-writer-disjnt-eq f g ⟷ (∀ p q x y.
disjnt (ptr-span p) (ptr-span q) ∨ ptr-val p = ptr-val q ⟶ f p x ∘ g q y = g q y ∘ f p x)

named-theorems *pointer-writer-disjnt-intros*

lemma *pointer-writer-disjnt-eq*:
assumes nm: ∃ n1 n2.
field-lookup (typ-uinfo-t TYPE('a::mem-type)) f1 0 =
Some (typ-uinfo-t TYPE('b::mem-type), n1) ∧
field-lookup (typ-uinfo-t TYPE('a)) f2 0 = Some (typ-uinfo-t TYPE('c::mem-type), n2)
assumes w1-w2: ⋀x y. pointer-writer-disjnt (λp. w1 p x) (λq. w2 q y)
shows disj-fn f1 f2 ⟶ pointer-writer-disjnt-eq
(λ(p::'a ptr). w1 (PTR('b) &(p→f1)))
(λ(p::'a ptr). w2 (PTR('c) &(p→f2))) (**is** ?disj ⟶ ?goal)
⟨proof⟩

lemma *pointer-writer-disjnt-sym*:
pointer-writer-disjnt f g ⟹ *pointer-writer-disjnt g f*
⟨proof⟩

lemma *pointer-writer-disjnt-id-left*:
pointer-writer-disjnt (λp h. h) w
⟨proof⟩

lemma *pointer-writer-disjnt-id-left'*:
pointer-writer-disjnt (λp. id) w

⟨*proof*⟩

lemma *pointer-writer-disjnt-comp-left*:
pointer-writer-disjnt w1 w ⟹ *pointer-writer-disjnt w2 w* ⟹
pointer-writer-disjnt (λ*p h*. *w1 p* (*w2 p h*)) *w*
⟨*proof*⟩

lemma *pointer-writer-disjnt-comp-left′*:
pointer-writer-disjnt w1 w ⟹ *pointer-writer-disjnt w2 w* ⟹
pointer-writer-disjnt (λ*p*. *w1 p* ∘ *w2 p*) *w*
⟨*proof*⟩

lemma *pointer-writer-disjnt-fold-left*:
list-all (λ*w′*. *pointer-writer-disjnt* (*f w′*) *w*) *ws* ⟹
pointer-writer-disjnt (λ*p*. *fold* (λ*w*. *f w p*) *ws*) *w*
⟨*proof*⟩

lemma *pointer-writer-disjntI*:
(⋀*p q h*. *w1 p* (*w2 q h*) = *w2 q* (*w1 p h*)) ⟹ *pointer-writer-disjnt w1 w2*
⟨*proof*⟩

lemma *pointer-writer-disjntD*:
pointer-writer-disjnt w1 w2 ⟹ *disjnt* (*ptr-span p*) (*ptr-span q*) ⟹
w1 p (*w2 q h*) = *w2 q* (*w1 p h*)
⟨*proof*⟩

lemma *pointer-writer-disjnt-ptr-map-left*:
fixes *w1* :: *′a*::*mem-type ptr* ⇒ *′s upd* **and** *w2* :: *′b*::*c-type ptr* ⇒ *′s upd*
assumes *w1-w2*: *pointer-writer-disjnt w1 w2*
assumes ⋀(*p*::*′c*::*c-type ptr*). *ptr-span* (*F p*) ⊆ *ptr-span p*
shows *pointer-writer-disjnt* (λ(*p*::*′c ptr*). *w1* (*F p*)) *w2*
⟨*proof*⟩

lemma *pointer-writer-disjnt-ptr-left*:
fixes *w1* :: *′a*::*mem-type ptr* ⇒ *′s upd* **and** *w2* :: *′b*::*mem-type ptr* ⇒ *′s upd*
assumes *w1-w2*: *pointer-writer-disjnt w1 w2*
assumes *n*:
∃ *n*. *field-lookup* (*typ-uinfo-t TYPE*(*′c*::*mem-type*)) *f 0* = *Some* (*typ-uinfo-t TYPE*(*′a*), *n*)
shows *pointer-writer-disjnt* (λ(*p*::*′c ptr*). *w1* (*PTR*(*′a*) &(*p*→*f*))) *w2*
⟨*proof*⟩

lemma *pointer-writer-disjnt-ptr-corce*:
fixes *w1* :: *′a*::*mem-type ptr* ⇒ *′s upd* **and** *w2* :: *′b*::*mem-type ptr* ⇒ *′s upd*
assumes *w1-w2*: *pointer-writer-disjnt w1 w2*
and *size*: *size-of TYPE*(*′a*) ≤ *size-of TYPE*(*′c*)
shows *pointer-writer-disjnt* (λ(*p*::*′c*::*c-type ptr*). *w1* (*PTR-COERCE*(*′c* → *′a*) *p*)) *w2*
⟨*proof*⟩

lemma *pointer-writer-disjnt-ptr-corce-signed*:
fixes *w1* :: *'a::len8 word ptr* ⇒ *'s upd* **and** *w2* :: *'b::mem-type ptr* ⇒ *'s upd*
assumes *w1-w2*: *pointer-writer-disjnt w1 w2*
shows *pointer-writer-disjnt* (λ(*p*::*'a signed word ptr*).
w1 (*PTR-COERCE*(*'a signed word* → *'a word*) *p*)) *w2*
⟨*proof*⟩

lemma *pointer-writer-disjnt-ptr-corce-signed'*:
fixes *w1* :: *'a::len8 word ptr* ⇒ *'s upd* **and** *w2* :: *'b::mem-type ptr* ⇒ *'s upd*
shows *pointer-writer-disjnt w2 w1* ⟹
pointer-writer-disjnt w2 (λ(*p*::*'a signed word ptr*).
w1 (*PTR-COERCE*(*'a signed word* → *'a word*) *p*))
⟨*proof*⟩

lemma (**in** *pointer-lense*) *pointer-writer-disjnt-replace-by-const*:
(⋀*x*. *pointer-writer-disjnt* (λ*p*. *w p* (λ-. *x*)) *w'*) ⟹ *pointer-writer-disjnt* (λ*p*. *w p f*) *w'*
⟨*proof*⟩

lemma (**in** *pointer-lense*) *read-commute-of-pointer-writer-disjnt*:
assumes *w'*: ⋀*f*. *pointer-writer-disjnt* (λ*p*. *w p f*) *w'*
assumes *p-q*: *disjnt* (*ptr-span p*) (*ptr-span q*)
shows *r* (*w' q h*) *p* = *r h p*
⟨*proof*⟩

lemma (**in** *pointer-lense*) *read-commute-of-pointer-writer-disjnt'*:
assumes *w'*: ⋀*f*. *pointer-writer-disjnt w'* (λ*p*. *w p f*)
assumes *p-q*: *disjnt* (*ptr-span p*) (*ptr-span q*)
shows *r* (*w' q h*) *p* = *r h p*
⟨*proof*⟩

lemma (**in** *pointer-lense*) *read-commute-of-commute*:
assumes *w'*: ⋀*p f*. *w p f o w'* = *w' o w p f*
shows *r* (*w' h*) *p* = *r h p*
⟨*proof*⟩

lemma (**in** *pointer-lense*) *read-commute-of-commute'*:
assumes *w'*: ⋀*p f*. *w' o w p f* = *w p f o w'*
shows *r* (*w' h*) *p* = *r h p*
⟨*proof*⟩

lemma (**in** *lense*) *get-commute-of-commute*:
assumes *w'*: ⋀*f*. *w' o upd f* = *upd f o w'*
shows *get* (*w' h*) = *get h*
⟨*proof*⟩

lemma (**in** *pointer-lense*) *pointer-writer-disjnt-replace-dep-by-const*:
assumes ∗: ⋀*x*. *pointer-writer-disjnt* (λ*p*. *w p x*) *w'*

shows *pointer-writer-disjnt* ($\lambda p\ s.\ w\ p\ (f\ (r\ s\ p))\ s$) w'
⟨*proof*⟩

lemma (**in** *pointer-lense*) *pointer-writer-disjnt-upd*[*pointer-writer-disjnt-intros*]:
pointer-writer-disjnt ($\lambda p.\ w\ p\ x$) ($\lambda p.\ w\ p\ y$)
⟨*proof*⟩

lemma (**in** *partial-pointer-lense*) *read-commute-of-pointer-writer-disjnt*:
assumes w': $\bigwedge f.$ *pointer-writer-disjnt* ($\lambda p.\ w\ p\ f$) w'
assumes *p-q*: *disjnt* (*ptr-span p*) (*ptr-span q*)
shows $r\ (w'\ q\ h)\ p\ y = r\ h\ p\ y$
⟨*proof*⟩

lemma *pointer-lense-upd-fun-of-lense*:
fixes *get upd* **assumes** *lense get upd*
shows *pointer-lense get* ($\lambda p\ f.$ *upd* (*upd-fun p f*))
⟨*proof*⟩

locale *open-types-heap-typing-state* = *open-types* $\mathcal{T}$ +
heap-typing-state heap-typing heap-typing-upd
for
$\mathcal{T}$ **and**
heap-typing :: $'t \Rightarrow$ *heap-typ-desc* **and**
heap-typing-upd :: (*heap-typ-desc* $\Rightarrow$ *heap-typ-desc*) $\Rightarrow$ $'t \Rightarrow$ $'t$

locale *typ-heap-simulation-open-types* =
typ-heap-simulation st r w v t-hrs t-hrs-update +
open-types-heap-typing-state $\mathcal{T}$ *heap-typing heap-typing-upd*
for
$\mathcal{T}$ **and**
st:: $'s \Rightarrow 't$ **and**
r:: $'t \Rightarrow ('a$::{*xmem-type*, *stack-type*}) *ptr* $\Rightarrow 'a$ **and**
w:: $'a$ *ptr* $\Rightarrow (('a \Rightarrow 'a) \Rightarrow 't \Rightarrow 't$ **and**
v:: $'t \Rightarrow 'a$ *ptr* $\Rightarrow$ *bool* **and**
t-hrs :: $'s \Rightarrow$ *heap-raw-state* **and**
t-hrs-update:: (*heap-raw-state* $\Rightarrow$ *heap-raw-state*) $\Rightarrow 's \Rightarrow 's$ **and**
heap-typing :: $'t \Rightarrow$ *heap-typ-desc* **and**
heap-typing-upd :: (*heap-typ-desc* $\Rightarrow$ *heap-typ-desc*) $\Rightarrow$ $'t \Rightarrow$ $'t$ +
assumes *heap-typing-commutes*[*simp*]: *heap-typing* (*st s*) = *hrs-htd* (*t-hrs s*)
assumes *heap-typing-upd-write-commute*:
$\bigwedge p\ f\ h\ d.$ *heap-typing-upd d* ($w\ p\ f\ h$) = $w\ p\ f$ (*heap-typing-upd d h*)
assumes *ptr-valid-imp-v*: $\bigwedge p\ h.$ *ptr-valid* (*heap-typing h*) $p \Longrightarrow v\ h\ p$
assumes *sim-stack-stack-byte-zero'*:
$\bigwedge p\ s.\ \forall\ a \in$ *ptr-span p*. *root-ptr-valid* (*hrs-htd* (*t-hrs s*)) (*PTR*(*stack-byte*) *a*) $\Longrightarrow$
$r\ (st\ s)\ p = ZERO('a)$
begin

lemma *heap-typing-write*[*simp*]: *heap-typing* ($w\ p\ f\ h$) = *heap-typing h*

⟨*proof*⟩

lemma *heap-typing-upd*[*simp*]: *r* (*heap-typing-upd d h*) = *r h*
⟨*proof*⟩

lemma *unchanged-typing-root-ptr-valid-preserved*:
unchanged-typing-on A t1 t2 ⟹ *ptr-span p* ⊆ *A*
⟹ *root-ptr-valid* (*heap-typing t1*) *p* = *root-ptr-valid* (*heap-typing t2*) *p*
⟨*proof*⟩

lemma *unchanged-typing-ptr-valid-preserved*:
assumes *unch*: *unchanged-typing-on UNIV t1 t2*
shows *ptr-valid* (*heap-typing t1*) *p* = *ptr-valid* (*heap-typing t2*) *p*
⟨*proof*⟩

lemma *unchanged-typing-on-write* [*unchanged-typing-on-simps*]:
unchanged-typing-on A t (*w p f t*)
⟨*proof*⟩

lemma *heap-typing-fold-upd-write*: *heap-typing* (*fold* (λ*i*. *w* (*x i*) (*g i*)) [*0*..<*n*] *t*)
= *heap-typing t*
⟨*proof*⟩

end

lemma *distinct-prop-conj*:
distinct-prop (λ*a b*. *R a b* ∧ *P a b*) *xs* ⟷ *distinct-prop R xs* ∧ *distinct-prop P xs*
⟨*proof*⟩

lemma *distinct-prop-zip-cons*:
list-all (λ(*c*, *d*). *P a b c d*) (*zip as bs*) ⟹
distinct-prop (λ(*a*, *b*) (*c*, *d*). *P a b c d*) (*zip as bs*) ⟹
distinct-prop (λ(*a*, *b*) (*c*, *d*). *P a b c d*) (*zip* (*a*#*as*) (*b*#*bs*))
⟨*proof*⟩

lemma *distinct-prop-zip-nil*:
distinct-prop (λ(*a*, *b*) (*c*, *d*). *P a b c d*) (*zip* [] [])
⟨*proof*⟩

lemma *list-all-zip-cons*:
P a b ⟹ *list-all* (λ(*a*, *b*). *P a b*) (*zip as bs*) ⟹
list-all (λ(*a*, *b*). *P a b*) (*zip* (*a*#*as*) (*b*#*bs*))
⟨*proof*⟩

lemma *list-all-zip-nil*:
list-all (λ(*a*, *b*). *P a b*) (*zip* [] [])
⟨*proof*⟩

named-theorems *disjnt-heap-fields-comp*

context *open-types*
begin

lemma *typ-heap-simulationI-part-addressable*:
fixes $R :: 't \Rightarrow 'a::\{xmem\text{-}type, stack\text{-}type\}\ ptr \Rightarrow 'a$
assumes *hrs*: *heap-typing-simulation* $\mathcal{T}$ *hrs hrs-upd heap-typing heap-typing-upd l*
assumes *fs*: *map-of* $\mathcal{T}$ (*typ-uinfo-t TYPE*('*a*)) = *Some fs*
assumes *lense g u*
assumes *len*[*simp*]: *length rs* = *length fs length ws* = *length fs*
assumes ∗:
list-all (λ(*f*, *r*, *w*). *typ-heap-simulation-of-field l hrs hrs-upd heap-typing-upd f r w*) (*zip fs* (*zip rs ws*))
and ∗∗: *distinct-prop* (λ(*f1*, *w1*) (*f2*, *w2*).
disj-fn f1 f2 ⟶ *pointer-writer-disjnt-eq w1 w2*)
(*zip fs ws*)
and *ws-u*: *list-all* (λ*w*. ∀ *p a f*. *w p a* ∘ *u f* = *u f* ∘ *w p a*) *ws*
and *l-u*: ⋀(*p*::'*a ptr*) (*x*::'*a*) (*s*::'*b*).
PTR-VALID('*a*) (*hrs-htd* (*hrs s*)) *p* ⟹
l (*hrs-upd* (*write-dedicated-heap p x*) *s*) = *u* (*upd-fun p* (λ*old*. *merge-addressable-fields old x*)) (*l s*)
and *r-stack*:
⋀*p s*. ∀ *a*∈*ptr-span p*. *root-ptr-valid* (*hrs-htd* (*hrs s*)) (*PTR*(*stack-byte*) *a*) ⟹
g (*l s*) *p* = *ZERO*(-)
and *heap-typing-u*: ⋀*x d h*. *heap-typing-upd d* (*u x h*) = *u x* (*heap-typing-upd d h*)
assumes *V*:
⋀*h p*. *V h p* ⟷ *PTR-VALID*('*a*) (*heap-typing h*) *p*
assumes *R*:
⋀*h p*. *R h p* = *fold* (λ*r*. *r h p*) *rs* (*g h p*)
assumes *W*:
⋀*p f h*. *W p f h* =
fold (λ*w*. *w p* (*f* (*R h p*))) *ws* (*u* (*upd-fun p* (λ*old*. *merge-addressable-fields old* (*f* (*R h p*)))) *h*)
shows *typ-heap-simulation-open-types* $\mathcal{T}$ *l R W V hrs hrs-upd heap-typing heap-typing-upd*
∧
(*pointer-lense g* (λ*p f*. *u* (*upd-fun p f*))) ∧
(∀ *w*. (∀ *x*. *list-all* (λ*w'*. *pointer-writer-disjnt* (λ*p*. *w' p x*) *w*) *ws*) ⟶
(∀ *x*. *pointer-writer-disjnt* (λ*p*. *u* (*upd-fun p* (λ-. *x*))) *w*) ⟶
(∀ *f*. *pointer-writer-disjnt* (λ*p*. *W p f*) *w*)) ∧
(∀ *w p*. (∀ *x*. *list-all* (λ*w'*. *w' p x o w* = *w o w' p x*) *ws*) ⟶
(∀ *x*. *u* (*upd-fun p* (λ-. *x*)) *o w* = *w o u* (*upd-fun p* (λ-. *x*))) ⟶
(∀ *f*. *W p f o w* = *w o W p f*))
(**is** *?t1* ∧ *?t3* ∧
(∀ *w*. *?ws w* ⟶ *?u w* ⟶ (∀ *f*. *?t2 f w*)) ∧
(∀ *w p*. *?ws2 w p* ⟶ *?u2 w p* ⟶ (∀ *f*. *?t4 w p f*)))

⟨*proof*⟩

lemma *typ-heap-simulationI-no-addressable*:
fixes $R :: 't \Rightarrow 'a::\{xmem\text{-}type, stack\text{-}type\}\ ptr \Rightarrow 'a$
assumes *heap-typing-simulation* $\mathcal{T}$ *hrs hrs-upd heap-typing heap-typing-upd l*
assumes *R-u*: *lense R u*
assumes *fs*: *map-of* $\mathcal{T}$ (*typ-uinfo-t TYPE*('*a*)) = *None*
and *l-u*: $\bigwedge(p::'a\ ptr)\ (x::'a)\ (s::'b)$.
PTR-VALID('*a*) (*hrs-htd* (*hrs s*)) *p* $\Longrightarrow$
l (*hrs-upd* (*write-dedicated-heap p x*) *s*) = *u* (*upd-fun p* (λ*old*. *merge-addressable-fields old x*)) (*l s*)
and *heap-typing-u*: $\bigwedge x\ d\ h$. *heap-typing-upd d* (*u x h*) = *u x* (*heap-typing-upd d h*)
and *r-stack*:
$\bigwedge p\ s.\ \forall\, a{\in}$*ptr-span p*. *root-ptr-valid* (*hrs-htd* (*hrs s*)) (*PTR*(*stack-byte*) *a*) $\Longrightarrow$
R (*l s*) *p* = *ZERO*(-)
assumes *V*:
$\bigwedge h\ p$. *V h p* $\longleftrightarrow$ *PTR-VALID*('*a*) (*heap-typing h*) *p*
assumes *W*:
$\bigwedge p\ f\ h$. *W p f h* = *u* (λ*h'*. *h'*(*p* := *f* (*h' p*))) *h*
shows *typ-heap-simulation-open-types* $\mathcal{T}$ *l R W V hrs hrs-upd heap-typing heap-typing-upd*
⟨*proof*⟩

lemma *typ-heap-simulationI-all-addressable*:
fixes $R :: 't \Rightarrow 'a::\{xmem\text{-}type, stack\text{-}type\}\ ptr \Rightarrow 'a$
assumes *hrs*: *heap-typing-simulation* $\mathcal{T}$ *hrs hrs-upd heap-typing heap-typing-upd l*
assumes *fs*: *map-of* $\mathcal{T}$ (*typ-uinfo-t TYPE*('*a*)) = *Some fs*
assumes *len*[*simp*]: *length rs* = *length fs length ws* = *length fs*
assumes ∗:
list-all (λ(*f*, *r*, *w*). *typ-heap-simulation-of-field l hrs hrs-upd heap-typing-upd f r w*) (*zip fs* (*zip rs ws*))
and ∗∗: *distinct-prop* (λ(*f1*, *w1*) (*f2*, *w2*).
disj-fn f1 f2 ⟶ *pointer-writer-disjnt-eq w1 w2*)
(*zip fs ws*)
assumes *all*: $\bigwedge a\ b$. *fold* (λ*x*. *merge-ti* (*the* (*field-ti TYPE*('*a*) *x*)) *a*) *fs b* = *a*
assumes *V*:
$\bigwedge h\ p$. *V h p* $\longleftrightarrow$ *PTR-VALID*('*a*) (*heap-typing h*) *p*
assumes *R*:
$\bigwedge h\ p\ x$. *R h p* = *fold* (λ*r*. *r h p*) *rs x*
assumes *W*:
$\bigwedge p\ f\ h$. *W p f h* = *fold* (λ*w*. *w p* (*f* (*R h p*))) *ws h*
shows *typ-heap-simulation-open-types* $\mathcal{T}$ *l R W V hrs hrs-upd heap-typing heap-typing-upd*
∧
(∀ *w*. (∀ *x*. *list-all* (λ*w'*. *pointer-writer-disjnt* (λ*p*. *w' p x*) *w*) *ws*) ⟶
(∀ *f*. *pointer-writer-disjnt* (λ*p*. *W p f*) *w*)) ∧
(∀ (*w*::'*t* ⇒ '*t*) *p*.
(∀ *x*. *list-all* (λ*w'*. *w' p x* ∘ *w* = *w* ∘ *w' p x*) *ws*) ⟶
(∀ *f*. *W p f* ∘ *w* = *w* ∘ *W p f*))

(**is** *?t1* ∧ (∀ *w*. *?ws w* ⟶ (∀ *f*. *?t2 f w*)) ∧ (∀ *w p*. *?ws2 p w* ⟶ (∀ *f*. *?t3 p f w*)))
⟨*proof*⟩

end

definition
field-with-lense ::
qualified-field-name ⇒ ('*a*::*c-type* ⇒ '*b*::*c-type*) ⇒ (('*b* ⇒ '*b*) ⇒ '*a* ⇒ '*a*) ⇒ *bool*
where
field-with-lense f g u ⟷
field-ti TYPE('*a*) *f* = *Some* (*adjust-ti* (*typ-info-t TYPE*('*b*)) *g* (*u o* (λ*x* -. *x*)))
∧
lense g u

lemma *update-desc-id*: *update-desc* (λ*x*. *x*) (λ*x* -. *x*) = *id*
⟨*proof*⟩

lemma *field-with-lense-Nil*: *field-with-lense* [] (λ*x*. *x*) (λ*f x*. *f x*)
⟨*proof*⟩

lemma *field-with-lense-Cons*:
fixes *g* :: '*a*::*mem-type* ⇒ '*b*::*mem-type* **and** *gs* :: '*b* ⇒ '*c*::*mem-type*
assumes *fs*: *field-with-lense fs gs us*
assumes *f*: *field-ti TYPE*('*a*) [*f*] = *Some* (*adjust-ti* (*typ-info-t TYPE*('*b*)) *g* (*u o* (λ*x* -. *x*)))
assumes *g-u*: *fg-cons g* (*u o* (λ*x* -. *x*))
assumes *u*: ⋀*f x*. *u f x* = *u* (λ-. *f* (*g x*)) *x*
shows *field-with-lense* (*f* # *fs*) (λ*x*. *gs* (*g x*)) (λ*f*. *u* (*us f*))
⟨*proof*⟩

lemma (**in** *typ-heap-simulation-open-types*) *typ-heap-simulation-of-field*:
assumes *f*: *field-with-lense f g u*
assumes *v*: ⋀*h p*. *PTR-VALID*('*a*) (*hrs-htd* (*t-hrs h*)) *p* ⟹ *v* (*st h*) *p*
assumes *g-zero*: *g ZERO*('*x*::*mem-type*) = *ZERO*('*a*)
shows *typ-heap-simulation-of-field st t-hrs t-hrs-update heap-typing-upd f*
(λ*h p*. *u* (λ-. *r h* (*PTR*('*a*) &(*p*→*f*))))
(λ*p x*. *w* (*PTR*('*a*) &(*p*→*f*)) (λ-. *g x*))
⟨*proof*⟩

context *pointer-array-lense*
begin

lemma *pointer-writer-disjnt-heap-array-map*[*pointer-writer-disjnt-intros*]:
assumes *w'*: ⋀*x*. *pointer-writer-disjnt* (λ*p*. *w p x*) *w'*
shows *pointer-writer-disjnt* (λ*p*. *heap-array-map p x*) *w'*
⟨*proof*⟩

lemma *pointer-writer-disjnt-heap-array-map-right*[*pointer-writer-disjnt-intros*]:
(⋀*x*. *pointer-writer-disjnt w'* (λ*p*. *w p x*)) ⟹ *pointer-writer-disjnt w'* (λ*p*. *heap-array-map p x*)
⟨*proof*⟩

lemma *disjnt-comp-heap-array-map*[*disjnt-heap-fields-comp*]:
assumes *w'*: ⋀*q x*. *w q x* ∘ *w'* = *w'* ∘ *w q x*
shows *heap-array-map p x* ∘ *w'* = *w'* ∘ *heap-array-map p x*
⟨*proof*⟩

end

lemma (**in** *open-types*) *valid-array-of-ptr-valid-arrayI*:
fixes *r* :: *'t* ⇒ *'a*::{*xmem-type*, *array-outer-max-size*} *ptr* ⇒ *'a*
fixes *p* :: (*'a*['*n*::{*finite*, *array-max-count*}]) *ptr*
assumes *f*: *map-of* 𝒯 (*typ-uinfo-t TYPE*(*'a*['*n*])) = *Some* (*array-fields CARD*(*'n*))
assumes *p*: *PTR-VALID*(*'a*['*n*]) *h'* *p*
assumes *v*: ⋀*p*. *PTR-VALID*(*'a*) *h'* *p* ⟹ *v h p*
shows *valid-array-base.valid-array v h p*
⟨*proof*⟩

lemma (**in** *open-types*) *valid-array-of-ptr-valid-array1*[*simp*]:
fixes *r* :: *'t* ⇒ *'a*::{*xmem-type*, *array-outer-max-size*} *ptr* ⇒ *'a*
fixes *p* :: (*'a*['*n*::{*finite*, *array-max-count*}]) *ptr*
assumes *f*:
map-of 𝒯 (*typ-uinfo-t TYPE*(*'a*['*n*])) = *Some* (*array-fields CARD*(*'n*))
assumes *p*: *PTR-VALID*(*'a*['*n*]) (*heap-typing h*) *p*
shows *valid-array-base.valid-array* (λ*h*. *PTR-VALID*(*'a*) (*heap-typing h*)) *h p*
⟨*proof*⟩

lemma (**in** *open-types*) *valid-array-of-ptr-valid-array2*[*simp*]:
fixes *r* :: *'t* ⇒ *'a*::{*xmem-type*, *array-inner-max-size*} *ptr* ⇒ *'a*
fixes *p* :: (*'a*['*n*::{*finite*, *array-max-count*}]['*m*::{*finite*, *array-max-count*}]) *ptr*
assumes *f2*:
map-of 𝒯 (*typ-uinfo-t TYPE*(*'a*['*n*]['*m*])) = *Some* (*array-fields CARD*(*'m*))
assumes *f1*:
map-of 𝒯 (*typ-uinfo-t TYPE*(*'a*['*n*])) = *Some* (*array-fields CARD*(*'n*))
assumes *p*: *PTR-VALID*(*'a*['*n*]['*m*]) (*heap-typing h*) *p*
shows *valid-array-base.valid-array* (*valid-array-base.valid-array*
(λ*h*. *PTR-VALID*(*'a*) (*heap-typing h*))) *h p*
⟨*proof*⟩

lemma (**in** *open-types*) *ptr-valid-unsigned*[*simp*]:
PTR-VALID(*'a*::*len8 signed word*) *h p* ⟷
PTR-VALID(*'a*::*len8 word*) *h* (*PTR-COERCE*(*'a signed word* → *'a word*) *p*)
⟨*proof*⟩

lemma *ucast-zero-word*:
UCAST(*'a*::*len8* → *'a signed*) *ZERO*(*'a word*) = *ZERO*(*'a signed word*)

⟨*proof*⟩

definition *signed-heap*::
(*'s* ⇒ *'a*::*len word ptr* ⇒ *'a word*) ⇒ (*'s* ⇒ *'a signed word ptr* ⇒ *'a signed word*)
where
signed-heap h s = *UCAST* (*'a*::*len* → *'a signed*) *o* (*h s*) *o PTR-COERCE*(*'a signed word* → *'a word*)

lemma *signed-heap-apply*[*simp*]:
signed-heap h s p = *UCAST* (*'a*::*len* → *'a signed*) (*h s* (*PTR-COERCE*(*'a signed word* → *'a word*) *p*))
⟨*proof*⟩

lemma *signed-typ-heap-simulation-of-typ-heap-simulation*:
fixes *r* :: - ⇒ - ⇒ *'a*::*len8 word*
assumes *r*:
typ-heap-simulation-open-types T *st r w v t-hrs t-hrs-update heap-typing heap-typing-upd*
shows *typ-heap-simulation-open-types* T *st*
(*signed-heap r*)
(λ*p m*. *w* (*PTR-COERCE*(*'a signed word* → *'a word*) *p*) (λ*w*. *UCAST*(*'a signed* → *'a*) (*m* (*UCAST*(*'a* → *'a signed*) *w*))))
(λ*h p*. *v h* (*PTR-COERCE*(*'a signed word* → *'a word*) *p*))
t-hrs t-hrs-update heap-typing heap-typing-upd
⟨*proof*⟩

lemma *array-typ-heap-simulation-of-typ-heap-simulation*:
fixes *r* :: - ⇒ - ⇒ *'a*::{*stack-type*, *xmem-type*, *array-outer-max-size*}
assumes *typ-heap-simulation-open-types* T *st r w v t-hrs t-hrs-update heap-typing heap-typing-upd*
assumes *f*: *map-of* T (*typ-uinfo-t TYPE*(*'a*['*n*::{*finite*, *array-max-count*}])) =
Some (*array-fields CARD*(*'n*))
shows
typ-heap-simulation-open-types T *st*
(*pointer-array-lense.heap-array r* :: - ⇒ (*'a*['*n*]) *ptr* ⇒ -)
(*pointer-array-lense.heap-array-map r w*)
(*valid-array-base.valid-array v*)
t-hrs t-hrs-update heap-typing heap-typing-upd
⟨*proof*⟩

lemma (**in** *typ-heap-simulation-open-types*) *stack-simulation-heap-typingI*:
assumes *hrs*: *heap-typing-simulation* T *t-hrs t-hrs-update heap-typing heap-typing-upd st*
assumes *sim-stack-alloc-heap-typing*:
⋀*p d s n*.
(*p*, *d*) ∈ *stack-allocs n* S *TYPE*(*'a*) (*hrs-htd* (*t-hrs s*)) ⟹
st (*t-hrs-update* (*hrs-mem-update* (*fold* (λ*i*. *heap-update* (*p* +$_p$ *int i*) *c-type-class.zero*) [*0*..<*n*]) ∘ *hrs-htd-update* (λ-. *d*)) *s*) =
(*heap-typing-upd* (λ-. *d*) (*st s*))
assumes *sim-stack-release-heap-typing*:

$\bigwedge(p::'a\ ptr)\ s\ n.\ (\bigwedge i.\ i < n \Longrightarrow root\text{-}ptr\text{-}valid\ (hrs\text{-}htd\ (t\text{-}hrs\ s))\ (p +_p int\ i))$
$\Longrightarrow$
$st\ (t\text{-}hrs\text{-}update\ (hrs\text{-}htd\text{-}update\ (stack\text{-}releases\ n\ p))\ s) =$
$heap\text{-}typing\text{-}upd\ (stack\text{-}releases\ n\ p)$
$(st\ (t\text{-}hrs\text{-}update\ (hrs\text{-}mem\text{-}update\ (fold\ (\lambda i.\ heap\text{-}update\ (p +_p int\ i)$
$c\text{-}type\text{-}class.zero)\ [0..<n]))\ s))$
shows *stack-simulation-heap-typing st r w t-hrs t-hrs-update heap-typing heap-typing-upd*
$\mathcal{S}\ \mathcal{T}$
⟨*proof*⟩

end

theory *NatBitwise*
imports
More-Lib
begin

lemma *lsb-nat-def*:
‹*lsb n* = *lsb* (*int n*)›
⟨*proof*⟩

instantiation *nat* :: *msb*
begin

definition
msb x = *msb* (*int x*)

instance ⟨*proof*⟩

end

lemma *not-msb-nat*:
‹¬ *msb n*› **for** *n* :: *nat*
⟨*proof*⟩

lemma *set-bit-nat-def*:
‹*set-bit x y z* = *nat* (*set-bit* (*int x*) *y z*)›
⟨*proof*⟩

lemma *nat-2p-eq-shiftl*:
(*2*::*nat*)^*x* = *1* << *x*
⟨*proof*⟩

lemma *shiftl-nat-def*:
(*x*::*nat*) << *y* = *nat* (*int x* << *y*)
⟨*proof*⟩

lemma *nat-shiftl-less-cancel*:
$n \leq m \Longrightarrow ((x :: nat) << n < y << m) = (x < y << (m - n))$
⟨*proof*⟩

lemma *nat-shiftl-lt-2p-bits*:
$(x::nat) < 1 << n \Longrightarrow \forall\, i \geq n.\ \neg\ x\ !!\ i$
⟨*proof*⟩

lemmas *nat-eq-test-bit* = *bit-eq-iff*
lemmas *nat-eq-test-bitI* = *bit-eq-iff*[*THEN iffD2*, *rule-format*]

end

Chapter 22

WA phase: Word Abstraction

theory *WordAbstract*
imports
L2Peephole
NatBitwise
begin

22.1 Basic Definitions

definition *WORD-MAX* $x \equiv ((2 \hat{} (\textit{len-of}\ x - 1) - 1) :: \textit{int})$
definition *WORD-MIN* $x \equiv (- (2 \hat{} (\textit{len-of}\ x - 1)) :: \textit{int})$
definition *UWORD-MAX* $x \equiv ((2 \hat{} (\textit{len-of}\ x)) - 1 :: \textit{nat})$

lemma *WORD-values* [*simplified*]:
$\textit{WORD-MAX}\ (\textit{TYPE}(8\ \textit{signed})) = (2 \hat{}\ 7 - 1)$
$\textit{WORD-MAX}\ (\textit{TYPE}(16\ \textit{signed})) = (2 \hat{}\ 15 - 1)$
$\textit{WORD-MAX}\ (\textit{TYPE}(32\ \textit{signed})) = (2 \hat{}\ 31 - 1)$
$\textit{WORD-MAX}\ (\textit{TYPE}(64\ \textit{signed})) = (2 \hat{}\ 63 - 1)$

$\textit{WORD-MIN}\ (\textit{TYPE}(8\ \textit{signed})) = - (2 \hat{}\ 7)$
$\textit{WORD-MIN}\ (\textit{TYPE}(16\ \textit{signed})) = - (2 \hat{}\ 15)$
$\textit{WORD-MIN}\ (\textit{TYPE}(32\ \textit{signed})) = - (2 \hat{}\ 31)$
$\textit{WORD-MIN}\ (\textit{TYPE}(64\ \textit{signed})) = - (2 \hat{}\ 63)$

$\textit{UWORD-MAX}\ (\textit{TYPE}(8)) = (2 \hat{}\ 8 - 1)$
$\textit{UWORD-MAX}\ (\textit{TYPE}(16)) = (2 \hat{}\ 16 - 1)$
$\textit{UWORD-MAX}\ (\textit{TYPE}(32)) = (2 \hat{}\ 32 - 1)$
$\textit{UWORD-MAX}\ (\textit{TYPE}(64)) = (2 \hat{}\ 64 - 1)$
⟨*proof*⟩

lemmas *WORD-values-add1* =
WORD-values [*THEN arg-cong* [**where** $f{=}\lambda x.\ x + 1$],
simplified semiring-norm, *simplified numeral-One*]

lemmas *WORD-values-minus1* =

WORD-values [*THEN arg-cong* [**where** $f{=}\lambda x.\ x - 1$],
simplified semiring-norm, *simplified numeral-One nat-numeral*]

lemmas *WORD-values-fold* [*L1unfold*] =
WORD-values [*symmetric*]
WORD-values-add1 [*symmetric*]
WORD-values-minus1 [*symmetric*]

lemma *WORD-signed-to-unsigned* [*simp*]:
$WORD\text{-}MAX\ TYPE('a\ signed) = WORD\text{-}MAX\ TYPE('a{::}len)$
$WORD\text{-}MIN\ TYPE('a\ signed) = WORD\text{-}MIN\ TYPE('a{::}len)$
$UWORD\text{-}MAX\ TYPE('a\ signed) = UWORD\text{-}MAX\ TYPE('a{::}len)$
⟨*proof*⟩

lemma *INT-MIN-comparisons* [*simp*]:
$[\![\ a \leq -\ (2\ \hat{}\ (len\text{-}of\ TYPE('a) - 1))\ ]\!] \Longrightarrow a \leq WORD\text{-}MIN\ (TYPE('a{::}len))$
$a < -\ (2\ \hat{}\ (len\text{-}of\ TYPE('a) - 1)) \Longrightarrow a < WORD\text{-}MIN\ (TYPE('a{::}len))$
$a \geq -\ (2\ \hat{}\ (len\text{-}of\ TYPE('a) - 1)) \Longrightarrow a \geq WORD\text{-}MIN\ (TYPE('a{::}len))$
$a > -\ (2\ \hat{}\ (len\text{-}of\ TYPE('a) - 1)) \Longrightarrow a \geq WORD\text{-}MIN\ (TYPE('a{::}len))$
⟨*proof*⟩

lemma *INT-MAX-comparisons* [*simp*]:
$a \leq (2\ \hat{}\ (len\text{-}of\ TYPE('a) - 1)) - 1 \Longrightarrow a \leq WORD\text{-}MAX\ (TYPE('a{::}len))$
$a < (2\ \hat{}\ (len\text{-}of\ TYPE('a) - 1)) - 1 \Longrightarrow a < WORD\text{-}MAX\ (TYPE('a{::}len))$
$a \geq (2\ \hat{}\ (len\text{-}of\ TYPE('a) - 1)) - 1 \Longrightarrow a \geq WORD\text{-}MAX\ (TYPE('a{::}len))$
$a > (2\ \hat{}\ (len\text{-}of\ TYPE('a) - 1)) - 1 \Longrightarrow a \geq WORD\text{-}MAX\ (TYPE('a{::}len))$
⟨*proof*⟩

lemma *UINT-MAX-comparisons* [*simp*]:
$x \leq (2\ \hat{}\ (len\text{-}of\ TYPE('a))) - 1 \Longrightarrow x \leq UWORD\text{-}MAX\ (TYPE('a{::}len))$
$x < (2\ \hat{}\ (len\text{-}of\ TYPE('a))) - 1 \Longrightarrow x \leq UWORD\text{-}MAX\ (TYPE('a{::}len))$
$x \geq (2\ \hat{}\ (len\text{-}of\ TYPE('a))) - 1 \Longrightarrow x \geq UWORD\text{-}MAX\ (TYPE('a{::}len))$
$x > (2\ \hat{}\ (len\text{-}of\ TYPE('a))) - 1 \Longrightarrow x > UWORD\text{-}MAX\ (TYPE('a{::}len))$
⟨*proof*⟩

lemma *is-up-SCAST-same-signed* [*simp*]: *is-up* $(SCAST\ (('a{::}len) \to 'a\ signed))$
⟨*proof*⟩

lemma *sint-ucast-signed* [*simp*, *L2opt*]:*sint* $(UCAST\ ('a{::}len \to 'a\ signed)\ x) =$ *sint x*
⟨*proof*⟩

22.2 Abstracting values and expressions

definition *introduce-typ-abs-fn f* ≡ *True*

declare *introduce-typ-abs-fn-def* [*simp*]

lemma *introduce-typ-abs-fn*:
introduce-typ-abs-fn f
⟨*proof*⟩

definition
abstract-bool-binop :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ ('*c* ⇒ '*a*)
⇒ ('*a* ⇒ '*a* ⇒ *bool*) ⇒ ('*c* ⇒ '*c* ⇒ *bool*) ⇒ *bool*
where
abstract-bool-binop P f X X' ≡ ∀ *a b*. *P* (*f a*) (*f b*) ⟶ (*X' a b* = *X* (*f a*) (*f b*))

definition
abstract-binop :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ ('*c* ⇒ '*a*)
⇒ ('*a* ⇒ '*a* ⇒ '*a*) ⇒ ('*c* ⇒ '*c* ⇒ '*c*) ⇒ *bool*
where
abstract-binop P f X X' ≡ ∀ *a b*. *P* (*f a*) (*f b*) ⟶ (*f* (*X' a b*) = *X* (*f a*) (*f b*))

definition *abstract-val P a f b* ≡ *P* ⟶ (*a* = *f b*)

definition *abs-var a f b* ≡ *abstract-val True a f b*

lemmas *basic-abstract-defs* =
abstract-bool-binop-def
abstract-binop-def
abstract-val-def
abs-var-def

lemma *abstract-val-trivial*:
abstract-val True (*f b*) *f b*
⟨*proof*⟩

lemma *abstract-binop-is-abstract-val*:
abstract-binop P f X X' = (∀ *a b*. *abstract-val* (*P* (*f a*) (*f b*)) (*X* (*f a*) (*f b*)) *f* (*X' a b*))
⟨*proof*⟩

lemma *abstract-expr-bool-binop*:
⟦ *abstract-bool-binop E f X X'*;
introduce-typ-abs-fn f;
abstract-val P a f a';
abstract-val Q b f b' ⟧ ⟹
abstract-val (*P* ∧ *Q* ∧ *E a b*) (*X a b*) *id* (*X' a' b'*)
⟨*proof*⟩

lemma *abstract-expr-binop*:
⟦ *abstract-binop E f X X'*;
abstract-val P a f a';
abstract-val Q b f b' ⟧ ⟹
abstract-val (*P* ∧ *Q* ∧ *E a b*) (*X a b*) *f* (*X' a' b'*)
⟨*proof*⟩

lemma *unat-abstract-bool-binops*:
abstract-bool-binop (λ- -. *True*) (*unat* :: ('*a*::*len*) *word* ⇒ *nat*) (<) (<)
abstract-bool-binop (λ- -. *True*) (*unat* :: ('*a*::*len*) *word* ⇒ *nat*) (≤) (≤)
abstract-bool-binop (λ- -. *True*) (*unat* :: ('*a*::*len*) *word* ⇒ *nat*) (=) (=)
⟨*proof*⟩

lemmas *unat-mult-simple* = *iffD1* [*OF unat-mult-lem* [*unfolded word-bits-len-of*]]

lemma *le-to-less-plus-one*:
((*a*::*nat*) ≤ *b*) = (*a* < *b* + *1*)
⟨*proof*⟩

lemma *unat-abstract-binops*:
abstract-binop (λ*a b*. *a* + *b* ≤ *UWORD-MAX TYPE*('*a*::*len*)) (*unat* :: '*a word* ⇒ *nat*) (+) (+)
abstract-binop (λ*a b*. *a* ∗ *b* ≤ *UWORD-MAX TYPE*('*a*)) (*unat* :: '*a word* ⇒ *nat*) (∗) (∗)
abstract-binop (λ*a b*. *a* ≥ *b*) (*unat* :: '*a word* ⇒ *nat*) (−) (−)
abstract-binop (λ*a b*. *True*) (*unat* :: '*a word* ⇒ *nat*) (*div*) (*div*)
abstract-binop (λ*a b*. *True*) (*unat* :: '*a word* ⇒ *nat*) (*mod*) (*mod*)
⟨*proof*⟩

lemma *unat-of-int*:
⟦*i* ≥ *0*; *i* < *2* ^ *LENGTH*('*a*)⟧ ⟹ *unat* (*of-int i* :: '*a*::*len word*) = *nat i*
⟨*proof*⟩

lemma *unat-of-int-signed*:
⟦*i* ≥ *0*; *i* < *2* ^ *LENGTH*('*a*)⟧ ⟹ *unat* (*of-int i* :: '*a*::*len signed word*) = *nat i*
⟨*proof*⟩

lemma *nat-sint*:
0 <=*s* (*x* :: '*a*::*len signed word*) ⟹ *nat* (*sint x*) = *unat x*
⟨*proof*⟩

lemma *int-unat-nonneg*:
0 <=*s* (*x* :: '*a*::*len signed word*) ⟹ *int* (*unat x*) = *sint x*
⟨*proof*⟩

lemma *uint-sint-nonneg*:
0 <=*s* (*x* :: '*a*::*len signed word*) ⟹ *uint x* = *sint x*
⟨*proof*⟩

lemma *unat-bitwise-abstract-binops*:
abstract-binop (λ*a b*. *True*) (*unat* :: *'a*::*len word* ⇒ *nat*) *Bit-Operations.and Bit-Operations.and*
abstract-binop (λ*a b*. *True*) (*unat* :: *'a*::*len word* ⇒ *nat*) *Bit-Operations.or Bit-Operations.or*
abstract-binop (λ*a b*. *True*) (*unat* :: *'a*::*len word* ⇒ *nat*) *Bit-Operations.xor Bit-Operations.xor*
⟨*proof*⟩

lemma *abstract-val-unsigned-bitNOT*:
abstract-val P x unat (*x'* :: *'a*::*len word*) ⟹
abstract-val P (*UWORD-MAX TYPE*(*'a*) − *x*) *unat* (*NOT x'*)
⟨*proof*⟩

lemma *snat-abstract-bool-binops*:
abstract-bool-binop (λ- -. *True*) (*sint* :: (*'a*::*len*) *signed word* ⇒ *int*) (<) (*word-sless*)
abstract-bool-binop (λ- -. *True*) (*sint* :: *'a signed word* ⇒ *int*) (≤) (*word-sle*)
abstract-bool-binop (λ- -. *True*) (*sint* :: *'a signed word* ⇒ *int*) (=) (=)
⟨*proof*⟩

lemma *snat-abstract-binops*:
abstract-binop (λ*a b*. *WORD-MIN TYPE*(*'a*::*len*) ≤ *a* + *b* ∧ *a* + *b* ≤ *WORD-MAX TYPE*(*'a*)) (*sint* :: *'a signed word* ⇒ *int*) (+) (+)
abstract-binop (λ*a b*. *WORD-MIN TYPE*(*'a*) ≤ *a* ∗ *b* ∧ *a* ∗ *b* ≤ *WORD-MAX TYPE*(*'a*)) (*sint* :: *'a signed word* ⇒ *int*) (∗) (∗)
abstract-binop (λ*a b*. *WORD-MIN TYPE*(*'a*) ≤ *a* − *b* ∧ *a* − *b* ≤ *WORD-MAX TYPE*(*'a*)) (*sint* :: *'a signed word* ⇒ *int*) (−) (−)
abstract-binop (λ*a b*. *WORD-MIN TYPE*(*'a*) ≤ *a sdiv b* ∧ *a sdiv b* ≤ *WORD-MAX TYPE*(*'a*)) (*sint* :: *'a signed word* ⇒ *int*) (*sdiv*) (*sdiv*)
abstract-binop (λ*a b*. *WORD-MIN TYPE*(*'a*) ≤ *a smod b* ∧ *a smod b* ≤ *WORD-MAX TYPE*(*'a*)) (*sint* :: *'a signed word* ⇒ *int*) (*smod*) (*smod*)
⟨*proof*⟩

lemma *sint-bitwise-abstract-binops*:
abstract-binop (λ*a b*. *True*) (*sint* :: *'a*::*len signed word* ⇒ *int*) *Bit-Operations.and Bit-Operations.and*
abstract-binop (λ*a b*. *True*) (*sint* :: *'a*::*len signed word* ⇒ *int*) *Bit-Operations.or Bit-Operations.or*
abstract-binop (λ*a b*. *True*) (*sint* :: *'a*::*len signed word* ⇒ *int*) *Bit-Operations.xor Bit-Operations.xor*
⟨*proof*⟩

lemma *abstract-val-signed-bitNOT*:
abstract-val P x sint (*x'* :: *'a*::*len signed word*) ⟹
abstract-val P (*NOT x*) *sint* (*NOT x'*)
⟨*proof*⟩

lemma *abstract-val-signed-unary-minus*:

⟦ *abstract-val P r sint r′* ⟧ ⟹
abstract-val (*P* ∧ (− *r*) ≤ *WORD-MAX TYPE*('*a*)) (− *r*) *sint* (− (*r′* :: ('*a* :: *len*) *signed word*))
⟨*proof*⟩

lemma *bang-big-nonneg*:
⟦ *0* <=*s* (*x*::'*a*::*len signed word*); *n* ≥ *size x* − *1* ⟧ ⟹ (*x* !! *n*) = *False*
⟨*proof*⟩

lemma *int-shiftr-nth*[*simp*]:
(*i* >> *n*) !! *m* = *i* !! (*n* + *m*) **for** *i* :: *int*
⟨*proof*⟩

lemma *int-shiftl-nth*[*simp*]:
(*i* << *n*) !! *m* = (*n* ≤ *m* ∧ *i* !! (*m* − *n*)) **for** *i* :: *int*
⟨*proof*⟩

lemma *sint-shiftr-nonneg*:
⟦ *0* <=*s* (*x* :: '*a*::*len signed word*); *0* ≤ *n*; *n* < *LENGTH*('*a*) ⟧ ⟹ *sint* (*x* >> *n*) = *sint x* >> *n*
⟨*proof*⟩

lemma *abstract-val-unsigned-unary-minus*:
⟦ *abstract-val P r unat r′* ⟧ ⟹
abstract-val P (*if r* = *0 then 0 else UWORD-MAX TYPE*('*a*::*len*) + *1* − *r*) *unat* (− (*r′* :: '*a word*))
⟨*proof*⟩

lemma *abstract-val-signed-shiftr-signed*:
⟦ *abstract-val Px x sint* (*x′* :: ('*a* :: *len*) *signed word*);
abstract-val Pn n sint (*n′* :: ('*b* :: *len*) *signed word*) ⟧ ⟹
abstract-val (*Px* ∧ *Pn* ∧ *0* ≤ *x* ∧ *0* ≤ *n* ∧ *n* < *int LENGTH*('*a*))
(*x* >> (*nat n*)) *sint* (*x′* >> (*unat n′*))
⟨*proof*⟩

lemma *abstract-val-signed-shiftr-unsigned*:
⟦ *abstract-val Px x sint* (*x′* :: ('*a* :: *len*) *signed word*);
abstract-val Pn n unat (*n′* :: ('*b* :: *len*) *word*) ⟧ ⟹
abstract-val (*Px* ∧ *Pn* ∧ *0* ≤ *x* ∧ *n* < *LENGTH*('*a*))
(*x* >> *n*) *sint* (*x′* >> *unat n′*)
⟨*proof*⟩

lemma *foo*: ¬ *n* − *i* < *LENGTH*('*a*::*len*) − *Suc 0* ⟹
n < *LENGTH*('*a*) − *Suc 0* = *False*

⟨*proof*⟩

lemma *sint-shiftl-nonneg*:
⟦ *0 <=s* (*x* :: *'a*::*len signed word*); *n* < *LENGTH*(*'a*); *sint x <<* *n* < *2^*(*LENGTH*(*'a*) − *1*) ⟧ ⟹
sint (*x <<* *n*) = *sint x <<* *n*
⟨*proof*⟩

lemma *abstract-val-signed-shiftl-signed*:
⟦ *abstract-val Px x sint* (*x'* :: (*'a* :: *len*) *signed word*);
abstract-val Pn n sint (*n'* :: (*'b* :: *len*) *signed word*) ⟧ ⟹
abstract-val (*Px* ∧ *Pn* ∧ *0* ≤ *x* ∧ *0* ≤ *n* ∧ *n* < *int LENGTH*(*'a*) ∧ *x <<* *nat* *n* < *2^*(*LENGTH*(*'a*) − *1*))
(*x <<* *nat n*) *sint* (*x' <<* *unat n'*)
⟨*proof*⟩

lemma *abstract-val-signed-shiftl-unsigned*:
⟦ *abstract-val Px x sint* (*x'* :: (*'a* :: *len*) *signed word*);
abstract-val Pn n unat (*n'* :: (*'b* :: *len*) *word*) ⟧ ⟹
abstract-val (*Px* ∧ *Pn* ∧ *0* ≤ *x* ∧ *n* < *LENGTH*(*'a*) ∧ *x <<* *n* < *2^*(*LENGTH*(*'a*) − *1*))
(*x <<* *n*) *sint* (*x' <<* *unat n'*)
⟨*proof*⟩

lemma *abstract-val-unsigned-shiftr-unsigned*:
⟦ *abstract-val Px x unat* (*x'* :: (*'a* :: *len*) *word*);
abstract-val Pn n unat (*n'* :: (*'a* :: *len*) *word*) ⟧ ⟹
abstract-val (*Px* ∧ *Pn*) (*x >>* *n*) *unat* (*x' >>* *unat n'*)
⟨*proof*⟩

lemma *abstract-val-unsigned-shiftr-signed*:
⟦ *abstract-val Px x unat* (*x'* :: (*'a* :: *len*) *word*);
abstract-val Pn n sint (*n'* :: (*'b* :: *len*) *signed word*) ⟧ ⟹
abstract-val (*Px* ∧ *Pn* ∧ *0* ≤ *n*) (*x >>* *nat n*) *unat* (*x' >>* *unat n'*)
⟨*proof*⟩

lemma *abstract-val-unsigned-shiftl-unsigned*:
⟦ *abstract-val Px x unat* (*x'* :: (*'a* :: *len*) *word*);
abstract-val Pn n unat (*n'* :: (*'b* :: *len*) *word*) ⟧ ⟹
abstract-val (*Px* ∧ *Pn* ∧ *n* < *LENGTH*(*'a*) ∧ *x <<* *n* < *2^LENGTH*(*'a*))
(*x <<* *n*) *unat* (*x' <<* *unat n'*)
⟨*proof*⟩

lemma *abstract-val-unsigned-shiftl-signed*:
⟦ *abstract-val Px x unat* (*x'* :: (*'a* :: *len*) *word*);
abstract-val Pn n sint (*n'* :: (*'b* :: *len*) *signed word*) ⟧ ⟹
abstract-val (*Px* ∧ *Pn* ∧ *0* ≤ *n* ∧ *n* < *int* (*LENGTH*(*'a*)) ∧ *x <<* *nat* *n* < *2^LENGTH*(*'a*))
(*x <<* *nat n*) *unat* (*x' <<* *unat n'*)

⟨*proof*⟩

lemma *signed-shiftl-c-guard-simp* :
⟦ *int bound* < 2^*LENGTH*('*a*); *a* ∗ 2^*b* < *int bound*; 0 ≤ *a* ⟧ ⟹
unat (*of-int a* :: '*a*::*len word*) ∗ 2 ^ *b* < *bound*
⟨*proof*⟩

lemmas *abstract-val-signed-ops* [*simplified simp-thms*] =
abstract-expr-bool-binop [*OF snat-abstract-bool-binops*(1)]
abstract-expr-bool-binop [*OF snat-abstract-bool-binops*(2)]
abstract-expr-bool-binop [*OF snat-abstract-bool-binops*(3)]
abstract-expr-binop [*OF snat-abstract-binops*(1)]
abstract-expr-binop [*OF snat-abstract-binops*(2)]
abstract-expr-binop [*OF snat-abstract-binops*(3)]
abstract-expr-binop [*OF snat-abstract-binops*(4)]
abstract-expr-binop [*OF snat-abstract-binops*(5)]
abstract-expr-binop [*OF sint-bitwise-abstract-binops*(1)]
abstract-expr-binop [*OF sint-bitwise-abstract-binops*(2)]
abstract-expr-binop [*OF sint-bitwise-abstract-binops*(3)]
abstract-val-signed-bitNOT
abstract-val-signed-unary-minus
abstract-val-signed-shiftr-signed
abstract-val-signed-shiftr-unsigned
abstract-val-signed-shiftl-signed
abstract-val-signed-shiftl-unsigned

lemmas *abstract-val-unsigned-ops* [*simplified simp-thms*] =
abstract-expr-bool-binop [*OF unat-abstract-bool-binops*(1)]
abstract-expr-bool-binop [*OF unat-abstract-bool-binops*(2)]
abstract-expr-bool-binop [*OF unat-abstract-bool-binops*(3)]
abstract-expr-binop [*OF unat-abstract-binops*(1)]
abstract-expr-binop [*OF unat-abstract-binops*(2)]
abstract-expr-binop [*OF unat-abstract-binops*(3)]
abstract-expr-binop [*OF unat-abstract-binops*(4)]
abstract-expr-binop [*OF unat-abstract-binops*(5)]
abstract-expr-binop [*OF unat-bitwise-abstract-binops*(1)]
abstract-expr-binop [*OF unat-bitwise-abstract-binops*(2)]
abstract-expr-binop [*OF unat-bitwise-abstract-binops*(3)]
abstract-val-unsigned-bitNOT
abstract-val-unsigned-unary-minus
abstract-val-unsigned-shiftr-signed
abstract-val-unsigned-shiftr-unsigned
abstract-val-unsigned-shiftl-signed
abstract-val-unsigned-shiftl-unsigned

lemma *mod-less*:
(*a* :: *nat*) < *c* ⟹ *a mod b* < *c*
⟨*proof*⟩

lemma *abstract-val-ucast*:
⟦ *introduce-typ-abs-fn* (*unat* :: ('*a*::*len*) *word* ⇒ *nat*);
abstract-val P v unat v' ⟧
⟹ *abstract-val* (*P* ∧ *v* ≤ *nat* (*WORD-MAX TYPE*('*a*)))
(*int v*) *sint* (*ucast* (*v'* :: '*a word*) :: '*a signed word*)
⟨*proof*⟩

lemma *abstract-val-heap-sword-template*:
⟦ *introduce-typ-abs-fn* (*sint* :: ('*a*::*len*) *signed word* ⇒ *int*);
abstract-val P p' id p ⟧
⟹ *abstract-val P* (*sint* (*ucast* (*heap-get s p'* :: '*a word*) :: '*a signed word*))
sint (*ucast* (*heap-get s p*) :: '*a signed word*)
⟨*proof*⟩

lemma *abstract-val-scast*:
⟦ *introduce-typ-abs-fn* (*sint* :: ('*a*::*len*) *signed word* ⇒ *int*);
abstract-val P C' sint C ⟧
⟹ *abstract-val* (*P* ∧ *0* ≤ *C'*) (*nat C'*) *unat* (*scast* (*C* :: ('*a*::*len*) *signed word*) :: ('*a*::*len*) *word*)
⟨*proof*⟩

lemma *abstract-val-scast-upcast*:
⟦ *len-of TYPE*('*a*::*len*) ≤ *len-of TYPE*('*b*::*len*);
abstract-val P C' sint C ⟧
⟹ *abstract-val P* (*C'*) *sint* (*scast* (*C* :: '*a signed word*) :: '*b signed word*)
⟨*proof*⟩

lemma *abstract-val-scast-downcast*:
⟦ *len-of TYPE*('*b*) < *len-of TYPE*('*a*::*len*);
abstract-val P C' sint C ⟧
⟹ *abstract-val P* (*sbintrunc* ((*len-of TYPE*('*b*::*len*) − *1*)) *C'*) *sint* (*scast* (*C* :: '*a signed word*) :: '*b signed word*)
⟨*proof*⟩

lemma *abstract-val-ucast-upcast*:
⟦ *len-of TYPE*('*a*::*len*) ≤ *len-of TYPE*('*b*::*len*);
abstract-val P C' unat C ⟧
⟹ *abstract-val P* (*C'*) *unat* (*ucast* (*C* :: '*a word*) :: '*b word*)
⟨*proof*⟩

lemma *abstract-val-ucast-downcast*:
⟦ *len-of TYPE*('*b*::*len*) < *len-of TYPE*('*a*::*len*);
abstract-val P C' unat C ⟧
⟹ *abstract-val P* (*C' mod* (*UWORD-MAX TYPE*('*b*) + *1*)) *unat* (*ucast* (*C* :: '*a word*) :: '*b word*)
⟨*proof*⟩

definition
valid-typ-abs-fn (*P* :: *'a* ⇒ *bool*) (*Q* :: *'a* ⇒ *bool*) (*A* :: *'c* ⇒ *'a*) (*C* :: *'a* ⇒ *'c*) ≡
(∀ *v*. *P v* ⟶ *A* (*C v*) = *v*) ∧ (∀ *v*. *Q* (*A v*) ⟶ *C* (*A v*) = *v*)

declare *valid-typ-abs-fn-def* [*simp*]

lemma *valid-typ-abs-fn-id*:
valid-typ-abs-fn (λ-. *True*) (λ-. *True*) *id id*
⟨*proof*⟩

lemma *valid-typ-abs-fn-unit*:
valid-typ-abs-fn (λ-. *True*) (λ-. *True*) *id* (*id* :: *unit* ⇒ *unit*)
⟨*proof*⟩

lemma *valid-typ-abs-fn-unat*:
valid-typ-abs-fn (λ*v*. *v* ≤ *UWORD-MAX TYPE*(*'a*::*len*)) (λ-. *True*) (*unat* :: *'a word* ⇒ *nat*) (*of-nat* :: *nat* ⇒ *'a word*)
⟨*proof*⟩

lemma *valid-typ-abs-fn-sint*:
valid-typ-abs-fn (λ*v*. *WORD-MIN TYPE*(*'a*::*len*) ≤ *v* ∧ *v* ≤ *WORD-MAX TYPE*(*'a*)) (λ-. *True*) (*sint* :: *'a signed word* ⇒ *int*) (*of-int* :: *int* ⇒ *'a signed word*)
⟨*proof*⟩

lemma *valid-typ-abs-fn-tuple*:
⟦ *valid-typ-abs-fn P-a Q-a abs-a conc-a*; *valid-typ-abs-fn P-b Q-b abs-b conc-b* ⟧
⟹
valid-typ-abs-fn (λ(*a*, *b*). *P-a a* ∧ *P-b b*) (λ(*a*, *b*). *Q-a a* ∧ *Q-b b*) (*map-prod abs-a abs-b*) (*map-prod conc-a conc-b*)
⟨*proof*⟩

lemma *valid-typ-abs-fn-tuple-split*:
⟦ *valid-typ-abs-fn P-a Q-a abs-a conc-a*; *valid-typ-abs-fn P-b Q-b abs-b conc-b* ⟧
⟹
valid-typ-abs-fn (λ(*a*, *b*). *P-a a* ∧ *P-b b*) (λ(*a*, *b*). *Q-a a* ∧ *Q-b b*) (λ(*a*, *b*). (*abs-a a*, *abs-b b*)) (*map-prod conc-a conc-b*)
⟨*proof*⟩

lemma *introduce-typ-abs-fn-tuple*:
⟦ *introduce-typ-abs-fn abs-a*; *introduce-typ-abs-fn abs-b* ⟧ ⟹
introduce-typ-abs-fn (*map-prod abs-a abs-b*)
⟨*proof*⟩

lemma *valid-typ-abs-fn-sum*:
⟦ *valid-typ-abs-fn P-a Q-a abs-a conc-a*; *valid-typ-abs-fn P-b Q-b abs-b conc-b* ⟧
⟹
valid-typ-abs-fn (*case-sum P-a P-b*) (*case-sum Q-a Q-b*) (*map-sum abs-a*

abs-b) (*map-sum conc-a conc-b*)
⟨*proof*⟩

lemma *introduce-typ-abs-fn-sum*:
⟦ *introduce-typ-abs-fn abs-a*; *introduce-typ-abs-fn abs-b* ⟧ ⟹
introduce-typ-abs-fn (*map-sum abs-a abs-b*)
⟨*proof*⟩

22.3 Refinement Lemmas

named-theorems *word-abs*

definition
corresTA P rx ex A C ≡ *corresXF* (λ*s*. *s*) (λ*r s*. *rx r*) (λ*r s*. *ex r*) *P A C*

definition *rel-word-abs ex rx* ≡ *rel-xval* (λ*c a*. *a* = *ex c*) (λ*c a*. *a* = *rx c*)

lemma *rel-word-abs-simps*[*simp*]:
rel-word-abs ex rx (*Result rc*) (*Exn la*) = *False*
rel-word-abs ex rx (*Exn lc*) (*Result ra*) = *False*
rel-word-abs ex rx (*Result rc*) (*Result ra*) = (*ra* = *rx rc*)
rel-word-abs ex rx (*Exn lc*) (*Exn la*) = (*la* = *ex lc*)
⟨*proof*⟩

lemma *corresTA-refines*:
corresTA P rx ex f_a f_c ⟹ *P s* ⟹ *refines* f_c f_a *s s* (*rel-prod* (*rel-word-abs ex rx*) (=))
⟨*proof*⟩

lemma *refines-corresTA*:
assumes *sim*: ⋀*s*. *P s* ⟹ *refines* f_c f_a *s s* (*rel-prod* (*rel-word-abs ex rx*) (=))
shows *corresTA P rx ex* f_a f_c
⟨*proof*⟩

lemma *corresTA-refines-conv*:
corresTA P rx ex f_a f_c ⟷ (∀ *s*. *P s* ⟶ *refines* f_c f_a *s s* (*rel-prod* (*rel-word-abs ex rx*) (=)))
⟨*proof*⟩

lemma *admissible-nondet-ord-corresTA* [*corres-admissible*]:
ccpo.admissible Inf (≥) (λ*A*. *corresTA P rx ex A C*)
⟨*proof*⟩

lemma *corresTA-top* [*corres-top*]: *corresTA P rx st* ⊤ *C*
⟨*proof*⟩

lemma *corresTA-assume-and-weaken-pre*:
assumes *A-C*: ⋀*s*. *P s* ⟹ *corresTA Q rt ex A C*
assumes *P-Q*: ⋀*s*. *P s* ⟹ *Q s*

shows *corresTA P rt ex A C*
⟨*proof*⟩

lemma *corresTA-L2-gets*:
⟦ ⋀*s*. *abstract-val* (*Q s*) (*C s*) *rx* (*C′ s*) ⟧ ⟹
corresTA Q rx ex (*L2-gets* (λ*s*. *C s*) *n*) (*L2-gets* (λ*s*. *C′ s*) *n*)
⟨*proof*⟩

lemma *corresTA-L2-modify*:
⟦ ⋀*s*. *abstract-val* (*P s*) (*m s*) *id* (*m′ s*) ⟧ ⟹
corresTA P rx ex (*L2-modify* (λ*s*. *m s*)) (*L2-modify* (λ*s*. *m′ s*))
⟨*proof*⟩

lemma (**in** *heap-state*) *corresTA-IO-modify-heap-paddingE*[*word-abs*]:
abstract-val P p id p′ ⟹ (⋀*s*. *abstract-val* (*Q s*) (*v s*) *id* (*v′ s*)) ⟹
corresTA (λ*s*. *P* ∧ *Q s*) *rx ex* (*IO-modify-heap-paddingE p v*) (*IO-modify-heap-paddingE p′ v′*)
⟨*proof*⟩

lemma *refines-throw*: *R* (*Exn x*, *s*) (*Exn y*, *t*) ⟹ *refines* (*throw x*) (*throw y*) *s t R*
⟨*proof*⟩

lemma *corresTA-L2-throw*:
⟦ *abstract-val Q C ex C′* ⟧ ⟹
corresTA (λ-. *Q*) *rx ex* (*L2-throw C n*) (*L2-throw C′ n*)
⟨*proof*⟩

lemma *corresTA-L2-skip*:
corresTA (λ-. *True*) *rx ex L2-skip L2-skip*
⟨*proof*⟩

lemma *corresTA-L2-fail*:
corresTA (λ-. *True*) *rx ex L2-fail L2-fail*
⟨*proof*⟩

lemma *corresTA-L2-seq′*:
fixes *L′* :: (*′e*, *′c1*, *′s*) *exn-monad*
fixes *R′* :: *′c1* ⇒ (*′e*, *′c2*, *′s*) *exn-monad*
fixes *L* :: (*′ea*, *′a1*, *′s*) *exn-monad*
fixes *R* :: *′a1* ⇒ (*′ea*, *′a2*, *′s*) *exn-monad*
shows
⟦ *corresTA P rx1 ex L L′*;
⋀*r*. *corresTA* (*Q* (*rx1 r*)) *rx2 ex* (*R* (*rx1 r*)) (*R′ r*) ⟧ ⟹
corresTA P rx2 ex
(*L2-seq L* (λ*r*. *L2-seq* (*L2-guard* (λ*s*. *Q r s*)) (λ-. *R r*)))

(*L2-seq* L' ($\lambda r.$ R' r))
⟨*proof*⟩

lemma *corresTA-L2-seq*:
⟦ *introduce-typ-abs-fn rx1*;
PROP THIN (*Trueprop* (*corresTA* P ($rx1$ :: $'a \Rightarrow 'b$) *ex* L L'));
PROP THIN ($\bigwedge r\ r'.$ *abs-var* r *rx1* $r' \Longrightarrow$ *corresTA* (Q r) *rx2 ex* (R r) (R' r')) ⟧ ⟹
corresTA P *rx2 ex* (*L2-seq* L ($\lambda r.$ *L2-seq* (*L2-guard* (Q r)) (λ-. R r))) (*L2-seq* L' R')
⟨*proof*⟩

lemma *corresTA-L2-seq-unused-result*:
⟦ *introduce-typ-abs-fn rx1*;
PROP THIN (*Trueprop* (*corresTA* P ($rx1$ :: $'a \Rightarrow 'b$) *ex* L L'));
PROP THIN (*Trueprop* (*corresTA* Q *rx2 ex* R R'))⟧ ⟹
corresTA P *rx2 ex* (*L2-seq* L ($\lambda r.$ *L2-seq* (*L2-guard* Q) (λ-. R))) (*L2-seq* L' (λ-. R'))
⟨*proof*⟩

lemma *corresTA-L2-seq-unit*:
fixes L' :: ($'e$, *unit*, $'s$) *exn-monad*
fixes R' :: *unit* $\Rightarrow$ ($'e$, $'r$, $'s$) *exn-monad*
fixes L :: ($'ea$, *unit*, $'s$) *exn-monad*
fixes R :: ($'ea$, $'ra$, $'s$) *exn-monad*
shows
⟦*PROP THIN* (*Trueprop* (*corresTA* P *id ex* L L'));
PROP THIN (*Trueprop* (*corresTA* Q *rx ex* R (R' ()))) ⟧ ⟹
corresTA P *rx ex*
(*L2-seq* L ($\lambda r.$ *L2-seq* (*L2-guard* Q) (λ-. R)))
(*L2-seq* L' R')
⟨*proof*⟩

lemma *corresTA-L2-catch'*:
fixes L' :: ($'e1$, $'c$, $'s$) *exn-monad*
fixes R' :: $'e1 \Rightarrow$ ($'e2$, $'c$, $'s$) *exn-monad*
fixes L :: ($'e1a$, $'ca$, $'s$) *exn-monad*
fixes R :: $'e1a \Rightarrow$ ($'e2a$, $'ca$, $'s$) *exn-monad*
shows
⟦*corresTA* P *rx ex1* L L';
$\bigwedge r.$ *corresTA* (Q (*ex1* r)) *rx ex2* (R (*ex1* r)) (R' r) ⟧ ⟹
corresTA P *rx ex2* (*L2-catch* L ($\lambda r.$ *L2-seq* (*L2-guard* ($\lambda s.$ Q r s)) (λ-. R r))) (*L2-catch* L' ($\lambda r.$ R' r))
⟨*proof*⟩

lemma *corresTA-L2-catch*:
⟦ *introduce-typ-abs-fn ex1*;
PROP THIN (*Trueprop* (*corresTA* P *rx ex1* L L'));
PROP THIN ($\bigwedge r\ r'.$ *abs-var* r *ex1* $r' \Longrightarrow$ *corresTA* (Q r) *rx ex2* (R r) (R' r'))

$\rrbracket \Longrightarrow$
corresTA P rx ex2 (L2-catch L (λr. L2-seq (L2-guard (λs. Q r s)) (λ-. R r))) (L2-catch L′ (λr. R′ r))
⟨*proof*⟩

term *corresTA P rx ex f g*

lemma *corresTA-yield*:
abstract-val True v′ (map-xval ex rx) v ⟹ corresTA P rx ex (yield v′) (yield v)
⟨*proof*⟩

lemma *map-sum-apply*: *map-sum ex rx = (λv. case v of Inl l ⇒ Inl (ex l) | Inr r ⇒ Inr (rx r))*
⟨*proof*⟩

lemma *refines-try-rel-prod*:
assumes *refines f g s t (rel-prod (rel-xval (rel-sum L R) R) S)*
shows *refines (try f) (try g) s t (rel-prod (rel-xval L R) S)*
⟨*proof*⟩

lemma *rel-map-xval-sum-rel-sum-conv*:
rel-xval (λc a. a = map-sum ex rx c) (λc a. a = rx c) =
rel-xval (rel-sum (λc a. a = ex c) (λc a. a = rx c)) (λc a. a = rx c)
⟨*proof*⟩

lemma *corresTA-L2-try′*:
assumes *corres-L-L′*: *corresTA P rx (map-sum ex rx) L L′*
shows *corresTA P rx ex (L2-try L) (L2-try L′)*
⟨*proof*⟩

lemma *corresTA-L2-while*:
assumes *init-corres*: *abstract-val Q i rx i′*
and *cond-corres*: *PROP THIN (⋀r r′ s. abs-var r rx r′ ⟹ abstract-val (G r s) (C r s) id (C′ r′ s))*
and *body-corres*: *PROP THIN (⋀r r′. abs-var r rx r′ ⟹ corresTA (P r) rx ex (B r) (B′ r′))*
shows *corresTA (λ-. Q) rx ex*
(L2-guarded-while (λr s. G r s) (λr s. C r s) (λr. L2-seq (L2-guard (λs. P r s)) (λ-. B r)) i x)
(L2-while (λr s. C′ r s) B′ i′ x)
⟨*proof*⟩

lemma *corresTA-L2-guard*:

⟦ ⋀s. abstract-val (Q s) (G s) id (G′ s) ⟧
⟹ corresTA (λ-. True) rx ex (L2-guard (λs. G s ∧ Q s)) (L2-guard (λs. G′ s))
⟨proof⟩

lemma *corresTA-L2-guard′*:
⟦⋀s. abstract-val (Q s) (G s) id (G′ s);
⋀s. R s ⟹ G s ∧ Q s⟧
⟹ corresTA (λ-. True) rx ex (L2-guard (λs. R s)) (L2-guard (λs. G′ s))
⟨proof⟩

lemma *corresTA-L2-guarded-simple*:
assumes G-G′: ⋀s. abstract-val (Q s) (G s) id (G′ s)
assumes f-f′: ⋀s. G′ s ⟹ Q s ⟹ G s ⟹ corresTA P rx ex f f′
shows corresTA (λ-. True) rx ex (L2-guarded (λs. G s ∧ Q s ∧ P s) f) (L2-guarded G′ f′)
⟨proof⟩

lemma *corresTA-L2-spec*:
(⋀s t. abstract-val (Q s) (P s t) id (P′ s t)) ⟹
corresTA Q rx ex (L2-spec {(s, t). P s t}) (L2-spec {(s, t). P′ s t})
⟨proof⟩

lemma *corresTA-L2-assume*:
(⋀s r t. abstract-val (Q s) (P s) (λX. (λ(x, y). (rx x, y)) ‘ X) (P′ s)) ⟹
corresTA Q rx ex (L2-assume P) (L2-assume P′)
⟨proof⟩

lemma *corresTA-L2-condition*:
⟦PROP THIN (Trueprop (corresTA P rx ex L L′));
PROP THIN (Trueprop (corresTA Q rx ex R R′));
⋀s. abstract-val (T s) (C s) id (C′ s) ⟧
⟹ corresTA T rx ex
(L2-condition (λs. C s)
(L2-seq (L2-guard P) (λ-. L))
(L2-seq (L2-guard Q) (λ-. R))
) (L2-condition (λs. C′ s) L′ R′)
⟨proof⟩

lemma *L2-call-L2-defs*: L2-call x emb ns = L2-catch x (λe. L2-throw (emb e) ns)
⟨proof⟩

lemma *corresTA-L2-call*:
⟦corresTA P rx ex′ A B;
⋀r r′. abs-var r ex′ r′ ⟹ abstract-val Q (emb r) ex (emb′ r′)
⟧ ⟹
corresTA (λs. P s ∧ Q) rx ex (L2-call A emb ns) (L2-call B emb′ ns)

⟨*proof*⟩

lemma *corresTA-L2-call'*:
⟦ *corresTA P f1 ex' A B*;
valid-typ-abs-fn Q1 Q1' f1 f1';
valid-typ-abs-fn Q2 Q2' f2 f2';
⋀*r r'*. *abs-var r ex' r'* ⟹ *abstract-val Q* (*emb r*) *ex* (*emb' r'*)
⟧ ⟹
corresTA (λ*s*. *P s* ∧ *Q*) *f2 ex*
(*L2-seq* (*L2-call A emb ns*) (*ETA-TUPLED* (λ*ret*. (*L2-seq* (*L2-guard* (λ-. *Q1' ret*)) (λ-. *L2-gets* (λ-. *f2* (*f1' ret*)) *ns*)))))
(*L2-call B emb' ns*)
⟨*proof*⟩

lemma *corresTA-L2-unknown*:
corresTA (λ-. *True*) *rx ex* (*L2-unknown x*) (*L2-unknown x*)
⟨*proof*⟩

lemma *corresTA-L2-call-exec-concrete*:
⟦ *corresTA P rx ex' A B* ;
⋀*r r'*. *abs-var r ex' r'* ⟹ *abstract-val Q* (*emb r*) *ex* (*emb' r'*)⟧ ⟹
corresTA (λ*s*. ∀ *s'*. *s* = *st s'* ⟶ *P s'* ∧ *Q*) *rx ex*
(*exec-concrete st* (*L2-call A emb ns*))
(*exec-concrete st* (*L2-call B emb' ns*))
⟨*proof*⟩

lemma *corresTA-L2-call-exec-abstract*:
⟦ *corresTA P rx ex' A B*;
⋀*r r'*. *abs-var r ex' r'* ⟹ *abstract-val Q* (*emb r*) *ex* (*emb' r'*) ⟧ ⟹
corresTA (λ*s*. *P* (*st s*) ∧ *Q*) *rx ex*
(*exec-abstract st* (*L2-call A emb ns*))
(*exec-abstract st* (*L2-call B emb' ns*))
⟨*proof*⟩

lemma *corresTA-L2-call-exec-concrete'*:
⟦ *corresTA P f1 ex' A B*;
valid-typ-abs-fn Q1 Q1' f1 f1';
valid-typ-abs-fn Q2 Q2' f2 f2';
⋀*r r'*. *abs-var r ex' r'* ⟹ *abstract-val Q* (*emb r*) *ex* (*emb' r'*)
⟧ ⟹
corresTA (λ*s*. ∀ *s'*. *s* = *st s'* ⟶ *P s'* ∧ *Q*) *f2 ex*
(*L2-seq* (*exec-concrete st* (*L2-call A emb ns*)) (λ*ret*. (*L2-seq* (*L2-guard* (λ-. *Q1' ret*)) (λ-. *L2-gets* (λ-. *f2* (*f1' ret*)) [])))))
(*exec-concrete st* (*L2-call B emb' ns*))

⟨*proof*⟩

lemma *corresTA-L2-call-exec-abstract'*:
⟦ *corresTA P f1 ex' A B*;
valid-typ-abs-fn Q1 Q1' f1 f1';
valid-typ-abs-fn Q2 Q2' f2 f2';
⋀*r r'*. *abs-var r ex' r'* ⟹ *abstract-val Q* (*emb r*) *ex* (*emb' r'*)
⟧ ⟹
corresTA (λ*s*. *P* (*st s*) ∧ *Q*) *f2 ex*
(*L2-seq* (*exec-abstract st* (*L2-call A emb ns*)) (λ*ret*. (*L2-seq* (*L2-guard* (λ-. *Q1' ret*)) (λ-. *L2-gets* (λ-. *f2* (*f1' ret*)) [])))))
(*exec-abstract st* (*L2-call B emb' ns*))
⟨*proof*⟩

Avoid higher-order unification issues by explicit application with ($):

- in concrete program position enforces 'obvious' instantiation
- in abstract program position enforces introduction of two separate variables for *a* and *b* instead of a higher-order flex-flex pair.

lemma *abstract-val-fun-app*:
⟦ *abstract-val Q b id b'*; *abstract-val P a id a'* ⟧ ⟹
abstract-val (*P* ∧ *Q*) (*f* $ (*a* $ *b*)) *f* (*a'* $ *b'*)
⟨*proof*⟩

lemma *corresTA-precond-to-guard*:
corresTA (λ*s*. *P s*) *rx ex A A'* ⟹ *corresTA* (λ-. *True*) *rx ex* (*L2-seq* (*L2-guard* (λ*s*. *P s*)) (λ-. *A*)) *A'*
⟨*proof*⟩

lemma *corresTA-precond-to-asm*:
⟦ ⋀*s*. *P s* ⟹ *corresTA* (λ-. *True*) *rx ex A A'* ⟧ ⟹ *corresTA P rx ex A A'*
⟨*proof*⟩

lemma *L2-guard-true*: *L2-seq* (*L2-guard* (λ-. *True*)) *A* = *A* ()
⟨*proof*⟩

lemma *corresTA-simp-trivial-guard*:
corresTA P rx ex (*L2-seq* (*L2-guard* (λ-. *True*)) *A*) *C* ≡ *corresTA P rx ex* (*A* ()) *C*
⟨*proof*⟩

lemma *corresTA-extract-preconds-of-call-init*:
⟦ *corresTA* (λ*s*. *P*) *rx ex A A'* ⟧ ⟹ *corresTA* (λ*s*. *P* ∧ *True*) *rx ex A A'*
⟨*proof*⟩

lemma *corresTA-extract-preconds-of-call-step*:

⟦ *corresTA* (λ*s*. (*abs-var a f a'* ∧ *R*) ∧ *C*) *rx ex A A'*; *abstract-val Y a f a'* ⟧
⟹ *corresTA* (λ*s*. *R* ∧ (*Y* ∧ *C*)) *rx ex A A'*
⟨*proof*⟩

lemma *corresTA-extract-preconds-of-call-final*:
⟦ *corresTA* (λ*s*. (*abs-var a f a'*) ∧ *C*) *rx ex A A'*; *abstract-val Y a f a'* ⟧
⟹ *corresTA* (λ*s*. (*Y* ∧ *C*)) *rx ex A A'*
⟨*proof*⟩

lemma *corresTA-extract-preconds-of-call-final'*:
⟦ *corresTA* (λ*s*. *True* ∧ *C*) *rx ex A A'* ⟧
⟹ *corresTA* (λ*s*. *C*) *rx ex A A'*
⟨*proof*⟩

lemma *corresTA-extract-preconds-of-call-init-prems*:
⟦ *corresTA* (λ*s*. *P* ∧ *True*) *rx ex A A'*⟧ ⟹ *corresTA* (λ*s*. *P*) *rx ex A A'*
⟨*proof*⟩

lemma *corresTA-extract-preconds-of-call-step-prems*:
⟦⋀*Y*. *abstract-val Y a f a'* ⟹ *corresTA* (λ*s*. *R* ∧ (*Y* ∧ *C*)) *rx ex A A'* ⟧
⟹ *corresTA* (λ*s*. (*abs-var a f a'* ∧ *R*) ∧ *C*) *rx ex A A'*
⟨*proof*⟩

lemma *corresTA-extract-preconds-of-call-final-prems*:
⟦⋀*Y*. *abstract-val Y a f a'* ⟹ *corresTA* (λ*s*. (*Y* ∧ *C*)) *rx ex A A'*⟧
⟹ *corresTA* (λ*s*. (*abs-var a f a'*) ∧ *C*) *rx ex A A'*
⟨*proof*⟩

lemma *corresTA-extract-preconds-of-call-final'-prems*:
⟦*corresTA* (λ*s*. *C*) *rx ex A A'* ⟧
⟹ *corresTA* (λ*s*. *True* ∧ *C*) *rx ex A A'*
⟨*proof*⟩

lemma *corresTA-case-prod*:
⟦ *introduce-typ-abs-fn rx1*;
introduce-typ-abs-fn rx2;
abstract-val (*Q x*) *x* (*map-prod rx1 rx2*) *x'*;
⋀*a b a' b'*. ⟦ *abs-var a rx1 a'*; *abs-var b rx2 b'* ⟧
⟹ *corresTA* (*P a b*) *rx ex* (*M a b*) (*M' a' b'*) ⟧ ⟹
corresTA (λ*s*. *case x of* (*a*, *b*) ⇒ *P a b s* ∧ *Q* (*a*, *b*)) *rx ex* (*case x of* (*a*, *b*) ⇒
M a b) (*case x' of* (*a*, *b*) ⇒ *M' a b*)
⟨*proof*⟩

lemma *abstract-val-case-prod*:
⟦ *abstract-val True r* (*map-prod f g*) *r'*;
⋀*a b a' b'*. ⟦ *abs-var a f a'*; *abs-var b g b'* ⟧
⟹ *abstract-val* (*P a b*) (*M a b*) *h* (*M' a' b'*) ⟧
⟹ *abstract-val* (*P* (*fst r*) (*snd r*))
(*case r of* (*a*, *b*) ⇒ *M a b*) *h*

(*case* r' *of* $(a,\ b) \Rightarrow M'\ a\ b$)
⟨*proof*⟩

lemma *abstract-val-case-prod-fun-app*:
⟦ *abstract-val True r* (*map-prod f g*) r';
$\bigwedge a\ b\ a'\ b'$. ⟦ *abs-var a f a'*; *abs-var b g b'* ⟧
$\Longrightarrow$ *abstract-val* (*P a b*) (*M a b s*) *h* (*M' a' b' s*) ⟧
$\Longrightarrow$ *abstract-val* (*P* (*fst r*) (*snd r*))
((*case r of* (*a*, *b*) ⇒ *M a b*) *s*) *h*
((*case r' of* (*a*, *b*) ⇒ *M' a b*) *s*)
⟨*proof*⟩

lemma *abstract-val-of-nat*:
abstract-val ($r \leq$ *UWORD-MAX TYPE*('*a*::*len*)) *r unat* (*of-nat r* :: '*a word*)
⟨*proof*⟩

lemma *abstract-val-of-int*:
abstract-val (*WORD-MIN TYPE*('*a*::*len*) $\leq r \wedge r \leq$ *WORD-MAX TYPE*('*a*)) *r*
sint (*of-int r* :: '*a signed word*)
⟨*proof*⟩

lemma *abstract-val-tuple*:
⟦ *abstract-val P a absL a'*;
abstract-val Q b absR b' ⟧ $\Longrightarrow$
abstract-val (*P* $\wedge$ *Q*) (*a*, *b*) (*map-prod absL absR*) (*a'*, *b'*)
⟨*proof*⟩

lemma *abstract-val-Inl*:
⟦ *abstract-val P a absL a'* ⟧ $\Longrightarrow$
abstract-val P (*Inl a*) (*map-sum absL absR*) (*Inl a'*)
⟨*proof*⟩

lemma *abstract-val-Inr*:
⟦ *abstract-val P b absR b'* ⟧ $\Longrightarrow$
abstract-val P (*Inr b*) (*map-sum absL absR*) (*Inr b'*)
⟨*proof*⟩

lemma *abstract-val-func*:
⟦ *abstract-val P a id a'*; *abstract-val Q b id b'* ⟧
$\Longrightarrow$ *abstract-val* (*P* $\wedge$ *Q*) (*f a b*) *id* (*f a' b'*)
⟨*proof*⟩

lemma *abstract-val-conj*:
⟦ *abstract-val P a id a'*;
abstract-val Q b id b' ⟧ $\Longrightarrow$
abstract-val (*P* $\wedge$ (*a* $\longrightarrow$ *Q*)) (*a* $\wedge$ *b*) *id* (*a'* $\wedge$ *b'*)
⟨*proof*⟩

lemma *abstract-val-disj*:
⟦ *abstract-val P a id a′*;
abstract-val Q b id b′ ⟧ ⟹
abstract-val (*P* ∧ (¬ *a* ⟶ *Q*)) (*a* ∨ *b*) *id* (*a′* ∨ *b′*)
⟨*proof*⟩

lemma *abstract-val-unwrap*:
⟦ *introduce-typ-abs-fn f*; *abstract-val P a f b* ⟧
⟹ *abstract-val P a id* (*f b*)
⟨*proof*⟩

lemma *abstract-val-uint*:
⟦ *introduce-typ-abs-fn unat*; *abstract-val P x unat x′* ⟧
⟹ *abstract-val P* (*int x*) *id* (*uint x′*)
⟨*proof*⟩

lemma *abstract-val-lambda*:
⟦ ⋀*v*. *abstract-val* (*P v*) (*a v*) *id* (*a′ v*) ⟧ ⟹
abstract-val (∀ *v*. *P v*) (λ*v*. *a v*) *id* (λ*v*. *a′ v*)
⟨*proof*⟩

lemma *corresTA-call-L1*:
abstract-val True arg-xf id arg-xf′ ⟹
corresTA (λ-. *True*) *id id*
(*L2-call-L1 arg-xf gs ret-xf l1body*)
(*L2-call-L1 arg-xf′ gs ret-xf l1body*)
⟨*proof*⟩

context *stack-heap-state*
begin
lemma *corresTA-with-fresh-stack-ptr*[*word-abs*]:
assumes *f*[*simplified THIN-def*, *rule-format*]: *PROP THIN* (⋀*p*. *corresTA Q rx ex* (*f*$_a$ *p*) (*f*$_c$ *p*))
assumes *init*: ⋀*s*. *abstract-val* (*P s*) (*init*$_a$ *s*) *id* (*init*$_c$ *s*)
shows *corresTA P rx ex*
(*with-fresh-stack-ptr n init*$_a$ (*L2-VARS* (λ*p*. (*L2-seq* (*L2-guard Q*) (λ-. *f*$_a$ *p*))) *nm*))
(*with-fresh-stack-ptr n init*$_c$ (*L2-VARS f*$_c$ *nm*))
⟨*proof*⟩
end

context *typ-heap-typing*
begin

lemma *corresTA-guard-with-fresh-stack-ptr*[*word-abs*]:
assumes *f*[*simplified THIN-def*, *rule-format*]: *PROP THIN* (⋀*p*. *corresTA Q rx*

ex (f_a *p*) (f_c *p*))
assumes *init*: $\bigwedge$*s*. *abstract-val* (*P s*) (*init*$_a$ *s*) *id* (*init*$_c$ *s*)
shows *corresTA P rx ex*
(*guard-with-fresh-stack-ptr n init*$_a$ (*L2-VARS* (λ*p*. (*L2-seq* (*L2-guard Q*) (λ-. f_a *p*))) *nm*))
(*guard-with-fresh-stack-ptr n init*$_c$ (*L2-VARS* f_c *nm*))
⟨*proof*⟩

lemma *corresTA-assume-with-fresh-stack-ptr*[*word-abs*]:
assumes *f*[*simplified THIN-def*, *rule-format*]: *PROP THIN* ($\bigwedge$*p*. *corresTA Q rx ex* (f_a *p*) (f_c *p*))
assumes *init*: $\bigwedge$*s*. *abstract-val* (*P s*) (*init*$_a$ *s*) *id* (*init*$_c$ *s*)
shows *corresTA P rx ex*
(*assume-with-fresh-stack-ptr n init*$_a$ (*L2-VARS* (λ*p*. (*L2-seq* (*L2-guard Q*) (λ-. f_a *p*))) *nm*))
(*assume-with-fresh-stack-ptr n init*$_c$ (*L2-VARS* f_c *nm*))
⟨*proof*⟩

lemma *corresTA-with-fresh-stack-ptr*[*word-abs*]:
assumes *f*[*simplified THIN-def*, *rule-format*]: *PROP THIN* ($\bigwedge$*p*. *corresTA Q rx ex* (f_a *p*) (f_c *p*))
assumes *init*: $\bigwedge$*s*. *abstract-val* (*P s*) (*init*$_a$ *s*) *id* (*init*$_c$ *s*)
shows *corresTA P rx ex*
(*with-fresh-stack-ptr n init*$_a$ (*L2-VARS* (λ*p*. (*L2-seq* (*L2-guard Q*) (λ-. f_a *p*))) *nm*))
(*with-fresh-stack-ptr n init*$_c$ (*L2-VARS* f_c *nm*))
⟨*proof*⟩
end

lemma *abstract-val-call-L1-args*:
abstract-val P x id x' $\Longrightarrow$ *abstract-val P y id y'* $\Longrightarrow$
abstract-val P (*x and y*) *id* (*x' and y'*)
⟨*proof*⟩

lemma *abstract-val-call-L1-arg*:
abs-var x id x' $\Longrightarrow$ *abstract-val P* (λ*s*. *f s* = *x*) *id* (λ*s*. *f s* = *x'*)
⟨*proof*⟩

lemma *abstract-val-abs-var* [*consumes 1*]:
⟦ *abs-var a f a'* ⟧ $\Longrightarrow$ *abstract-val True a f a'*
⟨*proof*⟩

lemma *abstract-val-abs-var-concretise* [*consumes 1*]:
⟦ *abs-var a A a'*; *introduce-typ-abs-fn A*; *valid-typ-abs-fn PA PC A* (*C* :: *'a* ⇒ *'c*) ⟧
$\Longrightarrow$ *abstract-val* (*PC a*) (*C a*) *id a'*

⟨*proof*⟩

lemma *abstract-val-abs-var-give-up* [*consumes 1*]:
⟦ *abs-var a id a'* ⟧ ⟹ *abstract-val True* (*A a*) *A a'*
⟨*proof*⟩

lemma *abstract-val-abs-var-sint-unat* [*consumes 1*]:
⟦ *abs-var a sint a'* ⟧ ⟹ *abstract-val* (*0* ≤ *a*) (*nat a*) *id* (*unat a'*)
⟨*proof*⟩

lemma *abstract-val-abs-var-uint-unat* [*consumes 1*]:
⟦ *abs-var a uint a'* ⟧ ⟹ *abstract-val True* (*nat a*) *id* (*unat a'*)
⟨*proof*⟩

lemma *abs-var-id*: (*abs-var a id a'*) = (*a'* = *a*)
⟨*proof*⟩

lemma *abstract-val-id*: *abstract-val P a id a*
⟨*proof*⟩
lemmas *abstract-val-id-unit-ptr* = *abstract-val-id* [**where** *a*= *a*::*unit ptr* **and** *P* = *True*] **for** *a*

lemma *abstract-val-id-True*: *abstract-val True a id a*
⟨*proof*⟩

lemmas *abs-var-id-unit-ptr* = *abs-var-id* [**where** *a*= *a*::*unit ptr*] **for** *a*

lemma *len-of-word-comparisons* [*L2opt*]:
len-of TYPE(*64*) ≤ *len-of TYPE*(*64*)
len-of TYPE(*32*) ≤ *len-of TYPE*(*64*)
len-of TYPE(*16*) ≤ *len-of TYPE*(*64*)
len-of TYPE(*8*) ≤ *len-of TYPE*(*64*)
len-of TYPE(*32*) ≤ *len-of TYPE*(*32*)
len-of TYPE(*16*) ≤ *len-of TYPE*(*32*)
len-of TYPE(*8*) ≤ *len-of TYPE*(*32*)
len-of TYPE(*16*) ≤ *len-of TYPE*(*16*)
len-of TYPE(*8*) ≤ *len-of TYPE*(*16*)
len-of TYPE(*8*) ≤ *len-of TYPE*(*8*)

len-of TYPE(*32*) < *len-of TYPE*(*64*)
len-of TYPE(*16*) < *len-of TYPE*(*64*)
len-of TYPE(*8*) < *len-of TYPE*(*64*)
len-of TYPE(*16*) < *len-of TYPE*(*32*)
len-of TYPE(*8*) < *len-of TYPE*(*32*)
len-of TYPE(*8*) < *len-of TYPE*(*16*)

len-of TYPE('a::len signed) = len-of TYPE('a)
(len-of TYPE('a) = len-of TYPE('a)) = True
⟨*proof*⟩

lemma *sbintrunc-eq*: *0 ≤ i ⟹ i < 2^n ⟹ sbintrunc n i = i*
⟨*proof*⟩

lemma *uint-ucast*:
uint (ucast x :: ('a :: len) word) = uint x mod 2 ^ LENGTH('a)
⟨*proof*⟩

lemma *uint-scast'*:
uint (SCAST('a::len → 'b::len) c) = sint c mod 2^LENGTH('b)
⟨*proof*⟩

lemma *uint-ucast'*:
uint (UCAST('a::len → 'b::len) c) = uint c mod 2^LENGTH('b)
⟨*proof*⟩

lemma *sint-ucast'*:
LENGTH('a) < LENGTH('b) ⟹ sint (UCAST('a::len → 'b::len) c) = uint c
⟨*proof*⟩

lemma *scast-ucast-eq-ucast* [*simp*, *L2opt*]:
LENGTH('a::len) < LENGTH('b::len) ⟹ LENGTH('b) ≤ LENGTH('c::len)
⟹
SCAST('b → 'c) (UCAST('a → 'b) x) = UCAST ('a → 'c) x
⟨*proof*⟩

lemma *scast-ucast-simps* [*simp*, *L2opt*]:
⟦ len-of TYPE('b) ≤ len-of TYPE('a); len-of TYPE('c) ≤ len-of TYPE('b) ⟧
⟹
(scast (ucast (a :: 'a::len word) :: 'b::len word) :: 'c::len word) = ucast a
⟦ len-of TYPE('c) ≤ len-of TYPE('a); len-of TYPE('c) ≤ len-of TYPE('b) ⟧
⟹
(scast (ucast (a :: 'a::len word) :: 'b::len word) :: 'c::len word) = ucast a
⟦ len-of TYPE('a) ≤ len-of TYPE('b); len-of TYPE('c) ≤ len-of TYPE('b) ⟧
⟹
(scast (ucast (a :: 'a::len word) :: 'b::len word) :: 'c::len word) = ucast a
⟦ len-of TYPE('a) ≤ len-of TYPE('b) ⟧ ⟹
(scast (scast (a :: 'a::len word) :: 'b::len word) :: 'c::len word) = scast a
⟦ len-of TYPE('b) ≤ len-of TYPE('a); len-of TYPE('c) ≤ len-of TYPE('b) ⟧
⟹
(ucast (scast (a :: 'a::len word) :: 'b::len word) :: 'c::len word) = scast a
⟦ len-of TYPE('c) ≤ len-of TYPE('a); len-of TYPE('c) ≤ len-of TYPE('b) ⟧
⟹
(ucast (scast (a :: 'a::len word) :: 'b::len word) :: 'c::len word) = scast a
⟦ len-of TYPE('a) ≤ len-of TYPE('b); len-of TYPE('c) ≤ len-of TYPE('b) ⟧

$\Longrightarrow$
(*ucast* (*scast* (*a* :: *'a*::*len word*) :: *'b*::*len word*) :: *'c*::*len word*) = *scast a*
⟦ *len-of TYPE*(*'c*) ≤ *len-of TYPE*(*'b*) ⟧ ⟹
(*ucast* (*ucast* (*a* :: *'a*::*len word*) :: *'b*::*len word*) :: *'c*::*len word*) = *ucast a*
⟦ *len-of TYPE*(*'a*) ≤ *len-of TYPE*(*'b*) ⟧ ⟹
(*ucast* (*ucast* (*a* :: *'a*::*len word*) :: *'b*::*len word*) :: *'c*::*len word*) = *ucast a*
⟦ *len-of TYPE*(*'a*) ≤ *len-of TYPE*(*'b*) ⟧ ⟹
(*scast* (*scast* (*a* :: *'a*::*len word*) :: *'b*::*len word*) :: *'c*::*len word*) = *scast a*
⟨*proof*⟩

declare *len-signed* [*L2opt*]

lemmas [*L2opt*] = *zero-sle-ucast-up*

lemma *zero-sle-ucast-WORD-MAX* [*L2opt*]:
(*0* <=*s* ((*ucast* (*b*::(*'a*::*len*) *word*)) :: (*'a*::*len*) *signed word*))
= (*uint b* ≤ *WORD-MAX* (*TYPE*(*'a*)))
⟨*proof*⟩

lemmas [*L2opt*] =
is-up is-down unat-ucast-upcast sint-ucast-eq-uint

lemmas [*L2opt*] =
ucast-down-add scast-down-add
ucast-down-minus scast-down-minus
ucast-down-mult scast-down-mult

lemma *eq-trivI*: *x* = *y* ⟹ *x* = *y*
⟨*proof*⟩

lemmas [*word-abs*] =
corresTA-L2-gets
corresTA-L2-modify
corresTA-L2-throw
corresTA-L2-skip
corresTA-L2-fail
corresTA-L2-seq
corresTA-L2-seq-unit
corresTA-L2-catch
corresTA-L2-try'

corresTA-L2-while
corresTA-L2-guard
corresTA-L2-guarded-simple
corresTA-L2-spec

corresTA-L2-assume
corresTA-L2-condition
corresTA-L2-unknown
corresTA-case-prod
corresTA-L2-call-exec-concrete'
corresTA-L2-call-exec-concrete
corresTA-L2-call-exec-abstract'
corresTA-L2-call-exec-abstract
corresTA-L2-call'
corresTA-L2-call
corresTA-call-L1

lemmas [*word-abs*] =
abstract-val-tuple
abstract-val-Inl
abstract-val-Inr
abstract-val-conj
abstract-val-disj
abstract-val-case-prod
abstract-val-trivial
abstract-val-of-int
abstract-val-of-nat
abstract-val-call-L1-args

lemmas *abs-var-rules* =
abstract-val-call-L1-arg
abstract-val-abs-var-sint-unat
abstract-val-abs-var-uint-unat
abstract-val-abs-var-give-up
abstract-val-abs-var-concretise
abstract-val-abs-var

lemmas *word-abs-base* [*word-abs*] =
valid-typ-abs-fn-id [**where** *'a='a::c-type*]
valid-typ-abs-fn-id [**where** *'a=bool*]
valid-typ-abs-fn-id [**where** *'a='gx c-exntype*]
valid-typ-abs-fn-tuple-split
valid-typ-abs-fn-tuple
valid-typ-abs-fn-sum
valid-typ-abs-fn-unit
valid-typ-abs-fn-sint
valid-typ-abs-fn-unat
len-of-word-comparisons

lemmas *word-abs-sword* =
abstract-val-signed-ops

abstract-val-scast
abstract-val-scast-upcast
abstract-val-scast-downcast
abstract-val-unwrap [**where** *f*=*sint*]
introduce-typ-abs-fn [**where** *f*=*sint* :: (*sword64* ⇒ *int*)]
introduce-typ-abs-fn [**where** *f*=*sint* :: (*sword32* ⇒ *int*)]
introduce-typ-abs-fn [**where** *f*=*sint* :: (*sword16* ⇒ *int*)]
introduce-typ-abs-fn [**where** *f*=*sint* :: (*sword8* ⇒ *int*)]

lemmas *word-abs-word* =
abstract-val-unsigned-ops
abstract-val-uint
abstract-val-ucast
abstract-val-ucast-upcast
abstract-val-ucast-downcast
abstract-val-unwrap [**where** *f*=*unat*]
introduce-typ-abs-fn [**where** *f*=*unat* :: (*word64* ⇒ *nat*)]
introduce-typ-abs-fn [**where** *f*=*unat* :: (*word32* ⇒ *nat*)]
introduce-typ-abs-fn [**where** *f*=*unat* :: (*word16* ⇒ *nat*)]
introduce-typ-abs-fn [**where** *f*=*unat* :: (*word8* ⇒ *nat*)]

lemmas *word-abs-default* =
introduce-typ-abs-fn [**where** *f*=*id* :: ('*a*::*c-type* ⇒ '*a*)]
introduce-typ-abs-fn [**where** *f*=*id* :: (*bool* ⇒ *bool*)]
introduce-typ-abs-fn [**where** *f*=*id* :: ('*gx c-exntype* ⇒ '*gx c-exntype*)]
introduce-typ-abs-fn [**where** *f*=*id* :: (*unit* ⇒ *unit*)]
introduce-typ-abs-fn-tuple
introduce-typ-abs-fn-sum

thm *word-abs*

lemma *int-bounds-to-nat-boundsF*: (*n*::*int*) < *numeral B* ⟹ *0* ≤ *n* ⟹ *nat n* < *numeral B*
⟨*proof*⟩

lemma *int-bounds-one-to-nat*: (*n*::*int*) < *1* ⟹ *0* ≤ *n* ⟹ *nat n* = *0*
⟨*proof*⟩

lemma *id-map-prod-unfold*: *id* = *map-prod id id*
⟨*proof*⟩

lemma *id-tuple-unfold*: *id* = (λ(*x*, *y*). (*id x*, *id y*))
⟨*proof*⟩

end

theory *WordPolish*
imports *WordAbstract*
begin

definition [*simplified*]: *LONG-MAX* ≡ (*2* :: *int*) ^ *63* − *1*
definition [*simplified*]: *LONG-MIN* ≡ − ((*2* :: *int*) ^ *63*)
definition [*simplified*]: *ULONG-MAX* ≡ (*2* :: *nat*) ^ *64* − *1*

definition [*simplified*]: *INT-MAX* ≡ (*2* :: *int*) ^ *31* − *1*
definition [*simplified*]: *INT-MIN* ≡ − ((*2* :: *int*) ^ *31*)
definition [*simplified*]: *UINT-MAX* ≡ (*2* :: *nat*) ^ *32* − *1*

definition [*simplified*]: *SHORT-MAX* ≡ (*2* :: *int*) ^ *15* − *1*
definition [*simplified*]: *SHORT-MIN* ≡ − ((*2* :: *int*) ^ *15*)
definition [*simplified*]: *USHORT-MAX* ≡ (*2* :: *nat*) ^ *16* − *1*

definition [*simplified*]: *CHAR-MAX* ≡ (*2* :: *int*) ^ *7* − *1*
definition [*simplified*]: *CHAR-MIN* ≡ − ((*2* :: *int*) ^ *7*)
definition [*simplified*]: *UCHAR-MAX* ≡ (*2* :: *nat*) ^ *8* − *1*

lemma *WORD-MAX-simps*:
WORD-MAX TYPE(*64*) = *LONG-MAX*
WORD-MAX TYPE(*32*) = *INT-MAX*
WORD-MAX TYPE(*16*) = *SHORT-MAX*
WORD-MAX TYPE(*8*) = *CHAR-MAX*
⟨*proof*⟩

lemma *WORD-MIN-simps*:
WORD-MIN TYPE(*64*) = *LONG-MIN*
WORD-MIN TYPE(*32*) = *INT-MIN*
WORD-MIN TYPE(*16*) = *SHORT-MIN*
WORD-MIN TYPE(*8*) = *CHAR-MIN*
⟨*proof*⟩

lemma *UWORD-MAX-simps*:
UWORD-MAX TYPE(*64*) = *ULONG-MAX*
UWORD-MAX TYPE(*32*) = *UINT-MAX*
UWORD-MAX TYPE(*16*) = *USHORT-MAX*
UWORD-MAX TYPE(*8*) = *UCHAR-MAX*
⟨*proof*⟩

lemma *MIN-MAX-lemmas-schema*:
sint (*s*::*'a*::*len signed word*) ≤ *WORD-MAX TYPE*(*'a*)
WORD-MIN TYPE(*'a*) ≤ *sint* (*s*::*'a*::*len signed word*)
unat (*u*::*'a*::*len word*) ≤ *UWORD-MAX TYPE*(*'a*)

$\neg$ (*sint* (*s*::'*a*::*len signed word*) > *WORD-MAX TYPE*('*a*))
$\neg$ (*WORD-MIN TYPE*('*a*) > *sint* (*s*::'*a*::*len signed word*))
$\neg$ (*unat* (*u*::'*a*::*len word*) > *UWORD-MAX TYPE*('*a*))
WORD-MIN TYPE('*a*) $\leq$ *WORD-MAX TYPE*('*a*)
0 $\leq$ *WORD-MAX TYPE*('*a*)
WORD-MIN TYPE('*a*) $\leq$ *0*
⟨*proof*⟩

lemmas *MIN-MAX-lemmas-pre* =
MIN-MAX-lemmas-schema[**where** '*a*=*64*]
MIN-MAX-lemmas-schema[**where** '*a*=*32*]
MIN-MAX-lemmas-schema[**where** '*a*=*16*]
MIN-MAX-lemmas-schema[**where** '*a*=*8*]

lemmas *INT-MIN-MAX-lemmas* [*simp*] =
MIN-MAX-lemmas-pre[*unfolded WORD-MAX-simps WORD-MIN-simps UWORD-MAX-simps*]

end

Chapter 23

TS phase: Type Strengthening (find suitable target monad)

theory *Refines-Spec*
imports
Option-MonadND
L2ExceptionRewrite
begin

lemma *gets-the-ocondition*:
Spec-Monad.gets-the (*ocondition C T F*) =
Spec-Monad.condition C (*Spec-Monad.gets-the T*) (*Spec-Monad.gets-the F*)
⟨*proof*⟩

lemma *gets-the-oreturn*:
Spec-Monad.gets-the (*oreturn x*) = *Spec-Monad.return x*
⟨*proof*⟩

lemma *gets-the-obind*:
Spec-Monad.gets-the (*obind f g*) =
Spec-Monad.bind (*Spec-Monad.gets-the f*) (λ*x*. *Spec-Monad.gets-the* (*g x*))
⟨*proof*⟩

⟨*ML*⟩

lemma *rel-map-the-Result*[*simp*]: *rel-map the-Result* (*Result v*) *b* ⟷ *v* = *b*
⟨*proof*⟩

lemma *holds-partial-post-state-Inf*:

assumes X: $\bigwedge x.\ x \in X \Longrightarrow$ *holds-partial-post-state* $P\ x$
shows *holds-partial-post-state* $P\ (\bigsqcap\ X)$
⟨*proof*⟩

lemma *holds-post-state-Inf*:
assumes X: $\bigwedge x.\ x \in X \Longrightarrow$ *holds-post-state* $P\ x$
shows $X \neq \{\} \Longrightarrow$ *holds-post-state* $P\ (\bigsqcap\ X)$
⟨*proof*⟩

lemma *admissible-runs-to*[*corres-admissible*]:
ccpo.admissible Inf $(\lambda x\ y.\ y \leq x)$ $(\lambda x.\ x \bullet s\ \{\!|\ Q\ |\!\})$
⟨*proof*⟩

lemma *spec-monad-Inf-run*: $(run\ (\bigsqcap A)\ s) = \bigsqcap\ ((\lambda f.\ (run\ f\ s))\ `\ A)$
⟨*proof*⟩

lemma *outcomes-Inf-run-succeeds-conv*:
$outcomes\ (\bigsqcap f{\in}A.\ run\ f\ t) =\ \ outcomes\ (\bigsqcap f \in \{f.\ f \in A \wedge succeeds\ f\ t\}.\ run\ f\ t)$
⟨*proof*⟩

lemma *succeeds-outcomes-Inf-Inter-conv*:
$g \in A \Longrightarrow succeeds\ g\ t \Longrightarrow outcomes\ (\bigsqcap f \in \{f \in A.\ succeeds\ f\ t\}.\ run\ f\ t) =$
$(\bigcap f{\in}\{f \in A.\ succeeds\ f\ t\}.\ outcomes\ (run\ f\ t))$
⟨*proof*⟩

lemma *admissible-refines-spec-fun-of-rel*:
assumes *prj*: *fun-of-rel R prj*
shows *ccpo.admissible Inf* $(\geq)$
$(\lambda(A{::}('e{::}default,\ 'a,\ 's)\ spec\text{-}monad)\ .\ refines\ C\ A\ s\ t\ (rel\text{-}prod\ R\ (=)))$
⟨*proof*⟩

lemma *admissible-refines-spec-funp*:
assumes *funp R*
shows *ccpo.admissible Inf* $(\geq)$
$(\lambda(A{::}('e{::}default,\ 'a,\ 's)\ spec\text{-}monad)\ .\ refines\ C\ A\ s\ t\ (rel\text{-}prod\ R\ (=)))$
⟨*proof*⟩

lemma *admissible-refines-spec-res*[*corres-admissible*]:
shows *ccpo.admissible Inf* $(\geq)$
$(\lambda(A{::}('a,\ 's)\ res\text{-}monad)\ .\ refines\ C\ A\ s\ t\ (rel\text{-}prod\ (rel\text{-}liftE\ )\ (=)))$
⟨*proof*⟩

lemma *admissible-refines-spec-exit-eq*[*corres-admissible*]:
shows *ccpo.admissible Inf* $(\geq)$
$(\lambda(A{::}('e,\ 'a,\ 's)\ exn\text{-}monad)\ .\ refines\ C\ A\ s\ t\ (rel\text{-}prod\ (rel\text{-}xval\ (=)\ (=))\ (=)))$
⟨*proof*⟩

lemma *admissible-refines-spec-exit-the-Nonlocal*[*corres-admissible*]:
shows *ccpo.admissible Inf* (≥)
(λ(*A*::('*e*, '*a*, '*s*) *exn-monad*) . *refines C A s t* (*rel-prod* ((*rel-xval* (*rel-Nonlocal*) (=))) (=)))
⟨*proof*⟩

lemma *gen-admissible-refines-gets-the*[*corres-admissible*]:
shows *ccpo.admissible option.lub-fun option.le-fun* (λ*A*. *refines C* (*gets-the A*) *s t* (*rel-prod* (*rel-liftE*) (=)))
⟨*proof*⟩

theorem *refines-option-top* [*corres-top*]: *refines f* (*gets-the Map.empty*) *s t R*
⟨*proof*⟩

23.1 Synthesize Rules Setup

Canonical format for the currently supported monads: *refines C* (*lift-to-spec A*) *s t* (*rel-prod rel-res* (=)) where *lift-spec*, *rel-res* is

- pure (Pure function): *return*, *rel-liftE*
- gets (Reader monad): *gets*,*rel-liftE*
- option (Option monad): *gets-the*,*rel-liftE*
- nondet: *id* (ommitted),*rel-liftE*
- exit:
  - *id* (ommitted),*rel-xval rel-Nonlocal* (=)
  - *id* (ommitted),*rel-xval* (=) (=) for those function that were lifted by the IO phase.

synthesize-rules *pure* **and** *reader* **and** *option* **and** *nondet* **and** *exit*

⟨*ML*⟩

Recursive functions are defined using **fixed-point**. The option, nondet and exit monad are setup to handle these definitions. Hence the minimal monad for recursive functions is the option monad.

23.2 Pure Monad

lemma *refines-L2-call-embed-pure*:
assumes *f*: *refines f* (*return f'*) *s s* (*rel-prod rel-liftE* (=))

shows *refines* (*L2-call f emb ns*) (*return* (*L2-VARS f′ ns*)::('*a*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-gets-pure*:
refines (*L2-gets* (λ-. *v*) *ns*) (*return* (*L2-VARS v ns*)::('*a*,'*s*) *res-monad*) *s s* (*rel-prod* (*rel-liftE*) (=))
⟨*proof*⟩

lemma *refines-L2-seq-pure*:
assumes *g* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (⋀*v t*. *refines* (*g v*) (*return* (*g′ v*)::('*a*,'*s*) *res-monad*) *t t* (*rel-prod rel-liftE* (=)))
assumes *f* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (*Trueprop* (*refines f* ((*return f′*)::('*b*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-seq f g*) (*return* (*let v* = *f′ in* (*g′ v*))::('*a*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-condition-pure*:
assumes *g*: *refines g* (*return g′*::('*a*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
assumes *f*: *refines f* (*return f′*::('*a*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-condition* (λ-. *c*) *f g*) (*return* (*if c then f′ else g′*)::('*a*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-try-L2-throw-pure*:
shows *refines* (*L2-try* (*L2-throw* (*Inr r*) *ns*)) (*return* (*L2-VARS r ns*)::('*a*,'*s*)*res-monad*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-try-L2-seq-pure*:
assumes *g*: ⋀*v t*. *refines* (*L2-try* (*g v*)) (*return* (*g′ v*)::('*a*,'*s*) *res-monad*) *t t* (*rel-prod rel-liftE* (=))
assumes *f*: *refines f* (*return f′*::('*b*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-try* (*L2-seq f g*)) (*return* (*let v* = *f′ in* (*g′ v*))::('*a*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-catch-pure*:
assumes *f*: *refines f* (*return f′*::('*b*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-catch f g*) (*return f′*::('*b*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-try-L2-condition-pure*:
assumes *f*: *refines* (*L2-try f*) (*return f′*::('*a*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE*

(=))
assumes *g*: *refines* (*L2-try g*) (*return g'*::('*a*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-try* (*L2-condition* (λ-. *c*) *f g*)) (*return* (*if c then f' else g'*)::('*a*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-try-pure*:
assumes *f*: *refines f* (*return f'*::('*a*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-try f*) (*return f'*::('*a*,'*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

print-synthesize-rules *pure*

lemmas *refines-monad-pure* =
refines-L2-call-embed-pure [*synthesize-rule pure* **and** *reader* **and** *option* **and** *nondet* **and** *exit priority*:*510*]
refines-L2-gets-pure [*synthesize-rule pure* **and** *reader* **and** *option* **and** *nondet* **and** *exit priority*:*510*]
refines-L2-seq-pure [*synthesize-rule pure* **and** *reader* **and** *option* **and** *nondet* **and** *exit priority*:*510 split*: *g* **and** *g'*]
refines-L2-condition-pure [*synthesize-rule pure* **and** *reader* **and** *option* **and** *nondet* **and** *exit priority*:*510*]
refines-L2-catch-pure [*synthesize-rule pure* **and** *reader* **and** *option* **and** *nondet* **and** *exit priority*:*510*]
refines-L2-try-L2-throw-pure [*synthesize-rule pure* **and** *reader* **and** *option* **and** *nondet* **and** *exit priority*:*520*]
refines-L2-try-L2-seq-pure [*synthesize-rule pure* **and** *reader* **and** *option* **and** *nondet* **and** *exit priority*:*520 split*: *g* **and** *g'*]
refines-L2-try-L2-condition-pure [*synthesize-rule pure* **and** *reader* **and** *option* **and** *nondet* **and** *exit priority*:*520*]
refines-L2-try-pure [*synthesize-rule pure* **and** *reader* **and** *option* **and** *nondet* **and** *exit priority*:*510*]

print-synthesize-rules *pure*

23.3 Reader Monad (Gets)

lemma *refines-L2-call-embed-reader*:
assumes *f*: *refines f* (*gets f'*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-call f emb ns*) (*gets* (*L2-VARS f' ns*)) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-gets-reader*:
refines (*L2-gets f ns*) (*gets* (*L2-VARS f ns*)) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-lift-pure-reader*:
assumes *f*: *refines f* (*return f'*) *s s* (*rel-prod rel-liftE* (=))
shows *refines f* (*gets* (λ-. *f'*)) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-seq-reader*:
assumes *g*: ⋀*v t*. *refines* (*g v*) (*gets* (*g' v*)) *t t* (*rel-prod rel-liftE* (=))
assumes *f*: *refines f* (*gets f'*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-seq f g*) (*gets* (λ*s*. *let v* = *f' s in* (*g' v s*))) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-condition-reader*:
assumes *g* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines g* (*gets g'*) *s s* (*rel-prod rel-liftE* (=))))
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines f* (*gets f'*) *s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-condition c f g*) (*gets* (λ*s*. *if c s then f' s else g' s*)) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-try-L2-throw-reader*:
shows *refines* (*L2-try* (*L2-throw* (*Inr r*) *ns*)) (*gets* (*L2-VARS* (λ-. *r*) *ns*)) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-try-L2-seq-reader*:
assumes *g* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (⋀*v t*. *refines* (*L2-try* (*g v*)) (*gets* (*g' v*)) *t t* (*rel-prod rel-liftE* (=)))
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines f* (*gets f'*) *s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-try* (*L2-seq f g*)) (*gets* (λ*s*. *let v* = *f' s in* (*g' v s*))) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-catch-reader*:
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines f* (*gets f'*) *s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-catch f g*) (*gets f'*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-try-L2-condition-reader*:
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines* (*L2-try f*) (*gets f'*) *s s* (*rel-prod rel-liftE* (=))))
assumes *g* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines* (*L2-try g*) (*gets g'*) *s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-try* (*L2-condition* (*c*) *f g*)) (*gets* (λ*s*. *if c s then f' s else g' s*)) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-try-reader*:
assumes *f*: *refines f* (*gets f'*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-try f*) (*gets f'*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemmas *refines-monad-reader* =
refines-L2-call-embed-reader [*synthesize-rule reader* **and** *option* **and** *nondet* **and** *exit priority*:*410*]
refines-L2-gets-reader [*synthesize-rule reader* **and** *option* **and** *nondet* **and** *exit priority*:*410*]
refines-lift-pure-reader [*synthesize-rule reader* **and** *option* **and** *nondet* **and** *exit priority*:*440*]
refines-L2-seq-reader [*synthesize-rule reader* **and** *option* **and** *nondet* **and** *exit priority*:*410 split*: *g* **and** *g'*]
refines-L2-condition-reader [*synthesize-rule reader* **and** *option* **and** *nondet* **and** *exit priority*:*410*]
refines-L2-catch-reader [*synthesize-rule reader* **and** *option* **and** *nondet* **and** *exit priority*:*410*]
refines-L2-try-L2-throw-reader [*synthesize-rule reader* **and** *option* **and** *nondet* **and** *exit priority*:*420*]
refines-L2-try-L2-seq-reader [*synthesize-rule reader* **and** *option* **and** *nondet* **and** *exit priority*:*420 split*: *g* **and** *g'*]
refines-L2-try-L2-condition-reader [*synthesize-rule reader* **and** *option* **and** *nondet* **and** *exit priority*:*420*]
refines-L2-try-reader [*synthesize-rule reader* **and** *option* **and** *nondet* **and** *exit priority*:*410*]

23.4 Option (Reader) Monad

lemma *refines-lift-reader-option*:
assumes *f*: *refines f* (*gets f'*) *s s* (*rel-prod rel-liftE* (=))
shows *refines f* (*gets-the* (*ogets f'*)) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-call-embed-option*:
assumes *f*: *refines f* (*gets-the f'*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-call f emb ns*) (*gets-the* (*L2-VARS f' ns*)) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-gets-option*:
refines (*L2-gets v ns*) (*gets-the* (*L2-VARS* (*ogets v*) *ns*)) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-seq-option*:
assumes *g* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (⋀*v t*. *refines* (*g v*) (*gets-the* (*g' v*)) *t t* (*rel-prod rel-liftE* (=)))

assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines f* (*gets-the f′*) *s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-seq f g*) (*gets-the* (*f′* |>> *g′*)) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-condition-option*:
assumes *g* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines g* (*gets-the g′*) *s s* (*rel-prod rel-liftE* (=))))
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines f* (*gets-the f′*) *s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-condition c f g*) (*gets-the* (*ocondition c f′ g′*)) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-try-L2-throw-option*:
shows *refines* (*L2-try* (*L2-throw* (*Inr r*) *ns*)) (*gets-the* (*L2-VARS* (*oreturn r*) *ns*)) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-try-L2-seq-option*:
assumes *g* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (⋀*v t*. *refines* (*L2-try* (*g v*)) (*gets-the* (*g′ v*)) *t t* (*rel-prod rel-liftE* (=)))
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines f* (*gets-the f′*) *s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-try* (*L2-seq f g*)) (*gets-the* (*f′* |>> *g′*)) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-try-L2-condition-option*:
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines* (*L2-try f*) (*gets-the f′*) *s s* (*rel-prod rel-liftE* (=))))
assumes *g* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines* (*L2-try g*) (*gets-the g′*) *s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-try* (*L2-condition c f g*)) (*gets-the* (*ocondition c f′ g′*)) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-try-option*:
assumes *f*: *refines f* (*gets-the f′*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-try f*) (*gets-the f′*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *le-whileLoop-succeeds-terminatesI*:
assumes ⋀*s*. *run* (*whileLoop C B r*) *s* ≠ ⊤ ⟹ *run f s* ≤ *run* (*whileLoop C B r*) *s*
shows *f* ≤ *whileLoop C B r*
⟨*proof*⟩

lemma *gets-the-whileLoop*:
fixes $C :: 'a \Rightarrow 's \Rightarrow bool$
shows *whileLoop C* ($\lambda a.$ *gets-the* (*B a*)) *r* =
((*gets-the* (*owhile C B r*))::('e::*default*, 'a, 's) *spec-monad*)
⟨*proof*⟩

lemma *refines-L2-while-option*:
assumes *f* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* ($\bigwedge v\ t.$ *refines* (*b v*) (*gets-the* (*b' v*)) *t t* (*rel-prod rel-liftE* (=)))
shows *refines* (*L2-while c b i ns*) (*gets-the* (*L2-VARS* (*owhile c b' i*) *ns*)) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-fail-option*:
shows *refines L2-fail* (*gets-the ofail*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-guard-option*:
shows *refines* (*L2-guard g*) (*gets-the* (*oguard g*)) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-guarded*:
assumes *f*: *g s* $\Longrightarrow$ *refines f* (*gets-the f'*) *s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-guarded g f*) (*gets-the* (*oguard g* |>> (λ-. *f'*))) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemmas *refines-monad-option* =
refines-lift-reader-option [*synthesize-rule option priority*:*350*]
refines-L2-call-embed-option [*synthesize-rule option priority*:*310*]
refines-L2-gets-option [*synthesize-rule option priority*:*310*]

refines-L2-seq-option [*synthesize-rule option priority*:*310 split*: *g* **and** *g'*]
refines-L2-condition-option [*synthesize-rule option priority*:*310*]
refines-L2-try-L2-throw-option [*synthesize-rule option priority*:*320*]
refines-L2-try-L2-seq-option [*synthesize-rule option priority*:*320 split*: *g* **and** *g'*]
refines-L2-try-L2-condition-option [*synthesize-rule option priority*:*320*]

refines-L2-try-option [*synthesize-rule option priority*:*310*]
refines-L2-while-option [*synthesize-rule option priority*:*310 split*: *b* **and** *b'*]
refines-L2-fail-option [*synthesize-rule option priority*:*310*]
refines-L2-guard-option [*synthesize-rule option priority*:*310*]
refines-L2-guarded [*synthesize-rule option priority*:*310*]

23.5 Nondet Monad

Note that *L2-catch* is already replaced by *L2-try* during exception rewriting in phase L2.

lemma (**in** *heap-state*) *refines-IO-modify-heap-paddingE-nondet*:
refines (*IO-modify-heap-paddingE p v*) (*IO-modify-heap-padding p v*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-call-embed-nondet*:
assumes *f*: *refines f f' s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-call f emb ns*) (*L2-VARS f' ns*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-call-embed-exn*:
assumes *f*: *refines f f' s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-call f emb ns*) (*liftE* (*L2-VARS f' ns*)) *s s* (*rel-prod* (*rel-xval L* (=)) (=))
⟨*proof*⟩

lemma *refines-liftE-exn*:
assumes *f*: *refines f f' s s* (*rel-prod rel-liftE* (=))
shows *refines f* (*liftE f'*) *s s* (*rel-prod* (*rel-xval L* (=)) (=))
⟨*proof*⟩

lemma *refines-L2-seq-nondet-polymorphic*:
assumes *r* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (⋀*v t*. *refines* (*g v*) (*g' v*) *t t* (*rel-prod rel-liftE* (=)))
assumes *f* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (*Trueprop* (*refines f f' s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-seq f g*) (*bind f' g'*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-seq-nondet*:
fixes *f'*:: ('*a*, '*s*) *res-monad*
assumes *r* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (⋀*v t*. *refines* (*g v*) (*g' v*) *t t* (*rel-prod rel-liftE* (=)))
assumes *f* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (*Trueprop* (*refines f f' s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-seq f g*) ((*bind f' g'*)::('*b*, '*s*) *res-monad*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-seq-exn*:
assumes *g* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (⋀*v t*. *refines* (*g v*) (*g' v*) *t t* (*rel-prod* (*rel-xval L* (=)) (=)))
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines f f' s s* (*rel-prod*

(*rel-xval L* (=)) (=))))
shows *refines* (*L2-seq f g*) (*bind f′ g′*) *s s* (*rel-prod* (*rel-xval L* (=)) (=))
⟨*proof*⟩

lemma *refines-try-bind-rel-liftE′*:
assumes *g* : ⋀*v s t*. *S s t* ⟹ *refines* (*try* (*g v*)) (*g′ v*) *s t* (*rel-prod rel-liftE S*)
assumes *f*: *refines f f′ s t* (*rel-prod rel-liftE S*)
shows *refines* (*try* (*bind f g*)) (*bind f′ g′*) *s t* (*rel-prod rel-liftE S*)
⟨*proof*⟩

lemma *refines-try-bind-rel-liftE*:
assumes *g* : ⋀*v t*. *refines* (*try* (*g v*)) (*g′ v*) *t t* (*rel-prod rel-liftE* (=))
assumes *f*: *refines f f′ s s* (*rel-prod rel-liftE* (=))
shows *refines* (*try* (*bind f g*)) (*bind f′ g′*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-try-L2-seq-nondet*:
assumes *g* [*unfolded THIN-def L2-defs*, *rule-format*]: *PROP THIN* (⋀*v t*. *refines* (*L2-try* (*g v*)) (*g′ v*) *t t* (*rel-prod rel-liftE* (=)))
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines f f′ s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-try* (*L2-seq f g*)) (*bind f′ g′*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-try-L2-condition-nondet*:
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines* (*L2-try f*) *f′ s s* (*rel-prod rel-liftE* (=))))
assumes *g* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines* (*L2-try g*) *g′ s s* (*rel-prod rel-liftE* (=))))
shows *refines* (*L2-try* (*L2-condition c f g*)) (*condition c f′ g′*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-try-rel-LiftE-nondet*:
assumes *f*: *refines f f′ s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-try f*) *f′ s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-catch-rel-LiftE-nondet*:
assumes *f*: *refines f f′ s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-catch f g*) *f′ s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-try-finally-rel-liftE*:
assumes *f*: *refines f f′ s s* (*rel-prod* (*rel-xval rel-liftE′* (=)) (=))
shows *refines* (*try f*) (*finally f′*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-try-finally-nondet*:
assumes *f*: *refines f f′ s s* (*rel-prod* (*rel-xval rel-liftE′* (=)) (=))
shows *refines* (*L2-try f*) (*finally f′*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-try-exn*:
assumes *f*: *refines f f′ s s* (*rel-prod* (*rel-xval* (*rel-sum L* (=)) (=)) (=))
shows *refines* (*L2-try f*) (*try f′*) *s s* (*rel-prod* (*rel-xval L* (=)) (=))
⟨*proof*⟩

lemma *refines-L2-condition-nondet*:
assumes *g* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines g g′ s s* (*rel-prod V* (=))))
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines f f′ s s* (*rel-prod V* (=))))
shows *refines* (*L2-condition c f g*) (*condition c f′ g′*) *s s* (*rel-prod V* (=))
⟨*proof*⟩

lemma *refines-L2-while-nondet*:
assumes *b* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (⋀*v t*. *refines* (*b v*) (*b′ v*) *t t* (*rel-prod rel-liftE* (=)))
shows *refines* (*L2-while c b i ns*) (*L2-VARS* (*whileLoop c b′ i*) *ns*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-while-exn*:
assumes *b* [*unfolded THIN-def*, *rule-format*]: *PROP THIN* (⋀*v t*. *refines* (*b v*) (*b′ v*) *t t* (*rel-prod* (*rel-xval L* (=)) (=)))
shows *refines* (*L2-while c b i ns*) (*L2-VARS* (*whileLoop c b′ i*) *ns*) *s s* (*rel-prod* ((*rel-xval L* (=))) (=))
⟨*proof*⟩

lemma *refines-L2-unknown-nondet*:
shows *refines* (*L2-unknown ns*) (*L2-VARS* (*select UNIV*) *ns*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-unknown-exn*:
shows *refines* (*L2-unknown ns*) (*L2-VARS* (*select UNIV*) *ns*) *s s* (*rel-prod* (*rel-xval L* (=)) (=))
⟨*proof*⟩

lemma *refines-L2-modify-nondet*:
shows *refines* (*L2-modify f*) (*modify f*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-gets-nondet*:
shows *refines* (*L2-gets f ns*) (*L2-VARS* (*gets f*) *ns*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-throw-exn*:
L x x′ ⟹
refines (*L2-throw x ns*) (*L2-VARS* (*throw x′*) *ns*) *s s* (*rel-prod* (*rel-xval L* (=)) (=))
⟨*proof*⟩

lemma *refines-L2-spec-nondet*:
shows *refines* (*L2-spec r*) (*assert-result-and-state* (λ*s*. {(*v*, *t*). (*s*, *t*) ∈ *r*})) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-assume-nondet*:
shows *refines* (*L2-assume r*) (*assume-result-and-state r*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-guard-nondet*:
shows *refines* (*L2-guard c*) (*guard c*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-guarded-nondet*:
assumes *f*: *c s* ⟹ *refines f f′ s s* (*rel-prod rel-liftE* (=))
shows *refines* (*L2-guarded c f*) (*bind* (*guard c*) (λ-. *f′*)) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-guarded-exn*:
assumes *f*: *c s* ⟹ *refines f f′ s s* (*rel-prod* (*rel-xval L* (=)) (=))
shows *refines* (*L2-guarded c f*) (*bind* (*guard c*) (λ-. *f′*)) *s s* (*rel-prod* (*rel-xval L* (=)) (=))
⟨*proof*⟩

lemma *refines-L2-fail-nondet*:
shows *refines L2-fail fail s s* (*rel-prod R* (=))
⟨*proof*⟩

lemma *refines-exec-concrete-gen-nondet*:
assumes *f*: ⋀*s*. *refines f f′ s s* (*rel-prod R* (=))
shows *refines* (*exec-concrete st f*) (*exec-concrete st f′*) *s s* (*rel-prod R* (=))
⟨*proof*⟩

lemmas *refines-exec-concrete-nondet* = *refines-exec-concrete-gen-nondet* [**where**

R=*rel-liftE*]
lemmas *refines-exec-concrete-exit* = *refines-exec-concrete-gen-nondet* [**where** *R*=*rel-xval L* (=)] **for** *L*

lemma *refines-exec-abstract-gen-nondet*:
assumes *f*: ⋀*s*. *refines f f' s s* (*rel-prod R* (=))
shows *refines* (*exec-abstract st f*) (*exec-abstract st f'*) *s s* (*rel-prod R* (=))
⟨*proof*⟩

lemmas *refines-exec-abstract-nondet* = *refines-exec-abstract-gen-nondet* [**where** *R*=*rel-liftE*]
lemmas *refines-exec-abstract-exit* = *refines-exec-abstract-gen-nondet* [**where** *R*=*rel-xval L* (=)] **for** *L*

lemma *rel-map-ResultI*:
rel-map Result x (*Result x*)
⟨*proof*⟩

lemma *rel-map-to-xval-InlI*:
rel-map to-xval (*Inl l*) (*Exn l*)
⟨*proof*⟩

lemma *rel-map-to-xval-InrI*:
rel-map to-xval (*Inr r*) (*Result r*)
⟨*proof*⟩

lemma *refines-rel-prod-eq-guard-on-exit*:
assumes *f*: *refines* f_c f_a *s s* (*rel-prod Q* (=))
shows *refines*
(*guard-on-exit* f_c *grd cleanup*)
(*guard-on-exit* f_a *grd cleanup*) *s s*
(*rel-prod Q* (=))
⟨*proof*⟩

lemma *refines-rel-prod-eq-assume-on-exit*:
assumes *f*: *refines* f_c f_a *s s* (*rel-prod Q* (=))
shows *refines*
(*assume-on-exit* f_c *grd cleanup*)
(*assume-on-exit* f_a *grd cleanup*) *s s*
(*rel-prod Q* (=))
⟨*proof*⟩

context *stack-heap-state*
begin

thm *refines-rel-prod-with-fresh-stack-ptr*
lemma *refines-rel-prod-L2-try-with-fresh-stack-ptr*:
assumes *init-eq*: $init_c$ *s* = $init_a$ *s*

assumes f: $\bigwedge s\ p.\ refines\ (L2\text{-}try\ (f_c\ p))\ (f_a\ p)\ s\ s\ (rel\text{-}prod\ Q\ (=))$
shows
refines
$(L2\text{-}try\ (with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init_c\ (L2\text{-}VARS\ f_c\ nm)))$
$(with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init_a\ (L2\text{-}VARS\ f_a\ nm))\ s\ s$
$(rel\text{-}prod\ Q\ (=))$
⟨*proof*⟩
end

context *typ-heap-typing*
begin

lemma *refines-rel-prod-guard-with-fresh-stack-ptr*:
assumes *init-eq*: $init_c\ s = init_a\ s$
assumes f: $\bigwedge s\ p.\ refines\ (f_c\ p)\ (f_a\ p)\ s\ s\ (rel\text{-}prod\ L\ (=))$
shows
refines
$(guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init_c\ (L2\text{-}VARS\ f_c\ nm))$
$(guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init_a\ (L2\text{-}VARS\ f_a\ nm))\ s\ s$
$(rel\text{-}prod\ L\ (=))$
⟨*proof*⟩

lemma *refines-rel-prod-assume-with-fresh-stack-ptr*:
assumes *init-eq*: $init_c\ s = init_a\ s$
assumes f: $\bigwedge s\ p.\ refines\ (f_c\ p)\ (f_a\ p)\ s\ s\ (rel\text{-}prod\ L\ (=))$
shows
refines
$(assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init_c\ (L2\text{-}VARS\ f_c\ nm))$
$(assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init_a\ (L2\text{-}VARS\ f_a\ nm))\ s\ s$
$(rel\text{-}prod\ L\ (=))$
⟨*proof*⟩

lemma *refines-rel-prod-with-fresh-stack-ptr*:
assumes *init-eq*: $init_c\ s = init_a\ s$
assumes f: $\bigwedge s\ p.\ refines\ (f_c\ p)\ (f_a\ p)\ s\ s\ (rel\text{-}prod\ L\ (=))$
shows
refines
$(with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init_c\ (L2\text{-}VARS\ f_c\ nm))$
$(with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init_a\ (L2\text{-}VARS\ f_a\ nm))\ s\ s$
$(rel\text{-}prod\ L\ (=))$
⟨*proof*⟩

lemma *refines-rel-prod-L2-try-guard-with-fresh-stack-ptr*:
assumes *init-eq*: $init_c\ s = init_a\ s$
assumes f: $\bigwedge s\ p.\ refines\ (L2\text{-}try\ (f_c\ p))\ (f_a\ p)\ s\ s\ (rel\text{-}prod\ L\ (=))$
shows
refines
$(L2\text{-}try\ (guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init_c\ (L2\text{-}VARS\ f_c\ nm)))$

(*guard-with-fresh-stack-ptr n init$_a$ (L2-VARS f$_a$ nm)) s s*
(*rel-prod L (=)*)
⟨*proof*⟩

lemma *refines-rel-prod-L2-try-assume-with-fresh-stack-ptr*:
assumes *init-eq*: *init$_c$ s = init$_a$ s*
assumes *f*: ⋀*s p*. *refines* (*L2-try* (*f$_c$ p*)) (*f$_a$ p*) *s s* (*rel-prod L* (=))
shows
refines
(*L2-try* (*assume-with-fresh-stack-ptr n init$_c$* (*L2-VARS f$_c$ nm*)))
(*assume-with-fresh-stack-ptr n init$_a$* (*L2-VARS f$_a$ nm*)) *s s*
(*rel-prod L* (=))
⟨*proof*⟩

lemma *refines-rel-prod-L2-try-with-fresh-stack-ptr*:
assumes *init-eq*: *init$_c$ s = init$_a$ s*
assumes *f*: ⋀*s p*. *refines* (*L2-try* (*f$_c$ p*)) (*f$_a$ p*) *s s* (*rel-prod L* (=))
shows
refines
(*L2-try* (*with-fresh-stack-ptr n init$_c$* (*L2-VARS f$_c$ nm*)))
(*with-fresh-stack-ptr n init$_a$* (*L2-VARS f$_a$ nm*)) *s s*
(*rel-prod L* (=))
⟨*proof*⟩

end

lemma *relcomppI-swapped*: *s b c* ⟹ *r a b* ⟹ (*r OO s*) *a c*
⟨*proof*⟩

lemmas *refines-monad-nondet* =
refines-L2-call-embed-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-L2-call-embed-exn [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-liftE-exn [*synthesize-rule nondet* **and** *exit priority*:*250*]
refines-L2-seq-nondet [*synthesize-rule nondet* **and** *exit priority*:*210 split*: *g* **and** *g'*]
refines-L2-seq-exn [*synthesize-rule nondet* **and** *exit priority*:*210 split*: *g* **and** *g'*]

refines-L2-catch-rel-LiftE-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-L2-try-L2-seq-nondet [*synthesize-rule nondet* **and** *exit priority*:*230 split*: *g* **and** *g'*]
refines-L2-try-L2-condition-nondet [*synthesize-rule nondet* **and** *exit priority*:*230*]
refines-L2-try-rel-LiftE-nondet [*synthesize-rule nondet* **and** *exit priority*:*220*]
refines-L2-try-finally-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-L2-try-exn [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-L2-condition-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-L2-while-nondet [*synthesize-rule nondet* **and** *exit priority*:*210 split*: *b* **and** *b′*]
refines-L2-while-exn [*synthesize-rule nondet* **and** *exit priority*:*210 split*: *b* **and** *b′*]

refines-L2-unknown-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-L2-unknown-exn [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-L2-modify-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-L2-gets-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-L2-throw-exn [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-L2-spec-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-L2-assume-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-L2-guard-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-L2-guarded-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-L2-guarded-exn [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-L2-fail-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-exec-concrete-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-exec-concrete-exit [*synthesize-rule nondet* **and** *exit priority*:*210*]

refines-exec-abstract-nondet [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-exec-abstract-exit [*synthesize-rule nondet* **and** *exit priority*:*210*]

rel-liftE-trivial [*synthesize-rule nondet* **and** *exit priority*:*210*]
rel-liftE′-Inr [*synthesize-rule nondet* **and** *exit priority*:*210*]
rel-sum-Inl [*synthesize-rule nondet* **and** *exit priority*:*210*]
rel-sum-Inr [*synthesize-rule nondet* **and** *exit priority*:*210*]
rel-xval-Exn [*synthesize-rule nondet* **and** *exit priority*:*210*]
rel-xval-Result [*synthesize-rule nondet* **and** *exit priority*:*210*]

relcomppI-swapped [*synthesize-rule nondet* **and** *exit priority*:*210*]
rel-map-ResultI [*synthesize-rule nondet* **and** *exit priority*:*210*]
rel-map-to-xval-InlI [*synthesize-rule nondet* **and** *exit priority*:*210*]
rel-map-to-xval-InrI [*synthesize-rule nondet* **and** *exit priority*:*210*]

refl [*synthesize-rule nondet* **and** *exit priority*: *210*]

context *typ-heap-typing*
begin

lemmas *refines-monad-nondet* =
refines-rel-prod-L2-try-guard-with-fresh-stack-ptr [*synthesize-rule nondet* **and** *exit priority*:*232*]
refines-rel-prod-L2-try-assume-with-fresh-stack-ptr [*synthesize-rule nondet* **and** *exit priority*:*232*]
refines-rel-prod-L2-try-with-fresh-stack-ptr [*synthesize-rule nondet* **and** *exit priority*:*232*]
refines-rel-prod-guard-with-fresh-stack-ptr [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-rel-prod-assume-with-fresh-stack-ptr [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-rel-prod-with-fresh-stack-ptr [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-rel-xval-guard-with-fresh-stack-ptr [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-rel-xval-assume-with-fresh-stack-ptr [*synthesize-rule nondet* **and** *exit priority*:*210*]
refines-rel-xval-with-fresh-stack-ptr [*synthesize-rule nondet* **and** *exit priority*:*210*]
end

context *stack-heap-state*
begin
lemmas *refines-nondet-monad* =
refines-rel-prod-L2-try-with-fresh-stack-ptr [*synthesize-rule nondet* **and** *exit priority*:*232*]
refines-rel-prod-with-fresh-stack-ptr [*synthesize-rule nondet* **and** *exit priority*:*210*]
end

context *heap-state*
begin
lemmas *refines-nondet-monad'* =
refines-IO-modify-heap-paddingE-nondet[*synthesize-rule nondet* **and** *exit priority*:*210*]
end
⟨*ML*⟩

23.5.1 Elimination of *L2-try* in the Error Monad

Eliminate Inner Exception

rules for elimination of *L2-try* when the inner exception layer is not needed, i.e., *rel-sum* (*rel-throwE L*) (=)

lemma *bind-bind-exception-or-result-conv*:
(*f* ≫= *g*) = (*bind-exception-or-result f* (λ*Exception e* ⇒ *yield* (*Exception e*) | *Result v* ⇒ *g v*))
⟨*proof*⟩

lemma *rel-throwE'-rel-throwE-conv*: *rel-throwE' L* = (*rel-map to-xval OO rel-throwE L*)
⟨*proof*⟩

lemma *rel-throwE* (*rel-throwE'* *L*) = (*rel-throwE* (*rel-map to-xval OO rel-throwE* *L*))
⟨*proof*⟩

lemma *refines-L2-try-L2-seq-exn*:
fixes *f*:: ((*'b* + *'c*), *'f*, *'a*) *exn-monad*
and *g*::*'f* ⇒ ((*'b* + *'c*), *'c*, *'a*) *exn-monad*
assumes *g* [*unfolded THIN-def L2-defs*, *rule-format*]:
PROP THIN (⋀*v t*. *refines* (*L2-try* (*g v*)) (*g'* *v*) *t t* (*rel-prod* ((*rel-xval L R*)) (=)))
assumes *f* [*unfolded THIN-def*]:
PROP THIN (*Trueprop* (*refines f f' s s* (*rel-prod* (*rel-xval* (*rel-throwE'* *L*) (=)) (=))))
shows *refines* (*L2-try* (*L2-seq f g*)) (*bind f' g'*) *s s* (*rel-prod* (*rel-xval L R*) (=))
⟨*proof*⟩

lemma *refines-L2-try-L2-condition-exit*:
assumes *f* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines* (*L2-try f*) *f'* *s s* (*rel-prod R* (=))))
assumes *g* [*unfolded THIN-def*]: *PROP THIN* (*Trueprop* (*refines* (*L2-try g*) *g'* *s s* (*rel-prod R* (=))))
shows *refines* (*L2-try* (*L2-condition c f g*)) (*condition c f' g'*) *s s* (*rel-prod R* (=))
⟨*proof*⟩

lemma *refines-try-rel-xval-rel-throwE'*:
assumes *refines f f' s s* (*rel-prod* (*rel-xval* (*rel-throwE'* *A*) *B*) *S*)
shows *refines* (*try f*) *f' s s* (*rel-prod* (*rel-xval A B*) *S*)
⟨*proof*⟩

lemma *refines-L2-try-rel-sum-rel-throwE-nondet*:
assumes *refines f f' s s* (*rel-prod* (*rel-xval* (*rel-throwE'* *A*) *B*) (=))
shows *refines* (*L2-try f*) *f' s s* (*rel-prod* (*rel-xval A B*) (=))
⟨*proof*⟩

lemma *refines-try-rel-throwE*:
assumes *refines f f' s s* (*rel-prod* (*rel-throwE* (*rel-throwE'* *L*)) *S*)
shows *refines* (*try f*) *f' s s* (*rel-prod* (*rel-throwE L*) *S*)
⟨*proof*⟩

lemma *refines-L2-try-rel-throwE-nondet*:
assumes *refines f f' s s* (*rel-prod* (*rel-throwE* (*rel-throwE'* *L*)) (=))
shows *refines* (*L2-try f*) *f' s s* (*rel-prod* (*rel-throwE L*) (=))
⟨*proof*⟩

lemmas *ts-L2-try-inner-exception* =
rel-throwE'-Inl[*synthesize-rule nondet* **and** *exit*]
rel-throwE-Exn[*synthesize-rule nondet* **and** *exit*]

refines-L2-try-L2-seq-exn [*synthesize-rule nondet* **and** *exit priority*: *218 split*: *g* **and** *g'*]
refines-L2-try-L2-condition-exit [*synthesize-rule nondet* **and** *exit priority*: *218*]
refines-L2-try-rel-throwE-nondet [*synthesize-rule nondet* **and** *exit priority*: *216*]
refines-L2-try-rel-sum-rel-throwE-nondet [*synthesize-rule nondet* **and** *exit priority*: *215*]
bundle *del-ts-L2-try-inner-exception* =
rel-throwE'-Inl[*synthesize-rule nondet* **and** *exit del*]
rel-throwE-Exn[*synthesize-rule nondet* **and** *exit del*]
refines-L2-try-L2-seq-exn [*synthesize-rule nondet* **and** *exit priority*: *218 split*: *g* **and** *g' del*]
refines-L2-try-L2-condition-exit [*synthesize-rule nondet* **and** *exit priority*: *218 del*]
refines-L2-try-rel-throwE-nondet [*synthesize-rule nondet* **and** *exit priority*: *216 del*]
refines-L2-try-rel-sum-rel-throwE-nondet [*synthesize-rule nondet* **and** *exit priority*: *215 del*]

print-synthesize-rules *exit* ‹*Trueprop* (*refines* (*L2-seq* - -) - - - (*rel-prod rel-liftE* (=))) ›

Eliminate *L2-try* over exiting branches

Eliminate *L2-try* over a condition, when one branch always exits (*rel-throwE* (*rel-sum L R*))

lemma *rel-map-to-xval-rel-xval-rel-sum-conv*:
rel-map to-xval OO rel-xval L R = *rel-sum L R OO rel-map to-xval*
⟨*proof*⟩

lemma *refines-L2-try-L2-seq-L2-condition-exit1*:
assumes *g* [*unfolded THIN-def*]:
PROP THIN (*Trueprop* (*refines* (*L2-try* (*L2-seq g h*)) *gh' s s* (*rel-prod LR* (=))))
assumes *f* [*unfolded THIN-def*]:
PROP THIN (*Trueprop* (*refines f f' s s* (*rel-prod* (*rel-throwE* (*rel-map to-xval OO LR*)) (=))))
shows *refines* (*L2-try* (*L2-seq* (*L2-condition c f g*) *h*)) (*condition c f' gh'*) *s s* (*rel-prod LR* (=))
⟨*proof*⟩

lemma *refines-L2-try-L2-seq-L2-condition-exit2*:
assumes *f* [*unfolded THIN-def*]:
PROP THIN (*Trueprop* (*refines* (*L2-try* (*L2-seq f h*)) *fh' s s* (*rel-prod LR* (=))))
assumes *g* [*unfolded THIN-def*]:
PROP THIN (*Trueprop* (*refines g g' s s* (*rel-prod* (*rel-throwE* (*rel-map to-xval OO LR*)) (=))))
shows *refines* (*L2-try* (*L2-seq* (*L2-condition c f g*) *h*)) (*condition c fh' g'*) *s s* (*rel-prod LR* (=))
⟨*proof*⟩

lemma *refines-L2-seq-L2-condition-rel-throwE1*:
assumes *g* [*unfolded THIN-def*]:
PROP THIN (*Trueprop* (*refines* (*L2-seq g h*) *gh' s s* (*rel-prod* (*rel-throwE LR*)(=))))
assumes *f* [*unfolded THIN-def*]:
PROP THIN (*Trueprop* (*refines f f' s s* (*rel-prod* (*rel-throwE LR*) (=))))
shows *refines* (*L2-seq* (*L2-condition c f g*) *h*) (*condition c f' gh'*) *s s* (*rel-prod* (*rel-throwE LR*) (=))
⟨*proof*⟩

lemma *refines-L2-seq-L2-condition-rel-throwE2*:
assumes *f* [*unfolded THIN-def*]:
PROP THIN (*Trueprop* (*refines* (*L2-seq f h*) *fh' s s* (*rel-prod* (*rel-throwE LR*) (=))))
assumes *g* [*unfolded THIN-def*]:
PROP THIN (*Trueprop* (*refines g g' s s* (*rel-prod* (*rel-throwE LR*) (=))))
shows *refines* (*L2-seq* (*L2-condition c f g*) *h*) (*condition c fh' g'*) *s s* (*rel-prod* (*rel-throwE LR*) (=))
⟨*proof*⟩

lemma *refines-bind-rel-throwE-first*:
assumes *f*: *refines f f' s s* (*rel-prod* (*rel-throwE LR*) *S*)
shows *refines* (*f* >>= *g*) *f' s s* (*rel-prod* (*rel-throwE LR*) *S*)
⟨*proof*⟩

lemma *refines-L2-seq-rel-throwE-throwE*:
assumes *f* [*unfolded THIN-def*]:
PROP THIN (*Trueprop* (*refines f f' s s* (*rel-prod* (*rel-throwE LR*) (=))))
shows *refines* (*L2-seq f g*) *f' s s* (*rel-prod* (*rel-throwE LR*) (=))
⟨*proof*⟩

lemma *refines-L2-seq-rel-throwE-throwE1*:
assumes *g*[*unfolded THIN-def*, *rule-format*] :
PROP THIN (⋀*v t*. *refines* (*g v*) (*g' v*) *t t* (*rel-prod* (*rel-throwE* (*rel-map to-xval OO rel-xval L R*)) (=)))
assumes *f* [*unfolded THIN-def*, *rule-format*]:
PROP THIN (*Trueprop* (*refines f f' s s* (*rel-prod* (*rel-xval* (*rel-throwE' L*) (=)) (=))))
shows *refines* (*L2-seq f g*) (*bind f' g'*) *s s* (*rel-prod* (*rel-throwE* (*rel-map to-xval OO rel-xval L R*)) (=))
⟨*proof*⟩

lemma *refines-L2-seq-rel-throwE-liftE*:
assumes *g* [*unfolded THIN-def*, *rule-format*]:
PROP THIN (⋀*v t*. *refines* (*g v*) (*g' v*) *t t* (*rel-prod* (*rel-throwE LR*) (=)))
assumes *f* [*unfolded THIN-def*]:
PROP THIN (*Trueprop* (*refines f f' s s* (*rel-prod rel-liftE* (=))))

shows *refines* (*L2-seq f g*) (*bind f′ g′*) *s s* (*rel-prod* (*rel-throwE LR*) (=))
⟨*proof*⟩

lemma *refines-L2-throw-rel-throwE-Inl*:
assumes *L l l′*
shows *refines* (*L2-throw* (*Inl l*) *ns*) (*throw* (*L2-VARS l′ ns*)) *s s* (*rel-prod* (*rel-throwE* (*rel-map to-xval OO rel-xval L R*)) (=))
⟨*proof*⟩

lemma *refines-L2-throw-rel-throwE-Inr*:
assumes *R r r′*
shows *refines* (*L2-throw* (*Inr r*) *ns*) (*return* (*L2-VARS r′ ns*)) *s s* (*rel-prod* (*rel-throwE* ((*rel-map to-xval OO rel-xval L R*))) (=))
⟨*proof*⟩

lemma *refines-L2-throw-rel-throwE*:
assumes *R r* (*Result r′*)
shows *refines* (*L2-throw r ns*) (*L2-VARS* (*return r′*) *ns*) *s s* (*rel-prod* (*rel-throwE R*) (=))
⟨*proof*⟩

lemmas *ts-L2-try-condition-exit* =

refines-L2-try-L2-seq-L2-condition-exit1[*synthesize-rule nondet* **and** *exit priority*:*217*]
refines-L2-try-L2-seq-L2-condition-exit2[*synthesize-rule nondet* **and** *exit priority*:*217*]

refines-L2-seq-L2-condition-rel-throwE1[*synthesize-rule nondet* **and** *exit priority*:*216*]
— Note that 1. this together with *refines ?f ?f′ ?s ?s* (*rel-prod* (*rel-throwE* (*rel-throwE′ ?L*)) (=)) ⟹ *refines* (*L2-try ?f*) *?f′ ?s ?s* (*rel-prod* (*rel-throwE ?L*) (=)) does not replace 2. ⟦*THIN* (*refines* (*L2-try* (*L2-seq ?g ?h*)) *?gh′ ?s ?s* (*rel-prod ?LR* (=))); *THIN* (*refines ?f ?f′ ?s ?s* (*rel-prod* (*rel-throwE* (*rel-project to-xval OO ?LR*)) (=)))⟧ ⟹ *refines* (*L2-try* (*L2-seq* (*L2-condition ?c ?f ?g*) *?h*)) (*condition ?c*
?f′
?gh′) *?s ?s* (*rel-prod ?LR* (=)) and vice versa. 1 is more compositional and also works deeply nested but only handles the case *rel-throwE*, whereas 2 also handles *rel-liftE* and *rel-sum* by pushing down *L2-try* into the branch of the conditional.
refines-L2-seq-L2-condition-rel-throwE2[*synthesize-rule nondet* **and** *exit priority*:*216*]
refines-L2-seq-rel-throwE-throwE [*synthesize-rule nondet* **and** *exit priority*:*213*]
refines-L2-seq-rel-throwE-throwE1[*synthesize-rule nondet* **and** *exit priority*:*212 split*: *g* **and** *g′*]
refines-L2-seq-rel-throwE-liftE [*synthesize-rule nondet* **and** *exit priority*:*212 split*: *g* **and** *g′*]
refines-L2-throw-rel-throwE-Inl [*synthesize-rule nondet* **and** *exit priority*:*212*]

refines-L2-throw-rel-throwE-Inr [*synthesize-rule nondet* **and** *exit priority*:*212*]
refines-L2-throw-rel-throwE [*synthesize-rule nondet* **and** *exit priority*:*212*]
bundle *del-ts-L2-try-condition-exit* =
refines-L2-try-L2-seq-L2-condition-exit1[*synthesize-rule nondet* **and** *exit priority*:*217 del*]
refines-L2-try-L2-seq-L2-condition-exit2[*synthesize-rule nondet* **and** *exit priority*:*217 del*]
refines-L2-seq-L2-condition-rel-throwE1[*synthesize-rule nondet* **and** *exit priority*:*216 del*]
refines-L2-seq-L2-condition-rel-throwE2[*synthesize-rule nondet* **and** *exit priority*:*216 del*]
refines-L2-seq-rel-throwE-throwE [*synthesize-rule nondet* **and** *exit priority*:*213 del*]
refines-L2-seq-rel-throwE-throwE1[*synthesize-rule nondet* **and** *exit priority*:*212 split*: *g* **and** *g′ del*]
refines-L2-seq-rel-throwE-liftE [*synthesize-rule nondet* **and** *exit priority*:*212 split*: *g* **and** *g′ del*]
refines-L2-throw-rel-throwE-Inl [*synthesize-rule nondet* **and** *exit priority*:*212 del*]
refines-L2-throw-rel-throwE-Inr [*synthesize-rule nondet* **and** *exit priority*:*212 del*]
refines-L2-throw-rel-throwE [*synthesize-rule nondet* **and** *exit priority*:*212 del*]

23.6 Error Monad (exit)

lemma *refines-L2-call-embed-exit*:
assumes *f*: *refines f f′ s s* (*rel-prod* (*rel-xval rel-Nonlocal* (=)) (=)) (=))
assumes *emb*: ⋀*e′*. *L* (*emb* (*Nonlocal e′*)) (*emb′ e′*)
shows *refines* (*L2-call f emb ns*) (*L2-VARS* (*map-value* (*map-exn emb′*) *f′*) *ns*) *s s* (*rel-prod* (*rel-xval L* (=)) (=))
⟨*proof*⟩

lemma *refines-L2-call-embed-exit-in-out*:
assumes *emb*: ⋀*e′*. *L* (*emb e′*) (*emb′ e′*)
assumes *f*: *refines f f′ s s* (*rel-prod* (*rel-xval* (=) (=)) (=))
shows *refines* (*L2-call f emb ns*) (*L2-VARS* (*map-value* (*map-exn emb′*) *f′*) *ns*) *s s* (*rel-prod* (*rel-xval L* (=)) (=))
⟨*proof*⟩

lemma *refines-L2-catch-exit*:
assumes *f*: *refines f f′ s s* (*rel-prod* (*rel-xval* (=) *R*) (=))
assumes *h*: ⋀*s′ v*. *refines* (*h v*) (*h′ v*) *s′ s′* (*rel-prod* (*rel-xval L R*) (=))
shows *refines* (*L2-catch f h*) (*catch f′ h′*) *s s* (*rel-prod* (*rel-xval L R*) (=))
⟨*proof*⟩

lemma *rel-xval-eq-refl*: *rel-xval* (=) (=) *x x*
⟨*proof*⟩

lemmas *refines-monad-exit* =

refines-L2-call-embed-exit [*synthesize-rule exit priority*:*110*]
refines-L2-call-embed-exit-in-out [*synthesize-rule exit priority*:*109 split*: *emb* **and** *emb'*]
refines-L2-catch-exit [*synthesize-rule exit priority*:*110 split*: *h* **and** *h'*]
rel-Nonlocal-Nonlocal [*synthesize-rule exit priority*:*110*]
rel-sum-eq [*synthesize-rule exit priority*:*210*]
rel-xval-eq-refl [*synthesize-rule exit priority*:*210*]

print-synthesize-rules *exit*

end

theory *TypeStrengthen*
imports
Refines-Spec
begin

⟨*ML*⟩

lemma *gets-the-ogets-return-conv* [*fun-ptr-simps*]: *gets-the* (*ogets* (λ-. *f*)) = *return f*
⟨*proof*⟩

lemma *gets-the-ogets-gets-conv* [*fun-ptr-simps*]: *gets-the* (*ogets f*) = *gets f*
⟨*proof*⟩

lemma *gets-the-ogets*: *gets-the* (*ogets f*) = *gets f*
⟨*proof*⟩

lemma *gets-the-obind*:
gets-the (*f* |>> *g*) = *gets-the f* >>= (λ*x*. *gets-the* (*g x*))
⟨*proof*⟩

lemma *gets-the-oguard*: *gets-the* (*oguard P*) = *guard P*
⟨*proof*⟩

lemma *gets-the-ocondition*:
gets-the (*ocondition P f g*) = *condition P* (*gets-the f*) (*gets-the g*)
⟨*proof*⟩

A best-effort approach to determine the simplest possible 'monad' for the final definition is implemented. We first try to define a function into the most restrictive monad and if that fails successively try more expressive monads until we finally hit the most expressive monad. In the original

autocorres version this phase was based on equations and not on on a simulation relation as all the other autocorres phases. With the switch to model recursive functions with a CCPO **fixed-point** instead of **function** with an explicit measure parameter this did no longer work, as equations are not *ccpo.admissible*. Fortunately simulation is admissible, so we changed this phase to *refines*, cf: `Refines_Spec.thy`. So the main purpose of this theory is to set up the available target monads by applying some meta information: `monad_types.ML`.

The correspondence equations have the format:

(*) p = *L2-call-lift* p'

where *L2-call-lift* depends on the (simpler) target monad and lifts the program p' from that simpler monad to the fully fledged monad we start with:

The program p is the definition we have from the last layer of autocorres (WA). The final definition will refer to p'.

For the code to work *L2-call-lift* has to be a distinct constant, as some matching is performed on that assumption. That is why some new definitions are introduced below.

Note that the final (most expressive) monad is characterised by the lifting function is *lift-exit-status* which merely removes the exception handling artefact from SIMPL by extracting the exception value $'a$ from $'a$ *c-exntype*. So it should be sufficiently expressive for any input C program.

When the proof for a certain monad fails it can either have a good reason (as the input function is just not expressible in that particular monad) or it can fail because the implementation is missing some rules.

Note some peculiarities on the current state of affairs:

- When a guard remains (e.g. bounds for an integer) you end up at least in the option monad (to model the possible failure).
- As recursive functions are currently implemented with **fixed-point** they are at least in the option monad.

⟨*ML*⟩

lemma *monotone-L2-VARS* [*partial-function-mono*]:
monotone R X a ⟹ *monotone R X* (λ*f*. *L2-VARS* (*a f*) *ns*)
⟨*proof*⟩

lemma *monotone-ocondition* [*partial-function-mono*]:
assumes *mono-X*: *monotone R* (*fun-ord Q*) *X*
assumes *mono-Y*: *monotone R* (*fun-ord Q*) *Y*

shows *monotone R* (*fun-ord Q*)
(λ*f*. (*ocondition C* (*X f*) (*Y f*)))
⟨*proof*⟩

declare *Complete-Partial-Order2.option.preorder* [*partial-function-mono*]

lemma *monotone-obind*[*partial-function-mono*]:
monotone R option.le-fun a ⟹ (⋀*x*. *monotone R option.le-fun* (λ*f*. *b f x*)) ⟹
monotone R option.le-fun (λ*f*. *obind* (*a f*) (*b f*))
⟨*proof*⟩

lemma *monotone-option-fun-const* [*partial-function-mono*]:
monotone R option.le-fun (λ*f*. *c*)
⟨*proof*⟩

lemma *option-while-eq-Some*:
option-while C B I = *Some F* ⟷ (*Some I*, *Some F*) ∈ *option-while*′ *C B*
⟨*proof*⟩

lemma *option-while*′*-monotone*:
assumes *B*: ⋀*r*. *flat-ord None* (*B r*) (*B*′ *r*)
assumes *b*: (*a*, *b*) ∈ *option-while*′ *C B b* ≠ *None* **shows** (*a*, *b*) ∈ *option-while*′ *C B*′
⟨*proof*⟩
lemma *monotone-option-while*[*partial-function-mono*]:
assumes *B*: ⋀*a*. *monotone R* (*flat-ord None*) (λ*f*. *B f a*)
shows *monotone R* (*flat-ord None*) (λ*f*. *option-while C* (*B f*) *I*)
⟨*proof*⟩

lemma *monotone-owhile*[*partial-function-mono*]:
(⋀*a*. *monotone R option.le-fun* (λ*f*. *B f a*)) ⟹
monotone R option.le-fun (λ*f*. *owhile C* (*B f*) *I*)
⟨*proof*⟩

⟨*ML*⟩

lemma *refines-lift-pure-option*:
assumes *f*: *refines f* (*return f*′) *s s* (*rel-prod rel-liftE* (=))
shows *refines f* (*gets-the* (*oreturn f*′)) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

⟨*ML*⟩

lemma *id-comps*:
$id\ o\ f = f$
$((\lambda s.\ s)\ o\ f) = f$
⟨*proof*⟩

lemma *gets-bind-ign*: $gets\ f >>= (\lambda x.\ m) = m$
⟨*proof*⟩

end

Chapter 24

Polishing the Final Outcome

theory *Polish*
imports *HeapLift WordPolish TypeStrengthen*
begin

context *heap-typing-state*
begin

lemma *unchanged-typing-bind*[*unchanged-typing*]:
  assumes [*runs-to-vcg*]: $f \cdot s$?{| $\lambda r.$ *unchanged-typing-on S s* |} ($\bigwedge r\ s.\ g\ r \cdot s$?{| $\lambda r.$ *unchanged-typing-on S s* |})
  shows (*bind f g*) $\cdot s$?{| $\lambda r.$ *unchanged-typing-on S s* |}
  ⟨*proof*⟩

lemma *unchanged-typing-while*[*unchanged-typing*]:
  assumes *B*: $\bigwedge r\ s.\ C\ r\ s \Longrightarrow (B\ r) \cdot s$?{| $\lambda r.$ *unchanged-typing-on S s*|}
  shows (*whileLoop C B i*) $\cdot s$?{| $\lambda r.$ *unchanged-typing-on S s*|}
  ⟨*proof*⟩

lemma *unchanged-typing-finally*[*unchanged-typing*]:
  assumes [*runs-to-vcg*]: $f \cdot s$?{| $\lambda r.$ *unchanged-typing-on S s*|}
  shows *finally f* $\cdot s$?{| $\lambda r.$ *unchanged-typing-on S s*|}
  ⟨*proof*⟩

lemma *unchanged-typing-try*[*unchanged-typing*]:
  assumes [*runs-to-vcg*]: $f \cdot s$?{| $\lambda r.$ *unchanged-typing-on S s*|}
  shows *try f* $\cdot s$?{| $\lambda r.$ *unchanged-typing-on S s*|}
  ⟨*proof*⟩

lemma *runs-to-partial-catch*[*unchanged-typing*]:
  assumes [*runs-to-vcg*]: $f \cdot s$?{|$\lambda r.$ *unchanged-typing-on S s*|}
  assumes [*runs-to-vcg*]: $\bigwedge r\ s.\ (g\ r) \cdot s$?{|$\lambda r.$ *unchanged-typing-on S s*|}
  shows (*catch f g*) $\cdot s$?{|$\lambda r.$ *unchanged-typing-on S s*|}
  ⟨*proof*⟩

lemma *runs-to-partial-bind-handle*[*unchanged-typing*]:
assumes [*runs-to-vcg*]: $f \cdot s$ *?*⦃$\lambda r.$ *unchanged-typing-on S s*⦄
assumes [*runs-to-vcg*]: $\bigwedge r\ s.\ (g\ r) \cdot s$ *?*⦃$\lambda r.$ *unchanged-typing-on S s*⦄
assumes [*runs-to-vcg*]: $\bigwedge r\ s.\ (h\ r) \cdot s$ *?*⦃$\lambda r.$ *unchanged-typing-on S s*⦄
shows $(bind\text{-}handle\ f\ g\ h) \cdot s$ *?*⦃$\lambda r.$ *unchanged-typing-on S s*⦄
⟨*proof*⟩

lemma *unchanged-typing-liftE*[*unchanged-typing*]:
assumes [*runs-to-vcg*]: $f \cdot s$ *?*⦃ $\lambda r.$ *unchanged-typing-on S s*⦄
shows *liftE* $f \cdot s$ *?*⦃ $\lambda r.$ *unchanged-typing-on S s*⦄
⟨*proof*⟩

end

context *open-types-heap-typing-state*
begin

lemma *unchanged-typing-ptr-valid-preserved*:
ptr-valid (*heap-typing t1*) *p* ⟹ *unchanged-typing-on UNIV t1 t2* ⟹ *ptr-valid* (*heap-typing t2*) *p*
⟨*proof*⟩

lemma *reaches-unchanged-typing-ptr-valid-preserved*:
reaches f s r t ⟹ *ptr-valid* (*heap-typing s*) *p* ⟹ $f \cdot s$ *?*⦃λ- *t. unchanged-typing-on UNIV s t*⦄ ⟹
ptr-valid (*heap-typing t*) *p*
⟨*proof*⟩
thm *unchanged-typing* [*no-vars*]
end

locale *typ-heap-simulation-open-types-stack* =
typ-heap-simulation-open-types $\mathcal{T}$ *st r w v t-hrs t-hrs-update heap-typing heap-typing-upd*
for
$\mathcal{T}$ **and**
st:: $'s \Rightarrow 't$ **and**
r:: $'t \Rightarrow ('a$::{*xmem-type*, *stack-type*}$)$ *ptr* $\Rightarrow 'a$ **and**
w:: $'a$ *ptr* $\Rightarrow ('a \Rightarrow 'a) \Rightarrow 't \Rightarrow 't$ **and**
v:: $'t \Rightarrow 'a$ *ptr* $\Rightarrow$ *bool* **and**
t-hrs :: $'s \Rightarrow$ *heap-raw-state* **and**
t-hrs-update:: (*heap-raw-state* $\Rightarrow$ *heap-raw-state*) $\Rightarrow 's \Rightarrow 's$ **and**
heap-typing :: $'t \Rightarrow$ *heap-typ-desc* **and**
heap-typing-upd :: (*heap-typ-desc* $\Rightarrow$ *heap-typ-desc*) $\Rightarrow 't \Rightarrow 't$ +
fixes $\mathcal{S}$:: *addr set*
begin
sublocale *typ-heap-typing r w heap-typing heap-typing-upd* $\mathcal{S}$
⟨*proof*⟩

lemma *unchanged-typing-on-with-fresh-stack-ptr*[*unchanged-typing*]:
fixes *f*::*'a ptr* ⇒ (*'e*::*default*, *'b*, *'t*) *spec-monad*
assumes [*runs-to-vcg*]:⋀*p t*. (*f p*) · *t* ?{λ- *t'*. *unchanged-typing-on A t t'*}
shows (*with-fresh-stack-ptr n init f*) · *t* ?{λ- *t'*. *unchanged-typing-on A t t'*}
⟨*proof*⟩

lemma *unchanged-typing-on-assume-with-fresh-stack-ptr*[*unchanged-typing*]:
fixes *f*::*'a ptr* ⇒ (*'e*::*default*, *'b*, *'t*) *spec-monad*
assumes [*runs-to-vcg*]:⋀*p t*. (*f p*) · *t* ?{λ- *t'*. *unchanged-typing-on A t t'*}
shows (*assume-with-fresh-stack-ptr n init f*) · *t* ?{λ- *t'*. *unchanged-typing-on A t t'*}
⟨*proof*⟩

lemma *unchanged-typing-on-guard-with-fresh-stack-ptr*[*unchanged-typing*]:
fixes *f*::*'a ptr* ⇒ (*'e*::*default*, *'b*, *'t*) *spec-monad*
assumes [*runs-to-vcg*]:⋀*p t*. (*f p*) · *t* ?{λ- *t'*. *unchanged-typing-on A t t'*}
shows (*guard-with-fresh-stack-ptr n init f*) · *t* ?{λ- *t'*. *unchanged-typing-on A t t'*}
⟨*proof*⟩

end

named-theorems *polish*
named-theorems *polish-cong* **and** *state-fold-congs*

declare *map-value-id*[*polish*]

lemmas [*polish*] = *mex-def meq-def*

declare *WORD-values-fold* [*polish*]

lemmas *WORD-bound-simps* [*polish*] =
WORD-MAX-simps
WORD-MIN-simps
UWORD-MAX-simps
WORD-signed-to-unsigned
INT-MIN-MAX-lemmas

declare *singleton-iff* [*polish*]
declare *the-Nonlocal.simps* [*polish*]
declare *map-exn-id* [*polish*]
declare *prod.case* [*polish*]

declare *mem-Collect-eq* [*polish*]

lemma *L2-VARS-polish* [*polish*]: *L2-VARS x ns* = *x*

⟨*proof*⟩

lemmas *liftE-atomic* [*polish*] =
liftE-unknown
liftE-top
liftE-bot
liftE-fail
liftE-yield-Exception
liftE-return
liftE-throw-exception-or-result
liftE-get-state
liftE-set-state
liftE-select
liftE-assert
liftE-assume
liftE-gets
liftE-guard
liftE-assert-opt
liftE-gets-the
liftE-modify

lemma *map-value-map-exn-id*[*simp*, *polish*]:
map-value (*map-exn* (λ*e'*. *e'*)) *f* = *f*
⟨*proof*⟩

lemma *gets-to-return* [*polish*]:
gets (λ*x*. *a*) = *return a*
⟨*proof*⟩

lemma *select-UNIV-unknown* [*polish*]: *select UNIV* = *unknown*
⟨*proof*⟩

lemma *unknown-unit* [*polish*, *simp*]: (*unknown* :: (*unit*,'*s*) *res-monad*) = *skip*
⟨*proof*⟩

lemma *condition-to-when* [*polish*]:
condition (λ*s*. *C*) *A skip* = *when C A*
⟨*proof*⟩

lemma *condition-to-unless* [*polish*]:
condition (λ*s*. *C*) *skip A* = *unless C A*
⟨*proof*⟩

lemma *bind-skip* [*simp*, *polish*]:
(*x* >>= (λ-. *skip*)) = *x*
⟨*proof*⟩

lemma *skip-bind* [*simp*, *polish*]:

(*skip* >>= *P*) = (*P* ())
⟨*proof*⟩

lemma *catch-skip*[*simp*, *polish*]: *catch skip f* = *skip*
⟨*proof*⟩

lemma *ogets-to-oreturn* [*polish*]: *ogets* (λ*s*. *P*) = *oreturn P*
⟨*proof*⟩

lemma *ocondition-ret-ret* [*polish*]:
ocondition P (*oreturn A*) (*oreturn B*) = *ogets* (λ*s*. *if P s then A else B*)
⟨*proof*⟩

lemma *ocondition-gets-ret* [*polish*]:
ocondition P (*ogets A*) (*oreturn B*) = *ogets* (λ*s*. *if P s then A s else B*)
⟨*proof*⟩

lemma *ocondition-ret-gets* [*polish*]:
ocondition P (*oreturn A*) (*ogets B*) = *ogets* (λ*s*. *if P s then A else B s*)
⟨*proof*⟩

lemma *ocondition-gets-gets* [*polish*]:
ocondition P (*ogets A*) (*ogets B*) = *ogets* (λ*s*. *if P s then A s else B s*)
⟨*proof*⟩

lemma *case-prod-trivial*[]: *NO-MATCH* (*g*::(*'e*::*default*, *'a*, *'s*) *spec-monad*) *f* ⟹ (λ(*x*,*y*). *f*) = (λ-. *f*)
— We avoid this rule during polish to keep brittle tuple structure (*case-prod*) and bound variable names in eg. (>>=). With the *NO-MATCH* setup the simplification might still trigger e.g. on conditions of while loops
⟨*proof*⟩

lemma *bind-case-prod-trivial*[*polish*]: *bind f* (λ(*x*, *y*). *g*) = *bind f* (λ-. *g*)
⟨*proof*⟩

lemma *condition-to-if* [*polish*]:
condition (λ*s*. *C*) (*return a*) (*return b*) = *return* (*if C then a else b*)
⟨*proof*⟩

lemma *guard-merge-bind*:
guard P >>= (λ-. *guard Q* >>= *M*) = *guard* (*P and Q*) >>= *M*
⟨*proof*⟩

lemma *guard-merge-bind'*:
guard P >>= (λ-. *guard Q* >>= *M*) = *guard* (λ*s*. *P s* ∧ *Q s*) >>= *M*
⟨*proof*⟩

lemma *guard-merge*:
guard P >>= (λ-. guard Q) = guard (P and Q)
⟨*proof*⟩

lemma *guard-merge′*:
guard P >>= (λ-. guard Q) = guard (λs. P s ∧ Q s)
⟨*proof*⟩

lemma *pred-conj-commute*: *(P and Q) = (Q and P)*
⟨*proof*⟩

lemma *guard-True-skip*[*polish*, *simp*]: *guard (λ-. True) = skip*
⟨*proof*⟩

lemma *guard-True-bind*: — subsumed by *skip >>= ?P = ?P* () and *guard (λ-. True) = skip*
(guard (λ-. True) >>= M) = M ()
⟨*proof*⟩

declare *assert-simps*(*1*) [*polish*]
declare *assume-simps*(*1*) [*polish*]

lemma *simple-bind-fail* [*simp*]:
(guard P >>= (λ-. fail)) = fail
(modify f >>= (λ-. fail)) = fail
(return x >>= (λ-. fail)) = fail
(gets f >>= (λ-. fail)) = fail
⟨*proof*⟩

declare *condition-fail-rhs* [*polish*]
declare *condition-fail-lhs* [*polish*]
declare *simple-bind-fail* [*polish*]
declare *condition-bind-fail* [*polish*]

lemma *whileLoop-fail*:
(whileLoop C (λ-. fail) i) = (guard (λs. ¬ C i s) >>= (λ-. return i))
⟨*proof*⟩

lemma *owhile-fail*:
(owhile C (λ-. ofail) i) = (oguard (λs. ¬ C i s) |>> (λ-. oreturn i))
⟨*proof*⟩

declare *whileLoop-fail* [*polish*]
declare *owhile-fail* [*polish*]

lemma *oguard-True* [*simp*, *polish*]: *oguard (λ-. True) = oreturn ()*
⟨*proof*⟩

lemma *oguard-False* [*simp*, *polish*]: *oguard* (λ-. *False*) = *ofail*
⟨*proof*⟩

declare *oreturn-bind* [*polish*]
declare *obind-return* [*polish*]
lemma *infinite-option-while'*: (*Some x*, *Some y*) ∉ *option-while'* (λ-. *True*) *B*
⟨*proof*⟩

lemma *expand-guard-conj* [*polish*]:
guard (λ*s*. *A s* ∧ *B s*) = (*do* {*guard* (λ*s*. *A s*); *guard* (λ*s*. *B s*) })
⟨*proof*⟩

lemma *oguard-K-bind-cong* [*polish-cong*]: *g* = *g'* ⟹ (⋀*s*. *g' s* ⟹ *c s* = *c' s*)
⟹ (*oguard g* >>= (λ-. *c*)) = (*oguard g'* >>= (λ-. *c'*))
⟨*proof*⟩

lemma *oguard-obind-cong*: *g* = *g'* ⟹ (⋀*s*. *g' s* ⟹ *c s* = *c' s*) ⟹
do {- <− *oguard g* ; *c*} = *do* {- <− *oguard g'*; *c'*}
⟨*proof*⟩

lemma *oguard-True-K-bind* [*polish*]: (*oguard* (λ-. *True*) >>= (λ-. *c*)) = *c*
⟨*proof*⟩

lemma *oguard-False-bind* [*polish*]: (*oguard* (λ-. *False*) >>= *c*) = *ofail*
⟨*proof*⟩

lemma *guard-False-bind* [*polish*]: (*guard* (λ-. *False*) >>= *c*) = *fail*
⟨*proof*⟩

lemma *expand-oguard-conj* [*polish*]:
oguard (λ*s*. *A s* ∧ *B s*) = (*obind* (*oguard* (λ*s*. *A s*)) (λ-. *oguard* (λ*s*. *B s*)))
⟨*proof*⟩

lemma *owhile-infinite-loop* [*simp*, *polish*]:
owhile (λ*r s*. *C*) *B x* = (*oguard* (λ*s*. ¬ *C*) |>> (λ-. *oreturn x*))
⟨*proof*⟩

declare *obind-return* [*polish*]
declare *bind-return* [*polish*]

lemma *fail-bind* [*simp*]:
fail >>= *f* = *fail*
⟨*proof*⟩

declare *fail-bind* [*polish*]

declare *ofail-bind* [*polish*]
declare *obind-fail* [*polish*]

declare *singleton-iff* [*polish*]
declare *when-True* [*polish*]
declare *when-False* [*polish*]

lemma *let-triv* [*polish*]: (*let x = y in x*) = *y*
⟨*proof*⟩

lemma *ucast-scast-same* [*polish*, *L2opt*, *simp*]:
ucast ((*scast x* :: (*'a*::*len*) *word*)) = (*x* :: *'a word*)
⟨*proof*⟩

lemma *word-of-int-of-nat* [*polish*, *L2opt*, *simp*]:
word-of-int (*int x*) = *of-nat x*
⟨*proof*⟩

lemma *return-if-P-1-0-bind* [*polish*]:
(*return* (*if P then 1 else 0*) >>= (λ*x*. *Q x*))
= (*return P* >>= (λ*x*. *Q* (*if x then 1 else 0*)))
⟨*proof*⟩

lemma *gets-if-P-1-0-bind* [*polish*]:
(*gets* (λ*s*. *if P s then 1 else 0*) >>= (λ*x*. *Q x*))
= (*gets P* >>= (λ*x*. *Q* (*if x then 1 else 0*)))
⟨*proof*⟩

lemma *if-P-1-0-neq-0* [*polish*, *simp*]:
((*if P then 1 else* (*0*::(*'a*::{*zero-neq-one*}))) ≠ *0*) = *P*
⟨*proof*⟩

lemma *if-P-1-0-eq-0* [*polish*, *simp*]:
((*if P then 1 else* (*0*::(*'a*::{*zero-neq-one*}))) = *0*) = (¬ *P*)
⟨*proof*⟩

lemma *if-if-same-output* [*polish*]:
(*if a then if b then x else y else y*) = (*if a* ∧ *b then x else y*)
(*if a then x else if b then x else y*) = (*if a* ∨ *b then x else y*)
⟨*proof*⟩

lemma *collect-guarded-conj*[*polish*]:
condition C1 (*do* { *guard G*; *gets* (λ*s*. *if C2 s then 1 else 0*) })
(*return 0*)
= *do* { *guard* (λ*s*. *C1 s* ⟶ *G s*);

gets (λs. *if C1 s* $\wedge$ *C2 s then 1 else 0*) }
⟨*proof*⟩

lemma *collect-guarded-disj*[*polish*]:
condition C1 (*return 1*)
(*do* {*guard G*; *gets* (λs. *if C2 s then 1 else 0*) })
= *do* {*guard* (λs. $\neg$ *C1 s* $\longrightarrow$ *G s*);
gets (λs. *if C1 s* $\vee$ *C2 s then 1 else 0*) }
⟨*proof*⟩

lemmas [*polish*] = *bind-assoc obind-assoc*

declare *not-not* [*polish*]

lemma *collect-then-cond-1-0*[*polish*]:
do {*cond* $\leftarrow$ *gets* (λs. *if P s then* (*1*::('*a*::{*zero-neq-one*})) *else 0*);
condition (λ-. *cond* $\neq$ *0*) *L R* }
= *condition P L R*
⟨*proof*⟩

lemma *collect-then-cond-1-0-assoc*[*polish*]:
(*do* {*cond* $\leftarrow$ *gets* (λs. *if P s then* (*1*::('*a*::{*zero-neq-one*})) *else 0*);
condition (λ-. *cond* $\neq$ *0*) *L R*
>>= *f* })
= (*condition P L R* >>= *f*)
⟨*proof*⟩

lemma *bind-return-bind* [*polish*]:
(*A* >>= (λx. (*return x* >>= (λy. *B x y*)))) = (*A* >>= (λx. *B x x*))
⟨*proof*⟩

lemma *obind-oreturn-obind* [*polish*]:
(*A* |>> (λx. (*oreturn x* |>> (λy. *B x y*)))) = (*A* |>> (λx. *B x x*))
⟨*proof*⟩

declare *obind-assoc* [*polish*]

declare *if-distrib* [**where** *f*=*scast*, *polish*, *simp*]
declare *if-distrib* [**where** *f*=*ucast*, *polish*, *simp*]
declare *if-distrib* [**where** *f*=*unat*, *polish*, *simp*]
declare *if-distrib* [**where** *f*=*uint*, *polish*, *simp*]
declare *if-distrib* [**where** *f*=*sint*, *polish*, *simp*]

declare *cast-simps* [*polish*]

lemma *Suc-0-eq-1* [*polish*]: *Suc 0 = 1*
⟨*proof*⟩

lemma *bind-return-case-prod* [*polish*, *simp*]:
(*do* {() ← *A0*; *return* () }) = *A0*
(*do* {(*a*) ← *A1*; *return* (*a*) }) = *A1*
(*do* {(*a*, *b*) ← *A2*; *return* (*a*, *b*) }) = *A2*
(*do* {(*a*, *b*, *c*) ← *A3*; *return* (*a*, *b*, *c*) }) = *A3*
(*do* {(*a*, *b*, *c*, *d*) ← *A4*; *return* (*a*, *b*, *c*, *d*) }) = *A4*
(*do* {(*a*, *b*, *c*, *d*, *e*) ← *A5*; *return* (*a*, *b*, *c*, *d*, *e*) }) = *A5*
(*do* {(*a*, *b*, *c*, *d*, *e*, *f*) ← *A6*; *return* (*a*, *b*, *c*, *d*, *e*, *f*) }) = *A6*
⟨*proof*⟩

lemma *bind-return-case-prod'* [*polish*, *simp*]:
(*A1* ⤜ *return* ⤜ *g1*) = (*A1* ⤜ *g1*)
(*A2* ⤜ (λ(*a*, *b*). (*return* (*a*, *b*) ⤜ *g2*))) = (*A2* ⤜ *g2*)
(*A3* ⤜ (λ(*a*, *b*, *c*). (*return* (*a*, *b*, *c*) ⤜ *g3*))) = (*A3* ⤜ *g3*)
(*A4* ⤜ (λ(*a*, *b*, *c*, *d*). (*return* (*a*, *b*, *c*, *d*) ⤜ *g4*))) = (*A4* ⤜ *g4*)
(*A5* ⤜ (λ(*a*, *b*, *c*, *d*, *e*). (*return* (*a*, *b*, *c*, *d*, *e*) ⤜ *g5*))) = (*A5* ⤜ *g5*)
(*A6* ⤜ (λ(*a*, *b*, *c*, *d*, *e*, *f*). (*return* (*a*, *b*, *c*, *d*, *e*, *f*) ⤜ *g6*))) = (*A6* ⤜ *g6*)
⟨*proof*⟩

lemma *obind-returnOk-prodE-case* [*polish*, *simp*]:
(*A1* |>> (λ*a*. *oreturn* (*a*))) = *A1*
(*A2* |>> (λ(*a*, *b*). *oreturn* (*a*, *b*))) = *A2*
(*A3* |>> (λ(*a*, *b*, *c*). *oreturn* (*a*, *b*, *c*))) = *A3*
(*A4* |>> (λ(*a*, *b*, *c*, *d*). *oreturn* (*a*, *b*, *c*, *d*))) = *A4*
(*A5* |>> (λ(*a*, *b*, *c*, *d*, *e*). *oreturn* (*a*, *b*, *c*, *d*, *e*))) = *A5*
(*A6* |>> (λ(*a*, *b*, *c*, *d*, *e*, *f*). *oreturn* (*a*, *b*, *c*, *d*, *e*, *f*))) = *A6*
⟨*proof*⟩

lemma *obind-returnOk-prodE-case'* [*polish*, *simp*]:
(*A1* |>> (λ*a*. (*oreturn* (*a*) |>> *g1*))) = *A1* |>> *g1*
(*A2* |>> (λ(*a*, *b*). (*oreturn* (*a*, *b*) |>> *g2*))) = *A2* |>> *g2*
(*A3* |>> (λ(*a*, *b*, *c*). (*oreturn* (*a*, *b*, *c*) |>> *g3*))) = *A3* |>> *g3*
(*A4* |>> (λ(*a*, *b*, *c*, *d*). (*oreturn* (*a*, *b*, *c*, *d*) |>> *g4*))) = *A4* |>> *g4*
(*A5* |>> (λ(*a*, *b*, *c*, *d*, *e*). (*oreturn* (*a*, *b*, *c*, *d*, *e*) |>> *g5*))) = *A5* |>> *g5*
(*A6* |>> (λ(*a*, *b*, *c*, *d*, *e*, *f*). (*oreturn* (*a*, *b*, *c*, *d*, *e*, *f*) |>> *g6*))) = *A6* |>> *g6*
⟨*proof*⟩

lemma *bind-fixup-1*:
(*bind A* (λ*x*. *case y of* (*a*, *b*) ⇒ *B a b x*)) =
(*case y of* (*a*, *b*) ⇒ *bind A* (λ*x*. *B a b x*))
⟨*proof*⟩

lemma *bind-fixup-2*:
(*bind* (*case y of* (*a*, *b*) ⇒ *B a b*) *A*) =
(*case y of* (*a*, *b*) ⇒ *bind* (*B a b*) *A*)
⟨*proof*⟩
lemma *finally-fixup*: (λ*x*. *finally* (*case x of* (*a*, *b*) ⇒ *f a b*)) = (λ(*a*,*b*). *finally* (*f a b*))
⟨*proof*⟩
lemma *try-fixup*: (λ*x*. *try* (*case x of* (*a*, *b*) ⇒ *f a b*)) = (λ(*a*,*b*). *try* (*f a b*))
⟨*proof*⟩
lemma *liftE-fixup*: (λ*x*. *liftE* (*case x of* (*a*, *b*) ⇒ *f a b*)) = (λ(*a*,*b*). *liftE* (*f a b*))
⟨*proof*⟩

lemmas *spec-monad-split-fixup* [*polish*]= — should we even take more from *L2-seq ?A* (λ*x*. *case ?y of* (*a*, *b*) ⇒ *?B a b x*) = (*case ?y of* (*a*, *b*) ⇒ *L2-seq ?A* (*?B a b*))
L2-seq (*case ?y of* (*a*, *b*) ⇒ *?B a b*) *?A* = (*case ?y of* (*a*, *b*) ⇒ *L2-seq* (*?B a b*) *?A*)
(*case* (*?x*, *?y*) *of* (*a*, *b*) ⇒ *?P a b*) = *?P ?x ?y*
(λ(*a*, *b*). *?P a*) = (λ(*a*, *x*, *y*). *?P a*)
(λ*x*. *liftE* (*case x of* (*a*, *b*) ⇒ *?f a b*)) = (λ(*a*, *b*). *liftE* (*?f a b*))
(λ*x*. *finally* (*case x of* (*a*, *b*) ⇒ *?f a b*)) = (λ(*a*, *b*). *finally* (*?f a b*))
(λ*x*. *try* (*case x of* (*a*, *b*) ⇒ *?f a b*)) = (λ(*a*, *b*). *try* (*?f a b*))
L2-guard (λ*s*. *case ?y of* (*a*, *b*) ⇒ *?G a b s*) = (*case ?y of* (*a*, *b*) ⇒ *L2-guard* (*?G a b*))
L2-guard (*case ?y of* (*a*, *b*) ⇒ *?G a b*) = (*case ?y of* (*a*, *b*) ⇒ *L2-guard* (*?G a b*))
L2-gets (*case ?y of* (*a*, *b*) ⇒ *?G a b*) = (*case ?y of* (*a*, *b*) ⇒ *L2-gets* (*?G a b*))
L2-modify (*case ?y of* (*a*, *b*) ⇒ *?G a b*) = (*case ?y of* (*a*, *b*) ⇒ *L2-modify* (*?G a b*))
L2-gets (λ-. *case ?y of* (*a*, *b*) ⇒ *?G a b*) = (*case ?y of* (*a*, *b*) ⇒ *L2-gets* (λ-. *?G a b*))
(λ(*a*, *b*). *L2-gets* (*?P a b*) *?n*) = (λ(*a*, *x*, *y*). *L2-gets* (*?P a* (*x*, *y*)) *?n*)
(λ(*a*, *b*). *L2-guard* (*?P a b*)) = (λ(*a*, *x*, *y*). *L2-guard* (*?P a* (*x*, *y*)))
(λ(*a*, *b*). *L2-modify* (*?P a b*)) = (λ(*a*, *x*, *y*). *L2-modify* (*?P a* (*x*, *y*)))
(λ(*a*, *b*). *L2-spec* (*?P a b*)) = (λ(*a*, *x*, *y*). *L2-spec* (*?P a* (*x*, *y*)))
(λ(*a*, *b*). *L2-assume* (*?P a b*)) = (λ(*a*, *x*, *y*). *L2-assume* (*?P a* (*x*, *y*)))
(λ(*a*, *b*). *L2-throw* (*?P a b*) *?n*) = (λ(*a*, *x*, *y*). *L2-throw* (*?P a* (*x*, *y*)) *?n*)
(λ(*a*, *b*). *L2-seq* (*?L a b*) (*?R a b*)) = (λ(*a*, *x*, *y*). *L2-seq* (*?L a* (*x*, *y*)) (*?R a* (*x*, *y*)))
(λ(*a*, *b*). *L2-while* (*?C a b*) (*?B a b*) (*?I a b*) *?n*) = (λ(*a*, *x*, *y*). *L2-while* (*?C a* (*x*, *y*)) (*?B a* (*x*, *y*)) (*?I a* (*x*, *y*)) *?n*)
(λ(*a*, *b*). *L2-unknown ?n*) = (λ(*a*, *x*, *y*). *L2-unknown ?n*)
(λ(*a*, *b*). *L2-catch* (*?L a b*) (*?R a b*)) = (λ(*a*, *x*, *y*). *L2-catch* (*?L a* (*x*, *y*)) (*?R a* (*x*, *y*)))
(λ(*a*, *b*). *L2-condition* (*?C a b*) (*?L a b*) (*?R a b*)) = (λ(*a*, *x*, *y*). *L2-condition* (*?C a* (*x*, *y*)) (*?L a* (*x*, *y*)) (*?R a* (*x*, *y*)))
(λ(*a*, *b*). *L2-call* (*?M a b*)) = (λ(*a*, *x*, *y*). *L2-call* (*?M a* (*x*, *y*)))
(λ(*a*, *b*). *liftE* (*?M a b*)) = (λ(*a*, *x*, *y*). *liftE* (*?M a* (*x*, *y*)))
(λ(*a*, *b*). *finally* (*?M a b*)) = (λ(*a*, *x*, *y*). *finally* (*?M a* (*x*, *y*)))

$(\lambda(a,\ b).\ try\ (?M\ a\ b)) = (\lambda(a,\ x,\ y).\ try\ (?M\ a\ (x,\ y)))$?
bind-fixup-1
bind-fixup-2
finally-fixup
try-fixup
liftE-fixup

lemma *scast-1-simps* [*simp*, *L2opt*, *polish*]:
scast (*1* :: (*'a*::*len*) *bit1 word*) = *1*
scast (*1* :: (*'a*::*len*) *bit0 word*) = *1*
scast (*1* :: (*'a*::*len*) *bit1 signed word*) = *1*
scast (*1* :: (*'a*::*len*) *bit0 signed word*) = *1*
⟨*proof*⟩

lemma *scast-1-simps-direct* [*simp*, *L2opt*, *polish*]:
scast (*1* :: *sword64*) = (*1* :: *word64*)
scast (*1* :: *sword64*) = (*1* :: *word32*)
scast (*1* :: *sword64*) = (*1* :: *word16*)
scast (*1* :: *sword64*) = (*1* :: *word8*)
scast (*1* :: *sword32*) = (*1* :: *word32*)
scast (*1* :: *sword32*) = (*1* :: *word16*)
scast (*1* :: *sword32*) = (*1* :: *word8*)
scast (*1* :: *sword16*) = (*1* :: *word16*)
scast (*1* :: *sword16*) = (*1* :: *word8*)
scast (*1* :: *sword8*) = (*1* :: *word8*)
⟨*proof*⟩

declare *scast-0* [*L2opt*, *polish*]

declare *Word.sint-0* [*polish*]
declare *More-Word.sint-0* [*polish*]

lemma *sint-1-eq-1-x* [*polish*, *simp*]:
sint (*1* :: ((*'a*::*len*) *bit0*) *word*) = *1*
sint (*1* :: ((*'a*::*len*) *bit1*) *word*) = *1*
sint (*1* :: ((*'a*::*len*) *bit0*) *signed word*) = *1*
sint (*1* :: ((*'a*::*len*) *bit1*) *signed word*) = *1*
⟨*proof*⟩

lemma *if-P-then-t-else-f-eq-t* [*L2opt*, *polish*]:
((*if P then t else f*) = *t*) = (*P* ∨ *t* = *f*)
⟨*proof*⟩

lemma *if-P-then-t-else-f-eq-f* [*L2opt*, *polish*]:
((*if P then t else f*) = *f*) = (¬ *P* ∨ *t* = *f*)
⟨*proof*⟩

lemma *sint-1-ne-sint-0*: *sint 1* $\neq$ *sint 0*
⟨*proof*⟩

lemmas *if-P-then-t-else-f-eq-f-simps* [*L2opt*, *polish*] =
if-P-then-t-else-f-eq-f [**where** *t* = *sint 1* **and** *f* = *sint 0*, *simplified sint-1-ne-sint-0 simp-thms*]
if-P-then-t-else-f-eq-t [**where** *t* = *sint 1* **and** *f* = *sint 0*, *simplified sint-1-ne-sint-0 simp-thms*]
if-P-then-t-else-f-eq-f [**where** *t* = *1* :: *int* **and** *f* = *0* :: *int*, *simplified zero-neq-one-class.one-neq-zero simp-thms*]
if-P-then-t-else-f-eq-t [**where** *t* = *1* :: *int* **and** *f* = *0* :: *int*, *simplified zero-neq-one-class.one-neq-zero simp-thms*]

lemma *boring-bind-K-bind* [*simp*, *polish*]:
(*gets X* >>= (λ-. *M*)) = *M*
(*return Y* >>= (λ-. *M*)) = *M*
⟨*proof*⟩

lemma *pred-and-true-var*[*simp*]: ((λ-. *True*) *and P*) = *P*
⟨*proof*⟩
lemma *pred-and-true*[*simp*]: (*P and* (λ-. *True*)) = *P*
⟨*proof*⟩

declare *pred-and-true-var* [*L2opt*, *polish*]
declare *pred-and-true* [*L2opt*, *polish*]

lemmas [*polish*] = *rel-simps eq-numeral-extra*

declare *ptr-add-0-id*[*polish*]
declare *ptr-coerce.simps*[*polish*]

declare *uint-nat*[*symmetric*, *polish*]

lemma *finally-throw*: *finally* (*throw x*) = *return x*
⟨*proof*⟩

lemma *finally-liftE*: *finally* (*liftE m*) = *m*
⟨*proof*⟩

lemma *bind-post-state-map-post-state*:
bind-post-state (*map-post-state f g*) *h* = *bind-post-state g* (λ*x*. *h* (*f x*))
⟨*proof*⟩

lemma *map-post-state-case-exception-or-result-distrib*:
map-post-state f (*case-exception-or-result g h v*) =
(*case-exception-or-result* (λ*x*. *map-post-state f* (*g x*)) (λ*x*. *map-post-state f* (*h x*)) *v*)
⟨*proof*⟩

lemma *map-post-state-case-exception-or-result-prod-distrib*:
map-post-state f
((*case v of* (*Exception e*, *t*) ⇒ *g e t*
| (*Result v*, *t*) ⇒ *h v t*)) =
((*case v of* (*Exception e*, *t*) ⇒ *map-post-state f* (*g e t*)
| (*Result v*, *t*) ⇒ *map-post-state f* (*h v t*)))
⟨*proof*⟩

lemma *finally-liftE-bind*: (*finally* ((*liftE L*) >>= (λ*r*. *R r*))) = (*L* >>= (λ*r*. (*finally* (*R r*))))
⟨*proof*⟩

lemma *finally-guardE*: *finally* (*do* { *r* <− *guard X*; *Y* }) = *do* {*r* <− *guard X*; *finally Y* }
⟨*proof*⟩

lemma *finally-condition-distrib*: *finally* (*condition c X Y*) = *condition c* (*finally X*) (*finally Y*)
⟨*proof*⟩

lemma *unite-Exception-option*:
unite (*Exception* (*v*::′*a option*)) = *Result* (*the v*)
⟨*proof*⟩

lemma *finally-bindE-liftE-throw*: *finally* (*X* >>= (λ*v*. *L2-VARS* (*throw v*) *ns*)) = *finally X*
⟨*proof*⟩

lemmas *finally-simps* =
finally-throw finally-liftE
finally-liftE-bind
finally-guardE
finally-condition-distrib
condition-fail-rhs condition-fail-lhs

bind-assoc

thm *finally-simps*
declare *finally-simps* [*polish*]

declare *finally-bindE-liftE-throw* [*simplified L2-VARS-def*, *polish*]

lemma *nested-bind*[*polish*, *simp*]:
do { *y* <− *f*;
return (*g y*)
} >>= *h* =
do { *y* <− *f*;
h (*g y*)
}
⟨*proof*⟩

lemma *select-UNIV-bind-const*[*simp*]: *select UNIV* >>= (λ-. *g*) = *g*
⟨*proof*⟩

lemma *do-bind-assoc*:
⋀*f fa fb*. *do* { *u* <− *f*::(*unit*, -) *res-monad*;
fa::(*unit*, -) *res-monad* } >>= *fb* = *do* {*u* <− *f*; *fa* >>=
fb::(*unit*, -) *res-monad* }
⟨*proof*⟩

24.1 Support to normalise guards and array index expressions

method *simp-guards* =
(*simp*
add:
guard-merge′ guard-merge-bind′ conj-commute
array-ptr-index-field-lvalue-conv
addressable-field-exec find-array-fields-Some
unat-less-helper
cong:
HOL.conj-cong)

⟨*ML*⟩
declare [[*simproc del*: *field-lvalue-unfold*]] — loops with ⟦*field-ti TYPE*(*?′a*) *?f* = *Some ?t*; *export-uinfo ?t* = *typ-uinfo-t TYPE*(*?′b*); *field-ti TYPE*(*?′b*) *?g* = *Some ?k*⟧ ⟹ &(*PTR*(*?′b*) &(*?p*→*?f*)→*?g*) = &(*?p*→*?f* @ *?g*)

⟨*ML*⟩

lemmas *whileLoop-congs-tupled* =
whileLoop-cong
whileLoop-cong [*split-tuple C* **and** *C′* **and** *B* **and** *B′ arity*: *2*]
whileLoop-cong [*split-tuple C* **and** *C′* **and** *B* **and** *B′ arity*: *3*]
whileLoop-cong [*split-tuple C* **and** *C′* **and** *B* **and** *B′ arity*: *4*]
whileLoop-cong [*split-tuple C* **and** *C′* **and** *B* **and** *B′ arity*: *5*]

whileLoop-cong [*split-tuple* C **and** C' **and** B **and** B' *arity*: *6*]
whileLoop-cong [*split-tuple* C **and** C' **and** B **and** B' *arity*: *7*]
whileLoop-cong [*split-tuple* C **and** C' **and** B **and** B' *arity*: *8*]
whileLoop-cong [*split-tuple* C **and** C' **and** B **and** B' *arity*: *9*]
whileLoop-cong [*split-tuple* C **and** C' **and** B **and** B' *arity*: *10*]
whileLoop-cong [*split-tuple* C **and** C' **and** B **and** B' *arity*: *11*]
whileLoop-cong [*split-tuple* C **and** C' **and** B **and** B' *arity*: *12*]
whileLoop-cong [*split-tuple* C **and** C' **and** B **and** B' *arity*: *13*]

lemma *finally-condition-throw-conv*: *finally* (*condition* c (*throw* x) g) = *condition* c (*return* x) (*finally* g)
⟨*proof*⟩

lemma *finally-unless-throw-conv*: *finally* (*do* {*unless* c (*throw* x); g}) = *condition* (λ-. ¬ c) (*return* x) (*finally* g)
⟨*proof*⟩

lemma *finally-bind-condition-throw-conv*: *finally* (*do* {*condition* c (*throw* x) *skip*; h}) = *condition* c (*return* x) (*finally* h)
⟨*proof*⟩

lemma *finally-return-conv*: *finally* (*return* x) = *return* x
⟨*proof*⟩

lemma *finally-bind-guard-conv*: *finally* (*do* {*guard* g; f}) = *do* {*guard* g; *finally* f}
⟨*proof*⟩

lemma *finally-bind-modify-conv*: *finally* (*do* {*modify* g; f}) = *do* {*modify* g; *finally* f}
⟨*proof*⟩

lemma *finally-bind-gets-conv*: *finally* ((*gets* g) ⤜ f) = *do* {r ← *gets* g; *finally* (f r)}
⟨*proof*⟩

lemma *finally-bind-gets-the-conv*: *finally* (*gets-the* g ⤜ f) = *do* {r ← *gets-the* g; *finally* (f r)}
⟨*proof*⟩

lemma *finally-bind-unknown-conv*: *finally* (*unknown* ⤜ f) = *do* {r ← *unknown*; *finally* (f r)}
⟨*proof*⟩

lemma *finally-bind-get-state-conv*: *finally* (*get-state* ⤜ f) = *do* {r ← *get-state*; *finally* (f r)}
⟨*proof*⟩

lemma *finally-bind-set-state-conv*: *finally* (*set-state* s ⤜ f) = *do* {r ← *set-state* s; *finally* (f r)}

⟨*proof*⟩

lemma *finally-bind-select-conv*: *finally* (*select S* ⤜ *f*) = *do* {*r* ← *select S*; *finally* (*f r*)}
⟨*proof*⟩

lemma *finally-bind-assert-conv*: *finally* (*assert P* ⤜ *f*) = *do* {*r* ← *assert P*; *finally* (*f r*)}
⟨*proof*⟩

lemma *finally-bind-assume-conv*: *finally* (*assume P* ⤜ *f*) = *do* {*r* ← *assume P*; *finally* (*f r*)}
⟨*proof*⟩

lemma *finally-bind-assert-opt-conv*: *finally* (*assert-opt v* ⤜ *f*) = *do* {*r* ← *assert-opt v*; *finally* (*f r*)}
⟨*proof*⟩

lemmas *finally-convs* =
finally-liftE finally-liftE-bind
finally-condition-throw-conv
finally-unless-throw-conv
finally-bind-condition-throw-conv
finally-return-conv
finally-bind-guard-conv
finally-bind-modify-conv
finally-bind-gets-conv
finally-bind-gets-the-conv
finally-bind-unknown-conv
finally-bind-get-state-conv
finally-bind-set-state-conv
finally-bind-select-conv
finally-bind-assert-conv
finally-bind-assume-conv
finally-bind-assert-opt-conv
finally-throw

lemma *try-bind-liftE-conv*: (*try* (*liftE f* ⤜ *g*)) = (*liftE f* ⤜ (λ*x*. *try* (*g x*)))
⟨*proof*⟩

lemma *try-bind-guard-conv*: *try* (*do* {*guard g*; *f*}) = *do* {*guard g*; *try f*}
⟨*proof*⟩

lemma *try-unless-throw-Inr-conv*:
try (*do* {*unless c* (*throw* (*Inr x*)); *g*}) = *condition* (λ-. ¬ *c*) (*return x*) (*try g*)
⟨*proof*⟩

lemma *try-unless-throw-Inl-conv*:
try (*do* {*unless c* (*throw* (*Inl x*)); *g*}) = *condition* (λ-. ¬ *c*) (*throw x*) (*try g*)

⟨*proof*⟩

lemma *try-when-throw-Inr-conv*:
try (*do* {*when c* (*throw* (*Inr x*)); *g*}) = *condition* (λ-. *c*) (*return x*) (*try g*)
⟨*proof*⟩

lemma *try-when-throw-Inl-conv*:
try (*do* {*when c* (*throw* (*Inl x*)); *g*}) = *condition* (λ-. *c*) (*throw x*) (*try g*)
⟨*proof*⟩

lemma *try-bind-condition-throw-Inr-conv*: *try* (*do* {*condition c* (*throw* (*Inr x*)) *skip*; *h*}) = *condition c* (*return x*) (*try h*)
⟨*proof*⟩

lemma *try-bind-condition-throw-Inl-conv*: *try* (*do* {*condition c* (*throw* (*Inl x*)) *skip*; *h*}) = *condition c* (*throw x*) (*try h*)
⟨*proof*⟩

lemma *try-condition-conv*: *try* (*condition c f g*) = *condition c* (*try f*) (*try g*)
⟨*proof*⟩

lemmas *try-convs* =
try-bind-liftE-conv
try-bind-guard-conv
try-unless-throw-Inr-conv
try-unless-throw-Inl-conv
try-when-throw-Inr-conv
try-when-throw-Inl-conv
try-bind-condition-throw-Inr-conv
try-bind-condition-throw-Inl-conv
try-condition-conv

lemma *runs-to-partial-case-prod* : (⋀*a b*. (*f a b*) ∙ *s* ?⦃ *Q*⦄) ⟹
(*case x of* (*a*, *b*) ⇒ *f a b*) ∙ *s* ?⦃ *Q*⦄
⟨*proof*⟩

24.2 Monad simplification with custom congruence rules

context *open-types*
begin

We supply a custom simplification method for spec monad expressions that gathers and propagates information from conditions and guards while descending into the term. The core motivation is to clean up repeated occurrences of *PTR-VALID*(*'a*) (*heap-typing s*) *p*. Allthough this is a state

dependent predicate it stays invariant as long as the typing does not change. So the goal of the simplification method is to prove preservation of such predicates while descending into the term. Supplying congruence rules to the simplifier is unfortunately not enough as we want to keep control over what kind of invariants are propagated. Unfortunately the simplifier has no concept of a *congprocs* that are triggered when descending into the term only the *simprocs* which are triggered by the usual bottom up simplifier strategy. To work around this limitation we implement *congprocs* by *simprocs*:

- We block the simplifier to descend into compound terms by adding trivial congruence rules to the simplifier, like $f >>= g \equiv f >>= g$
- We add a simproc that then triggers on the $f >>= g$ and manually extends the context and invokes the simplifier on the subterms.

end

definition *ADD-FACT*:: $('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ **where**
ADD-FACT P s = *P s*
definition *PRESERVED-FACTS*:: ('*e*::*default*, '*a*, '*s*) *spec-monad* ⇒ '*s* ⇒ ('*e*, '*a*) *exception-or-result* ⇒ '*s* ⇒ *bool* **where**
PRESERVED-FACTS f s r t = (*reaches f s r t*)
definition *PRESERVED-FACTS-WHILE* :: ('*a* ⇒ '*s* ⇒ *bool*) ⇒ ('*a* ⇒ ('*e*::*default*, '*a*, '*s*) *spec-monad*) ⇒ '*a* ⇒ '*s* ⇒ '*a* ⇒ '*s* ⇒ *bool* **where**
PRESERVED-FACTS-WHILE C B i s r t = *whileLoop-unroll-reachable C B i s r t*

definition *RUN-CASE-PROD* ::
('*a* ⇒ '*b* ⇒ ('*c*::*default*, '*d*, '*e*) *spec-monad*) ⇒ '*a* × '*b* ⇒ '*e* ⇒
('*f* ⇒ '*g* ⇒ ('*c*, '*d*, '*e*) *spec-monad*) ⇒ '*f* × '*g* ⇒ *bool* **where**
RUN-CASE-PROD f x s f' x' ⟷ *run* (*case-prod f x*) *s* = *run* (*case-prod f' x'*) *s*

lemma *ADD-FACT-D*: *ADD-FACT P s* ⟹ *P s*
⟨*proof*⟩

lemma *RUN-CASE-PROD-I*:*x* ≡ *x'* ⟹ *run* (*case-prod f x'*) *s* ≡ *run* (*case-prod f' x'*) *s* ⟹ *RUN-CASE-PROD f x s f' x'*
⟨*proof*⟩

named-theorems
monad-simp-simp simplification rules to derive facts **and**
monad-simp-augment-rule conditional rules to augment facts **and**
monad-simp-split-tuple-cong congruence rules to control tuple expansion

lemma *PRESERVED-FACTS-runs-to-partial*: *PRESERVED-FACTS f s r t* ⟹ *f* · *s* ?⦃*Q*⦄ ⟹ *Q r t*

⟨*proof*⟩

lemma *STOPI*: $x \equiv y \Longrightarrow x \equiv STOP\ y$ ⟨*proof*⟩

lemma *expand-unused-case-prod*:
$(\lambda(x,y::('c * 'd)).\ f\ x) = (\lambda(x,\ y1,\ y2).\ f\ x)$
⟨*proof*⟩

lemma *PRESERVED-FACTS-return-eq*: *PRESERVED-FACTS* (*return x*) *s* (*Result y*) *t* $\Longrightarrow y = x$
⟨*proof*⟩

⟨*ML*⟩
declare [[*simproc del*: *monad-run-simproc*]]

lemma *spec-monad-ext'*: $(\bigwedge s.\ run\ f\ s \equiv run\ f'\ s) \Longrightarrow f \equiv f'$
⟨*proof*⟩

⟨*ML*⟩

We make a global setup here, as the locale one within *heap-typing-state* fails. The reason is that the pattern is not taken into account in simproc equality when inserted into the net. Hence multiple instances (e.g. for *globals* and *lifted-globals*) are considered a duplicate when adding them both to the net but the simplifier does not match the pattern for *globals* on a *lifted-globals* instance. In the upcoming Isabelle2024 this issue should be resolved.

⟨*ML*⟩
declare [[*simproc del*: *unchanged-typing-spec-monad*]]

⟨*ML*⟩
declare [[*simproc del*: *spec-monad-simproc*]]

⟨*ML*⟩

context *open-types-heap-typing-state*
begin
lemma *PRESERVED-FACTS-unchanged-typing-ptr-valid-preserved*[*monad-simp-simp*]:
reaches f s r t $\Longrightarrow$ *ptr-valid* (*heap-typing s*) *p* $\Longrightarrow f \cdot s\ ?\{\!|\lambda\text{-}\ t.\ unchanged\text{-}typing\text{-}on\ UNIV\ s\ t|\!\}\ \Longrightarrow$
ptr-valid (*heap-typing t*) *p*
⟨*proof*⟩

lemma *PRESERVED-FACTS-WHILE-unchanged-typing-ptr-valid-preserved*[*monad-simp-simp*]:
assumes *reach*: *whileLoop-unroll-reachable C B i s r t*

assumes *valid*: *ptr-valid* (*heap-typing s*) *p*
assumes *B*: $\bigwedge r\ t.\ C\ r\ t \Longrightarrow (B\ r) \cdot t\ ?\{\lambda\text{-}.\ unchanged\text{-}typing\text{-}on\ UNIV\ t\}$
shows *ptr-valid* (*heap-typing t*) *p*
⟨*proof*⟩

end

lemma *PRESERVED-FACTS-run-bind-cong*[*monad-simp-cong split*: *g* **and** *g'*]:
run f s = *run f' s* ⟹ (⋀*t r*. *PRESERVED-FACTS f' s* (*Result r*) *t* ⟹ *run* (*g r*) *t* = *run* (*g' r*) *t*) ⟹
(*run* (*bind f g*) *s*) = (*run* (*bind f' g'*) *s*)
⟨*proof*⟩

lemma *ADD-FACT-run-bind-guard-cong*[*monad-simp-cong*]:
P s = *P' s* ⟹ (*ADD-FACT P' s* ⟹ *run f s* = *run f' s*) ⟹
(*run* (*bind* (*guard P*) (λ-. *f*)) *s*) = (*run* (*bind* (*guard P'*) (λ-. *f'*)) *s*)
⟨*proof*⟩

lemma *PRESERVED-FACTS-WHILE-run-whileLoop-cong*[*monad-simp-cong split*: *B* **and** *B'* **and** *C* **and** *C'*]:
assumes *i*: *i* = *i'*
assumes *C-B*[*unfolded PRESERVED-FACTS-WHILE-def*]:
⋀*t r*. *PRESERVED-FACTS-WHILE C B i s r t* ⟹ (*C r t* ≡ *C' r t*) &&& (*C' r t* ⟹ (*run* (*B r*) *t* ≡ *run* (*B' r*) *t*))
shows *run* (*whileLoop C B i*) *s* = *run* (*whileLoop C' B' i'*) *s*
⟨*proof*⟩

lemma *whileLoop-condition-only-cong*[*monad-simp-split-tuple-cong*]:
C = *C'* ⟹ *whileLoop C B I* = *whileLoop C' B I*
⟨*proof*⟩

lemma *PRESERVED-FACTS-run-bind-exception-or-result-cong*[*monad-simp-cong split*: *g* **and** *g'*]:
run f s = *run f' s* ⟹ (⋀*t r*. *PRESERVED-FACTS f' s r t* ⟹ *run* (*g r*) *t* = *run* (*g' r*) *t*) ⟹
(*run* (*bind-exception-or-result f g*) *s*) = (*run* (*bind-exception-or-result f' g'*) *s*)
⟨*proof*⟩

lemma *PRESERVED-FACTS-run-on-exit'-cong*[*monad-simp-cong split*: *g* **and** *g'*]:
run f s = *run f' s* ⟹ (⋀*t r*. *PRESERVED-FACTS f' s r t* ⟹ *run g t* = *run g' t*) ⟹
(*run* (*on-exit' f g*) *s*) = (*run* (*on-exit' f' g'*) *s*)
⟨*proof*⟩

lemma *PRESERVED-FACTS-run-on-exit-cong*[*monad-simp-cong split*: *g* **and** *g'*]:
run f s = *run f' s* ⟹
(⋀*t r*. *PRESERVED-FACTS f' s r t* ⟹ *run* ((*state-select g*::('*e*::*default*, *unit*, '*s*) *spec-monad*)) *t* = *run* (*state-select g'*) *t*) ⟹
(*run* (*on-exit f g*::('*e*::*default*, '*a*, '*s*) *spec-monad*) *s*) = (*run* (*on-exit f' g'*) *s*)

⟨*proof*⟩

lemma *ADD-FACT-run-condition-cong* [*monad-simp-cong*]:
assumes $c\ s = c'\ s$
assumes $ADD\text{-}FACT\ c'\ s \Longrightarrow run\ f\ s = run\ f'\ s$
assumes $ADD\text{-}FACT\ (\lambda s.\ \neg\ c'\ s)\ s \Longrightarrow run\ g\ s = run\ g'\ s$
shows $run\ (condition\ c\ f\ g)\ s\ = run\ (condition\ c'\ f'\ g')\ s$
⟨*proof*⟩

lemma *ADD-FACT-run-when* [*monad-simp-cong*]:
assumes $c = c'$
assumes $ADD\text{-}FACT\ (\lambda\text{-}.\ c')\ s \Longrightarrow run\ f\ s = run\ f'\ s$
shows $run\ (when\ c\ f)\ s\ = run\ (when\ c'\ f')\ s$
⟨*proof*⟩

lemma *PRESERVED-FACTS-run-catch-cong* [*monad-simp-cong split*: g **and** g']:
$run\ f\ s = run\ f'\ s \Longrightarrow (\bigwedge t\ e.\ PRESERVED\text{-}FACTS\ f'\ s\ (Exn\ e)\ t \Longrightarrow run\ (g\ e)\ t = run\ (g'\ e)\ t) \Longrightarrow$
$(run\ (catch\ f\ g)\ s) = (run\ (catch\ f'\ g')\ s)$
⟨*proof*⟩

lemma *run-finally-cong* [*monad-simp-cong*]:
assumes $run\ f\ s = run\ f'\ s$
shows $run\ (finally\ f)\ s = run\ (finally\ f')\ s$
⟨*proof*⟩

lemma *run-liftE-cong* [*monad-simp-cong*]:
assumes $run\ f\ s = run\ f'\ s$
shows $run\ (liftE\ f)\ s = run\ (liftE\ f')\ s$
⟨*proof*⟩

lemma *run-guard-cong* [*monad-simp-cong*]:
assumes $P\ s = P'\ s$
shows $run\ (guard\ P)\ s = run\ (guard\ P')\ s$
⟨*proof*⟩

lemma *run-try-cong* [*monad-simp-cong*]:
assumes $run\ f\ s = run\ f'\ s$
shows $run\ (try\ f)\ s = run\ (try\ f')\ s$
⟨*proof*⟩

lemma *run-case-prod-cong*[*monad-simp-cong*]: $RUN\text{-}CASE\text{-}PROD\ f\ x\ s\ f'\ x' \Longrightarrow$

$(run\ (case\text{-}prod\ f\ x)\ s) = (run\ (case\text{-}prod\ f'\ x')\ s)$
⟨*proof*⟩

lemma *run-bind-when-throw-cong* [*monad-simp-cong*]:
$c = c' \Longrightarrow e = e' \Longrightarrow (ADD\text{-}FACT\ (\lambda\text{-}.\ \neg\ c')\ s \Longrightarrow run\ f\ s = run\ f'\ s) \Longrightarrow$
$run\ (bind\ (when\ c\ (throw\ e))\ (\lambda\text{-}.\ f))\ s = run\ (bind\ (when\ c'\ (throw\ e'))\ (\lambda\text{-}.\ f'))$

s
⟨*proof*⟩

⟨*ML*⟩

lemma *with-fresh-stack-ptr-stop-cong* [*monad-simp-global-stop-cong*]:
typ-heap-typing.with-fresh-stack-ptr r w heap-typing heap-typing-upd $\mathcal{S}$ *n init f* ≡
typ-heap-typing.with-fresh-stack-ptr r w heap-typing heap-typing-upd $\mathcal{S}$ *n init f*
⟨*proof*⟩

lemma *assume-with-fresh-stack-ptr-stop-cong* [*monad-simp-global-stop-cong*]:
typ-heap-typing.assume-with-fresh-stack-ptr r w heap-typing heap-typing-upd $\mathcal{S}$ *n init f* ≡
typ-heap-typing.assume-with-fresh-stack-ptr r w heap-typing heap-typing-upd $\mathcal{S}$ *n init f*
⟨*proof*⟩

lemma *guard-with-fresh-stack-ptr-stop-cong* [*monad-simp-global-stop-cong*]:
typ-heap-typing.guard-with-fresh-stack-ptr r w heap-typing heap-typing-upd $\mathcal{S}$ *n init f* ≡
typ-heap-typing.guard-with-fresh-stack-ptr r w heap-typing heap-typing-upd $\mathcal{S}$ *n init f*
⟨*proof*⟩

context *typ-heap-simulation-open-types-stack*
begin

lemma *PRESERVED-FACTS-assume-stack-alloc-ptr-valid-preserved*[*monad-simp-simp*]:
reaches (*assume-stack-alloc n init*) *s x t* ⟹ *ptr-valid* (*heap-typing s*) (*p*::*'b*::*mem-type ptr*) ⟹
typ-uinfo-t TYPE(*'b*) ≠ *typ-uinfo-t TYPE*(*stack-byte*) ⟹
ptr-valid (*heap-typing t*) *p*
⟨*proof*⟩

lemma *PRESERVED-FACTS-with-fresh-stack-ptr-cong*[*monad-simp-cong split*: *f* **and** *f'*]:
fixes *f*::*'a ptr* ⇒ (*'e*::*default*, *'d*, *'t*) *spec-monad*
assumes ⋀*t p*. *PRESERVED-FACTS* (*assume-stack-alloc n init*::(*'e*::*default*, *'a ptr*, *'t*) *spec-monad*) *s* (*Result p*) *t* ⟹
run (*f p*) *t* = *run* (*f' p*) *t*
shows *run* (*with-fresh-stack-ptr n init f*) *s* = *run* (*with-fresh-stack-ptr n init f'*) *s*
⟨*proof*⟩

lemma *PRESERVED-FACTS-assume-with-fresh-stack-ptr-cong*[*monad-simp-cong split*: *f* **and** *f'*]:
fixes *f*::*'a ptr* ⇒ (*'e*::*default*, *'d*, *'t*) *spec-monad*
assumes ⋀*t p*. *PRESERVED-FACTS* (*assume-stack-alloc n init*::(*'e*::*default*, *'a*

ptr, *'t*) *spec-monad*) *s* (*Result p*) *t* ⟹
run (*f p*) *t* = *run* (*f' p*) *t*
shows *run* (*assume-with-fresh-stack-ptr n init f*) *s* = *run* (*assume-with-fresh-stack-ptr n init f'*) *s*
⟨*proof*⟩

lemma *PRESERVED-FACTS-guard-with-fresh-stack-ptr-cong*[*monad-simp-cong split*: *f* **and** *f'*]:
fixes *f*::*'a ptr* ⇒ (*'e*::*default*, *'d*, *'t*) *spec-monad*
assumes ⋀*t p*. *PRESERVED-FACTS* (*assume-stack-alloc n init*::(*'e*::*default*, *'a ptr*, *'t*) *spec-monad*) *s* (*Result p*) *t* ⟹
run (*f p*) *t* = *run* (*f' p*) *t*
shows *run* (*guard-with-fresh-stack-ptr n init f*) *s* = *run* (*guard-with-fresh-stack-ptr n init f'*) *s*
⟨*proof*⟩

lemma *ptr-valid-assume-stack-alloc* [*monad-simp-derive-rule*]:
PRESERVED-FACTS (*assume-stack-alloc n init*::(*'e*::*default*, *'a ptr*, *'t*) *spec-monad*) *s* (*Result p*) *t* ⟹
ADD-FACT (λ*t*. *PTR-VALID*(*'a*) (*heap-typing t*) *p*) *t*
⟨*proof*⟩

end

⟨*ML*⟩

lemmas *word-augment-rules*[*augment-rule*] =
word-le-nat-alt [*THEN iffD1*]
word-less-nat-alt [*THEN iffD1*]
word-le-def [*THEN iffD1*]
word-less-def [*THEN iffD1*]
word-sle-def [*THEN iffD1*]
word-sless-alt [*THEN iffD1*]
lemmas *word-monad-simp*[*monad-simp-simp*] =
word-le-nat-alt [*THEN iffD2*]
word-less-nat-alt [*THEN iffD2*]
word-le-def [*THEN iffD2*]
word-less-def [*THEN iffD2*]
word-sle-def [*THEN iffD2*]
word-sless-alt [*THEN iffD2*]

thm *monad-simp-augment-rule*
thm *monad-simp-simp*

method *array-index-to-ptr-arith-simp* **uses** *simp cong* =
(*use TrueI*[*array-bound-mksimps*, *simproc add*: *field-lvalue-unfold*]
in ‹*monad-simp*

simp add:*field-lvalue-array-index'* *simp*
simp del: *field-lvalue-append*
cong: *if-cong cong*›)

end

theory *Runs-To-VCG-StackPointer*
imports
HeapLift
Refines-Spec
begin

definition *distinct-span xs* ≡
distinct-prop (λ(*v1*, *s1*) (*v2*, *s2*). *disjnt* {*v1* ..+ *s1*} {*v2* ..+ *s2*}) *xs*

definition *distinct-spans xs* ≡
distinct-prop (λ(*v1*, *n1*, *s1*) (*v2*, *n2*, *s2*). *disjnt* {*v1* ..+ *n1* ∗ *s1*} {*v2* ..+ *n2* ∗ *s2*}) *xs*

lemma *distinct-span-spans-conv*:
distinct-span xs ⟷ *distinct-spans* (*map* (λ(*v*, *s*). (*v*, *1*, *s*)) *xs*)
⟨*proof*⟩

definition *lvp-distinct xs* ≡
distinct-spans (*map* (λ(*d*, *v*, *n*, *s*). (*v*, *n*, *s*)) *xs*) ∧
distinct (*map* (λ(*d*, *v*, *n*, *s*). (*d*, *v*)) *xs*) ∧
sorted-wrt (λ(*d1*, -) (*d2*, -). *stack-free d1* ⊆ *stack-free d2*) *xs* ∧
(∀(*d*, *v*, *n*, *s*)∈*set xs*.
disjnt {*v* ..+ *n* ∗ *s*} (*stack-free d*) ∧ *s* > *0*)

lemma *lvp-distinct-pairwise-disjnt*:
⟦ *lvp-distinct xs*; *xs* ! *m1* = (*d1*, *v1*, *n1*, *s1*); *xs* ! *m2* = (*d2*, *v2*, *n2*, *s2*); *m1* < *m2*; *m2* < *length xs* ⟧ ⟹
disjnt {*v1* ..+ *n1* ∗ *s1*} {*v2* ..+ *n2* ∗ *s2*}
⟨*proof*⟩

lemma *lvp-distinct-spansD*: *lvp-distinct xs* ⟹
distinct-spans (*map* (λ(*d*, *v*, *n*, *s*). (*v*, *n*, *s*)) *xs*)
⟨*proof*⟩

named-theorems *stack-ptr-simps*

context *stack-simulation-heap-typing* **begin**

lemmas [*stack-ptr-simps*] =
stack-ptr-acquire-def
stack-ptr-release-def

write-same-ZERO
stack-releases-stack-allocs-inverse
root-ptr-valid-ptr-valid

lemma *runs-to-guard-with-fresh-stack-ptr*[*runs-to-vcg*]:
assumes *f*[*runs-to-vcg*]: $\bigwedge d\ ptr\ vs.$
$(ptr,\ d) \in stack\text{-}allocs\ n\ \mathcal{S}\ TYPE('a)\ (heap\text{-}typing\ s) \Longrightarrow$
$vs \in init\ s \Longrightarrow$
$length\ vs = n \Longrightarrow$
$(\forall\, i{\in}\{0..{<}n\}.\ r\ s\ (ptr +_p int\ i) = ZERO('a)) \Longrightarrow$
$(\forall\, i{\in}\{0..{<}n\}.\ root\text{-}ptr\text{-}valid\ d\ (ptr +_p int\ i)) \Longrightarrow$
$(f\ ptr) \cdot (stack\text{-}ptr\text{-}acquire\ n\ vs\ ptr\ d\ s)$
$\{\!|\lambda r\ u.\ (\forall\, i{<}n.\ root\text{-}ptr\text{-}valid\ (heap\text{-}typing\ u)\ (ptr +_p int\ i)) \wedge Q\ r\ (stack\text{-}ptr\text{-}release\ n\ ptr\ u)|\!\}$
shows
$(guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init\ f) \cdot s\ \{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *runs-to-with-fresh-stack-ptr*[*runs-to-vcg*]:
assumes *f*[*runs-to-vcg*]:$\bigwedge d\ ptr\ vs.$
$(ptr,\ d) \in stack\text{-}allocs\ n\ \mathcal{S}\ TYPE('a)\ (heap\text{-}typing\ s) \Longrightarrow$
$vs \in init\ s \Longrightarrow$
$length\ vs = n \Longrightarrow$
$(\forall\, i{\in}\{0..{<}n\}.\ r\ s\ (ptr +_p int\ i) = ZERO('a)) \Longrightarrow$
$(\forall\, i{\in}\{0..{<}n\}.\ root\text{-}ptr\text{-}valid\ d\ (ptr +_p int\ i)) \Longrightarrow$
$(f\ ptr) \cdot (stack\text{-}ptr\text{-}acquire\ n\ vs\ ptr\ d\ s)\ \{\!|\lambda r\ u.\ Q\ r\ (stack\text{-}ptr\text{-}release\ n\ ptr\ u)|\!\}$
shows
$(assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init\ f) \cdot s\ \{\!|\ Q\ |\!\}$
$(with\text{-}fresh\text{-}stack\text{-}ptr\ \ \ n\ init\ f) \cdot s\ \{\!|\ Q\ |\!\}$
⟨*proof*⟩

lemma *lvp-distinct-singleton-local-with-params*:
fixes $p{::}'a\ ptr$
assumes $(p,\ d') \in stack\text{-}allocs\ n\ \mathcal{S}\ TYPE('a)\ d$
assumes $root\text{-}ptr\text{-}valid\ d'\ p$
assumes *distinct-span params*
assumes $\forall\,(v,\ s) \in set\ params.\ disjnt\ \{v\ ..{+}\ s\}\ (stack\text{-}free\ d)$
assumes $\forall\,(v,\ s) \in set\ params.\ s > 0$
shows $lvp\text{-}distinct\ ((d',\ ptr\text{-}val\ p,\ n,\ size\text{-}td\ (typ\text{-}uinfo\text{-}t\ TYPE('a)))\#(map\ (\lambda(v,\ s).\ (d,\ v,\ 1,\ s))\ params))$
⟨*proof*⟩

lemma *runs-to-init-local-variables-and-parameters*:
assumes
$n > 0$
distinct-span params
$\forall\,(v,\ sz){\in}set\ params.\ disjnt\ \{v..{+}sz\}\ (stack\text{-}free\ (heap\text{-}typing\ s))$
$\forall\,(v,\ sz){\in}set\ params.\ 0 < sz$

$\bigwedge d\ ptr\ vs.$
$(ptr,\ d) \in stack\text{-}allocs\ n\ \mathcal{S}\ TYPE('a)\ (heap\text{-}typing\ s) \Longrightarrow$
$vs \in init\ s \Longrightarrow$
$length\ vs = n \Longrightarrow$
$\forall i{\in}\{0..{<}n\}.\ r\ s\ (ptr +_p int\ i) = ZERO('a) \Longrightarrow$
$\forall i{\in}\{0..{<}n\}.\ root\text{-}ptr\text{-}valid\ d\ (ptr +_p int\ i) \Longrightarrow$
$lvp\text{-}distinct\ ((d,\ ptr\text{-}val\ ptr,\ n,\ size\text{-}td\ (typ\text{-}uinfo\text{-}t\ TYPE('a)))\#(map\ (\lambda(v,\ sz).\ (heap\text{-}typing\ s,\ v,\ 1,\ sz))\ params)) \Longrightarrow$
$f\ ptr \cdot (stack\text{-}ptr\text{-}acquire\ n\ vs\ ptr\ d\ s)\ \{\!|\ \lambda r\ u.\ Q\ r\ (stack\text{-}ptr\text{-}release\ n\ ptr\ u)\ |\!\}$

shows *assume-with-fresh-stack-ptr n init f · s {|* *Q* *|}*
⟨*proof*⟩

lemma *lvp-distinct-add-stack-allocs*:
fixes *p*::'*a ptr*
assumes $(p,\ d') \in stack\text{-}allocs\ n\ \mathcal{S}\ TYPE('a)\ (fst\ (hd\ xs))$
assumes $n > 0$
assumes *root-ptr-valid d' p*
assumes *lvp-distinct xs*
shows $lvp\text{-}distinct\ ((d',\ ptr\text{-}val\ p,\ n,\ size\text{-}td\ (typ\text{-}uinfo\text{-}t\ TYPE('a)))\#xs)$
⟨*proof*⟩

lemma *runs-to-add-local-variable*:
assumes
lvp-distinct xs
heap-typing s = fst (hd xs)
$n > 0$
$\bigwedge d\ ptr\ vs.$
$(ptr,\ d) \in stack\text{-}allocs\ n\ \mathcal{S}\ TYPE('a)\ (heap\text{-}typing\ s) \Longrightarrow$
$vs \in init\ s \Longrightarrow$
$length\ vs = n \Longrightarrow$
$\forall i{\in}\{0..{<}n\}.\ r\ s\ (ptr +_p int\ i) = ZERO('a) \Longrightarrow$
$\forall i{\in}\{0..{<}n\}.\ root\text{-}ptr\text{-}valid\ d\ (ptr +_p int\ i) \Longrightarrow$
$lvp\text{-}distinct\ ((d,\ ptr\text{-}val\ ptr,\ n,\ size\text{-}td\ (typ\text{-}uinfo\text{-}t\ TYPE('a)))\#xs) \Longrightarrow$
$(f\ ptr) \cdot (stack\text{-}ptr\text{-}acquire\ n\ vs\ ptr\ d\ s)\ \{\!|\ \lambda r\ u.\ Q\ r\ (stack\text{-}ptr\text{-}release\ n\ ptr\ u)\ |\!\}$
shows $(assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ n\ init\ f) \cdot s\ \{\!|\ Q\ |\!\}$
⟨*proof*⟩

end

end

theory *AutoCorres*
imports
LocalVarExtract

AutoCorresSimpset
Polish
Runs-To-VCG-StackPointer

keywords
autocorres
init−autocorres
final−autocorres:: *thy-decl*
begin

no-syntax *-Lab*:: *'a bexp* ⇒ (*'a*,*'p*,*'f*) *com* ⇒ *bdy*
(‹-•/-› [*1000*,*71*] *81*) — avoid syntax conflict with $f \cdot s \; \{\!|\; Q \;|\!\}$

declare *word-neq-0-conv* [*simp del*]
declare *neq0-conv* [*simp del*]
declare *fun-upd-apply*[*simp del*]
declare *fun-upd-same* [*simp add*]
lemma *o-const-simp*[*simp*]: (λ*x*. *C*) ∘ *f* = (λ*x*. *C*)
⟨*proof*⟩

lemma *corresTA-trivial*: *corresTA* (λ-. *True*) (λ*x*. *x*) (λ*x*. *x*) *A A*
⟨*proof*⟩

lemma *L2Tcorres-trivial-from-in-out-parameters*:
IOcorres P Q st rx ex A C ⟹ *L2Tcorres id A A*
⟨*proof*⟩

lemma *L2Tcorres-trivial-from-local-var-extract*:
L2corres st rx ex P A C ⟹ *L2Tcorres id A A*
⟨*proof*⟩

lemma *corresTA-trivial-from-heap-lift*:
L2Tcorres st A C ⟹ *corresTA* (λ-. *True*) (λ*x*. *x*) (λ*x*. *x*) *A A*
⟨*proof*⟩

lemma *corresXF-from-L2-call*:
L2-call c-WA emb ns = *A* ⟹ *corresXF* (λ*s*. *s*) (λ*rv s*. *rv*) (λ*r t*. *emb r*) (λ-. *True*) *A c-WA*
⟨*proof*⟩

definition *ac-corres'* *exn st check-termination AF* Γ *rx ex G* ≡
λ*A B*. ∀ *s*. (*G s* ∧ *succeeds A* (*st s*)) ⟶
(∀ *t*. Γ ⊢ ⟨*B*, *Normal s*⟩ ⇒ *t* ⟶

(*case t of*
Normal s′ ⇒ (*Result* (*rx s′*), *st s′*) ∈ *outcomes* (*run A* (*st s*))
| *Abrupt s′* ⇒ (*exn* (*ex s′*), *st s′*) ∈ *outcomes* (*run A* (*st s*))
| *Fault e* ⇒ *e* ∈ *AF*
| - ⇒ *False*))
∧ (*check-termination* ⟶ Γ ⊢ *B* ↓ *Normal s*)

lemma *ac-corres′-nd-monad*:
assumes *ac*: *ac-corres st check-termination AF* Γ *rx ex G B C*
assumes *refines*: ⋀*s*. *refines B A s s* (*rel-prod rel-liftE* (=))
shows *ac-corres′* (λ-. *Exception* (*default*::*unit*)) *st check-termination AF* Γ *rx ex G A C*
⟨*proof*⟩

lemma *refines-spec-rel-Nonlocal-conv*:
shows *refines f g s t* (*rel-prod* (*rel-xval rel-Nonlocal* (=)) (=))
⟷ *refines f* (*map-value* (*map-exn Nonlocal*) *g*) *s t* (*rel-prod* (=) (=))
⟨*proof*⟩

lemmas *refines-eq-convs* = *refines-spec-rel-Nonlocal-conv sum.rel-eq rel-xval-eq Relation.eq-OO*

lemma *ac-corres′-exception-monad*:
assumes *ac*: *ac-corres st check-termination AF* Γ *rx ex G B C*
assumes *refines*: ⋀*s*. *refines B A s s* (*rel-prod* (=) (=))
shows *ac-corres′ Exn st check-termination AF* Γ *rx ex G A C*
term *map-value* (*map-exn Nonlocal*) *A*
⟨*proof*⟩

lemma *ac-corres-chain*:
⟦ *L1corres check-termination Gamma c-L1 c*;
L2corres st-L2 rx-L2 ex-L2 P-L2 c-L2 c-L1;
L2Tcorres st-HL c-HL c-L2;
corresTA P-WA rx-WA ex-WA c-WA c-HL;
L2-call c-WA emb ns = *A*
⟧ ⟹
ac-corres (*st-HL o st-L2*) *check-termination* {*AssumeError*, *StackOverflow*} *Gamma* (*rx-WA o rx-L2*) (*emb o ex-WA o ex-L2*) (*P-L2 and* (*P-WA o st-HL o st-L2*)) *A c*

⟨*proof*⟩

lemma *ac-corres-chain-sim-nd-monad*:
⟦ *L1corres check-termination Gamma c-L1 c*;
L2corres st-L2 rx-L2 ex-L2 P-L2 c-L2 c-L1;
IOcorres P-IO Q-IO st-IO rx-IO ex-IO c-IO c-L2;
L2Tcorres st-HL c-HL c-IO;
corresTA P-WA rx-WA ex-WA c-WA c-HL;
⋀*s*. *refines c-WA A s s* (*rel-prod rel-liftE* (=))
⟧ ⟹

ac-corres' (λ-. *Exception* (*default*::*unit*)) (*st-HL o st-IO o st-L2*) *check-termination* {*AssumeError*, *StackOverflow*} *Gamma*
(λ*s*. (*rx-WA o* (λ*v*. *rx-IO v* (*st-L2 s*)) *o rx-L2*) *s*)
(λ*s*. (*ex-WA o* (λ*e*. *ex-IO e* (*st-L2 s*)) *o ex-L2*) *s*)
(*P-L2 and* (*P-IO o st-L2*) *and* (*P-WA o st-HL o st-IO o st-L2*)) *A c*
⟨*proof*⟩

lemma *ac-corres-chain-sim-exception-monad*:
⟦ *L1corres check-termination Gamma c-L1 c*;
L2corres st-L2 rx-L2 ex-L2 P-L2 c-L2 c-L1;
IOcorres P-IO Q-IO st-IO rx-IO ex-IO c-IO c-L2;
L2Tcorres st-HL c-HL c-IO;
corresTA P-WA rx-WA ex-WA c-WA c-HL;
⋀*s*. *refines c-WA A s s* (*rel-prod* (=) (=))
⟧ ⟹
ac-corres' Exn (*st-HL o st-IO o st-L2*) *check-termination* {*AssumeError*, *StackOverflow*} *Gamma*
(λ*s*. (*rx-WA o* (λ*v*. *rx-IO v* (*st-L2 s*)) *o rx-L2*) *s*)
(λ*s*. (*ex-WA o* (λ*e*. *ex-IO e* (*st-L2 s*)) *o ex-L2*) *s*)
(*P-L2 and* (*P-IO o st-L2*) *and* (*P-WA o st-HL o st-IO o st-L2*)) *A c*
⟨*proof*⟩

lemmas *ac-corres-chain-sims* = *ac-corres-chain-sim-nd-monad ac-corres-chain-sim-exception-monad*

definition *FUNCTION-BODY-NOT-IN-INPUT-C-FILE* ≡ *fail*

lemma [*polish*]:
guard (λ*s*. *UNDEFINED-FUNCTION*) = *FUNCTION-BODY-NOT-IN-INPUT-C-FILE*
(*FUNCTION-BODY-NOT-IN-INPUT-C-FILE* >>= *m*) = *FUNCTION-BODY-NOT-IN-INPUT-C-FILE*
unknown >>= (λ*x*. *FUNCTION-BODY-NOT-IN-INPUT-C-FILE*) = *FUNCTION-BODY-NOT-IN-INPUT-C-FILE*
liftE FUNCTION-BODY-NOT-IN-INPUT-C-FILE = *FUNCTION-BODY-NOT-IN-INPUT-C-FILE*
⟨*proof*⟩

lemmas *ac-statistics-rewrites* =

L1-seq-def

L2-defs'

There might be unexpected simplification 'unfolding' of *id* due to eta-expansion: *id* might be expanded (e.g. by resolution to) *id*. Now the simp rule *id ?x* = *?x* triggers and rewrites it λ*x*. *x*. Folding this back to *id* might help in those cases.

named-theorems
l1-corres **and** *l2-corres* **and** *io-corres* **and** *hl-corres* **and** *wa-corres* **and** *ts-corres* **and** *ac-corres* **and**

known-function-corres **and** *known-function*

lazy-named-theorems
l1-def **and** *l2-def* **and** *io-def* **and** *hl-def* **and** *wa-def* **and** *ts-def* **and** *ac-def*
named-theorems
heap-update-syntax

lemma *fold-id*: $(\lambda x.\ x) = id$
⟨*proof*⟩

lemma *fold-id-unit*: $(\lambda\text{-}.\ ()) = id$
⟨*proof*⟩

⟨*ML*⟩

context *globals-stack-heap-state*
begin
⟨*ML*⟩
end

⟨*ML*⟩

end

Part IV

Documentation

Chapter 25

Quickstart

Japheth Lim, Rohan Jacob-Rao, David Greenaway and Norbert Schirmer

25.1 Introduction

AutoCorres is a tool that attempts to simplify the formal verification of C programs in the Isabelle/HOL theorem prover. It allows C code to be automatically abstracted to produce a higher-level functional specification.

AutoCorres relies on the C-Parser [7] developed by Michael Norrish at NICTA. This tool takes raw C code as input and produces a translation in SIMPL [8], an imperative language written by Norbert Schirmer on top of Isabelle. AutoCorres takes this SIMPL code to produce a "monadic" specification, which is intended to be simpler to reason about in Isabelle. The composition of these two tools (AutoCorres applied after the C-Parser) can then be used to reason about C programs.

This guide is written for users of Isabelle/HOL, with some knowledge of C, to get started proving properties of C programs. Using AutoCorres in conjunction with the verification condition generator (VCG) *runs-to-vcg*, one should be able to do this without an understanding of SIMPL nor all the details of the monadic representation produced by AutoCorres. We will see how this is possible in the next chapter.

25.2 A First Proof with AutoCorres

We will now show how to use these tools to prove correctness of some very simple C functions.

25.2.1 Two simple functions: `min` and `max`

Consider the following two functions, defined in a file `minmax.c`, which (we expect) return the minimum and maximum respectively of two unsigned integers.

```
unsigned min(unsigned a, unsigned b)
{
    if (a <= b) {
        return a;
    } else {
        return b;
    }
}

unsigned max(unsigned a, unsigned b)
{
    return UINT_MAX - (
                min(UINT_MAX - a, UINT_MAX - b));
}
```

It is easy to see that `min` is correct, but perhaps less obvious why `max` is correct. AutoCorres will hopefully allow us to prove these claims without too much effort.

25.2.2 Invoking the C-parser

As mentioned earlier, AutoCorres does not handle C code directly. The first step is to apply the C-Parser by using **install-C-file** to obtain a SIMPL translation. We do this using the `install-C-file` command in Isabelle, as shown.

install-C-file *sources/minmax.c*

For every function in the C source file, the C-Parser generates a corresponding Isabelle definition. These definitions are placed in an Isabelle "locale", whose name matches the input filename. For our file *minmax.c*, the C-Parser will place definitions in the locale *minmax*.[1]

For our purposes, we just have to remember to enter the appropriate locale before writing our proofs. This is done using the `context` keyword in Isabelle.

Let's look at the C-Parser's outputs for `min` and `max`, which are contained in the theorems `min_body_def` and `max_body_def`. These are simply definitions of the generated names *min_body* and *max_body*. We can also see here how our work is wrapped within the *minmax* context.

context *minmax-simpl* **begin**

[1] The C-parser uses locales to avoid having to make certain assumptions about the behaviour of the linker, such as the concrete addresses of symbols in your program.

thm *min-body-def*

min-body ≡
TRY
  lvar-nondet-init (λ*upd*. *min.ret*′:=$_{\mathcal{L}}$ *upd*);;
  IF ´*min.a* ≤ ´*min.b THEN*
    creturn global-exn-var′*-*′*-update* (λ*upd*. *min.ret*′:=$_{\mathcal{L}}$ *upd*)
     (λ*s*. *s* ·$_{\mathcal{L}}$ *min.a*)
  ELSE
    creturn global-exn-var′*-*′*-update* (λ*upd*. *min.ret*′:=$_{\mathcal{L}}$ *upd*)
     (λ*s*. *s* ·$_{\mathcal{L}}$ *min.b*)
  FI;;
  Guard DontReach {} *SKIP*
CATCH ccatchreturn global-exn-var′*-*′
END

thm *max-body-def*

max-body ≡
TRY
  lvar-nondet-init (λ*upd*. *max.ret*′:=$_{\mathcal{L}}$ *upd*);;
  (´*max.ret*′*unsigned*′*1* :== *CALL*$_e$ *minmax.min*(− *1* − ´*max.a*,− *1* − ´*max.b*);;
   creturn global-exn-var′*-*′*-update* (λ*upd*. *max.ret*′:=$_{\mathcal{L}}$ *upd*)
    (λ*s*. − *1* − (*s* ·$_{\mathcal{L}}$ *max.ret*′*unsigned*′*1*)));;
  Guard DontReach {} *SKIP*
CATCH ccatchreturn global-exn-var′*-*′
END

end

The definitions above show us the SIMPL generated for each of the functions; we can see that C-parser has translated `min` and `max` very literally and no detail of the C language has been omitted. For example:

- C `return` statements have been translated into exceptions which are caught at the outside of the function's body;
- *Guard* statements are used to ensure that behaviour deemed 'undefined' by the C standard does not occur. In the above functions, we see that a guard statement is emitted that ensures that program execution does not hit the end of the function, ensuring that we always return a value (as is required by all non-`void` functions).
- Function parameters are modelled as local variables, which are setup prior to a function being called. Return variables are also modelled as local variables, which are then read by the caller.

While a literal translation of C helps to improve confidence that the translation is sound, it does tend to make formal reasoning an arduous task.

25.2.3 Invoking AutoCorres

Now let's use AutoCorres to simplify our functions. This is done using the **autocorres** command, in a similar manner to the **install-C-file** command:

autocorres *minmax.c*

AutoCorres produces a definition in the `minmax` locale for each function body produced by the C parser. For example, our `min` function is defined as follows:

context *minmax-all-corres* **begin**
thm *min'-def*

$$min'\ ?a\ ?b \equiv if\ ?a \leq ?b\ then\ ?a\ else\ ?b$$

Each function's definition is named identically to its name in C, but with a prime mark (`'`) appended. For example, our functions `min` above was named *min'*, while the function `foo_Bar` would be named *foo-Bar'*.

AutoCorres does not require you to trust its translation is sound, but also emits a *correspondence* or *refinement* proof, as follows:

thm *min'-ac-corres*

$$\begin{aligned}&ac\text{-}corres'\ (\lambda\text{-}.\ Exception\ ())\ (lift\text{-}global\text{-}heap \circ (\lambda s.\ s) \circ globals)\ True\\ &\{AssumeError,\ StackOverflow\}\ \Gamma\ (\lambda s.\ s \cdot_{\mathcal{L}} min.ret')\ global\text{-}exn\text{-}var'\text{-}'\\ &((\lambda s.\ s \cdot_{\mathcal{L}} min.a = ?a')\ and\ (\lambda s.\ s \cdot_{\mathcal{L}} min.b = ?b')\ and\\ &\ (\lambda\text{-}.\ True) \circ globals\ and\\ &\ (\lambda x.\ abs\text{-}var\ ?a\ id\ ?a' \wedge abs\text{-}var\ ?b\ id\ ?b') \circ lift\text{-}global\text{-}heap \circ (\lambda s.\ s) \circ\\ &\ globals)\\ &(return\ (min'\ ?a\ ?b))\ (Call\ minmax.min)\end{aligned}$$

Informally, this theorem states that, assuming the abstract function *min'* can be proven to not fail for a partciular input, then for the associated input, the concrete C SIMPL program also will not fault, will always terminate, and will have a corresponding end state to the generated abstract program.

For more technical details, see [2] and [3] or chapter 26.

25.2.4 Verifying `min`

In the abstracted version of *min'*, we can see that AutoCorres has simplified away the local variable reads and writes in the C-parser translation of `min`, simplified away the exception throwing and handling code, and also simplified away the unreachable guard statement at the end of the function. In fact, *min'* has been simplified to the point that it exactly matches Isabelle's built-in function *min*:

thm *min-def*

min ?a ?b = (if ?a ≤ ?b then ?a else ?b)

So, verifying *min′* (and by extension, the C function `min`) should be easy:

lemma *min′-is-min*: *min′ a b = min a b*
⟨*proof*⟩

25.2.5 Verifying `max`

Now we also wish to verify that *max′* implements the built-in function *max*. *min′* was nearly too simple to bother verifying, but *max′* is a bit more complicated. Let's look at AutoCorres' output for `max`:

thm *max′-def*

max′ ?a ?b ≡ − 1 − min′ (− 1 − ?a) (− 1 − ?b)

At this point, you might still doubt that *max′* is indeed correct, so perhaps a proof is in order. The basic idea is that subtracting from `UINT_MAX` flips the ordering of unsigned ints. We can then use *min′* on the flipped numbers to compute the maximum.

The next lemma proves that subtracting from `UINT_MAX` flips the ordering. To prove it, we convert all words to *int*'s, which does not change the meaning of the statement.

lemma *n1-minus-flips-ord*:
((a :: word32) ≤ b) = ((−1 − a) ≥ (−1 − b))
⟨*proof*⟩

And now for the main proof:

lemma *max′-is-max*: *max′ a b = max a b*
⟨*proof*⟩

end

In the next section, we will see how to use AutoCorres to simplify larger, more realistic C programs.

25.3 More Complex Functions with AutoCorres

In the previous section we saw how to use the C-Parser and AutoCorres to prove properties about some very simple C programs.

Real life C programs tend to be far more complex however; they read and write to local variables, have loops and use pointers to access the heap. In this section we will look at some simple programs which use loops and access the heap to show how AutoCorres can allow such constructs to be reasoned about.

25.3.1 A simple loop: `mult_by_add`

Our C function `mult_by_add` implements a multiply operation by successive additions:

```
unsigned mult_by_add(unsigned a, unsigned b)
{
    unsigned result = 0;
    while (a > 0) {
        result += b;
        a--;
    }
    return result;
}
```

We start by translating the program using the C parser and AutoCorres, and entering the generated locale `mult_by_add`.

install-C-file *sources/mult-by-add.c*
autocorres [*ts-rules = nondet*] *mult-by-add.c*

The C parser produces the SIMPL output as follows:

thm *mult-by-add-body-def*

```
mult-by-add-body ≡
TRY
  lvar-nondet-init (λupd. ret'':=ℒ upd);;
  (´result :== SCAST(32 signed → 32) 0;;
   (WHILE SCAST(32 signed → 32) 0 < ´a DO
      ´result :== ´result + ´b;;
      ´a :== ´a − SCAST(32 signed → 32) 1
    OD;;
    creturn global-exn-var''-update (λupd. ret'':=ℒ upd)
     (λs. s ·ℒ result)));;
  Guard DontReach {} SKIP
CATCH ccatchreturn global-exn-var''
END
```

Which is abstracted by AutoCorres to the following:

thm *mult-by-add'-def*

```
mult-by-add' ?a ?b ≡
do {
  (a, result) ← whileLoop (λ(a, result) s. 0 < a)
                  (λ(a, result). return (a − 1, result + ?b))
                 (?a, 0);
  return result
}
```

In this case AutoCorres has abstracted `mult_by_add` into a *monadic* form. Monads are a pattern frequently used in functional programming to represent imperative-style control-flow; an in-depth introduction to monads can be found elsewhere.

The monads used by AutoCorres in this example is a *non-deterministic state monad*; program state is implicitly passed between each statement, and results of computations may produce more than one (or possibly zero) results[2].

The HOL type is called specification monad: ($'e$, $'a$, $'s$) *spec-monad*, where

- $'e$ type for exception / error results,
- $'r$ type of the result and
- $'s$ type of the state.

  When $'e$ is instantiated with the unit type *unit*, there is an abbreviation ($'a$, $'s$) *res-monad*, for *result monad*, When it is instantiated with a proper type we have an abbreviation ($'e$, $'a$, $'s$) *exn-monad*, for *exception monad*.

  To uniformly model results and exceptions as values the type ($'e$, $'a$) *exception-or-result* is used. It is constructed as a sum type where we exclude a default value from the exception type $'e$. So when $'e$ is instantiated with *unit* the type is isomorphic to the result type $'a$. Type *unit* is only inhabited by the unique unit value () which is also default value. This instance is abbreviated with $'a$ *val*. For proper exceptions we instantiate $'e$ with an option type $'x$ *option*. This instance is abbreviated with ($'x$, $'a$) *xval*.

  Constructors and pattern matching:

- ($'e$, $'a$) *exception-or-result*:*Exception e*, *Result v*, and pattern matching $\lambda x.$ *case x of Exception e* $\Rightarrow$ *f e* | *Result v* $\Rightarrow$ *g v*
- $'a$ *val*:*Result v*, with pattern matching λ*Res v*. *g v*
- ($'e$, $'a$) *xval*: *Exn e*, *Result v*, and pattern matching $\lambda x.$ *case x of Exn e* $\Rightarrow$ *f e* | *Result v* $\Rightarrow$ *g v*

  This enconding allows us to give a uniform definition of the monadic (>>=) which works for the generic specification monad and thus also for its instances the result and the exception monad.

[2]Non-determinism becomes useful when modelling hardware, for example, where the exact results of the hardware cannot be determined ahead of time.

'Hoare Triples'

The bulk of *mult-by-add′* is wrapped inside the *whileLoop monad combinator*, which is really just a fancy way of describing the method used by AutoCorres to encode (potentially non-terminating) loops using monads.

If we want to describe the behaviour of this program, we can use Hoare logic as follows:

$$P\ s \Longrightarrow \textit{mult-by-add}'\ a\ b \cdot s \ \{\!|\ Q\ |\!\}$$

This predicate states that, assuming P holds on the initial program state, if we execute *mult-by-add′ a b*, then Q will hold on the final state of the program.

There are a few details: while P has type $'s \Rightarrow \textit{bool}$ (i.e., takes a state and returns true if it satisifies the precondition), Q has an additional parameter $'r \Rightarrow 's \Rightarrow \textit{bool}$. The additional parameter $'r$ is the *return value* of the function; so, in our *mult-by-add′* example, it will be the result of the multiplication. Note that the precondition is not part of the 'hoare-triple' but is an ordinary Isabelle assumption on the initial state *s*. That way we can directly use Isabelle/Isar to decompose the precondition and can also refer to the initial state within the postcondition. The foundational constant for this hoare triple is *runs-to* which means total correctness: we have to proof that the program terminates and that there is no undefined behaviour (all guards must hold). There is also a weaker notion for partial correctness *runs-to-partial* with syntax $f \cdot s\ ?\{\!|\ Q\ |\!\}$.

For example one useful property to prove on our program would be:

$$\textit{mult-by-add}'\ a\ b \cdot s \ \{\!|\ \lambda \textit{Res}\ r\ \text{-}.\ r = a * b\ |\!\}$$

That is, for all possible input states, our `mult_by_add'` function returns the product of *a* and *b*.

Our proof of *mult-by-add′* could then proceed as follows:

lemma *mult-by-add-correct*:
$\textit{mult-by-add}'\ a\ b \cdot s \ \{\!|\ \lambda \textit{Res}\ r\ \text{-}.\ r = a * b\ |\!\}$

Unfold abstracted definition

⟨*proof*⟩

The main tool is the proof method *runs-to-vcg*, a verification condition generator for monadic programs. It uses weakest precondition style rules which are collected in named theorems collection *runs-to-vcg*. In that collection there is no rule for *whileLoop*. The verification generator will stop there unless you specify a rule to use. This was done in the proof above by instantiating the rule *runs-to-whileLoop-res* with an invariant and a measure and by adding it to the verification condition generator via attribute *runs-to-vcg*. We finally discharge the remaining subgoals left from *runs-to-vcg* with standard proof tools of Isabelle.

In the next section, we will look at how we can use AutoCorres to verify a C program that reads and writes to the heap.

25.3.2 `swap`

Here, we use AutoCorres to verify a C program that reads and writes to the heap. Our C function, `swap`, swaps two words in memory:

```
void swap(unsigned *a, unsigned *b)
{
    unsigned t = *a;
    *a = *b;
    *b = t;
}
```

Again, we translate the program using the C parser and AutoCorres.

install-C-file *sources/swap.c*
autocorres [*heap-abs-syntax*, *ts-rules* = *nondet*] *swap.c*

Most heap operations in C programs consist of accessing a pointer. AutoCorres abstracts the global C heap by creating one heap for each type. (In our simple `swap` example, it creates only a *Word-32.word32* heap.) This makes verification easier as long as the program does not access the same memory as two different types.

There are other operations that are relevant to program verification, such as changing the type that occupies a given region of memory. AutoCorres will not abstract any functions that use these operations, so verifying them will be more complicated (but still possible).

The C parser expresses `swap` like this:

thm *swap-body-def*

```
swap-body ≡
TRY
  Guard C-Guard {|c-guard ´a|} (´t :== h-val (hrs-mem ´t-hrs) ´a);;
  (Guard C-Guard {|c-guard ´a|}
    (Guard C-Guard {|c-guard ´b|}
      (´globals :==
        t-hrs-'-update
         (hrs-mem-update (heap-update ´a (h-val (hrs-mem ´t-hrs) ´b)))));;
   Guard C-Guard {|c-guard ´b|}
    (´globals :== t-hrs-'-update (hrs-mem-update (heap-update ´b ´t))))
CATCH ccatchreturn global-exn-var'-'
END
```

AutoCorres abstracts the function to this:

thm *swap'-def*

```
swap' ?a ?b ≡ do {
  guard (λs. IS-VALID(32 word) s ?a);
  guard (λs. IS-VALID(32 word) s ?b);
  t ← gets (λs. s[?a]);
  modify (λs. s[?a := s[?b]]);
  modify (λs. s[?b := t])
}
```

There are some things to note:

The function contains guards (assertions) that the pointers `a` and `b` are valid. We need to prove that these guards never fail when verifying `swap`. Conversely, when verifying any function that calls `swap`, we need to show that the arguments are valid pointers.

We saw a monadic program in the previous section, but here the monad is actually being used to carry the program heap.

Now we prove that `swap` is correct. We use x and y to "remember" the initial values so that we can talk about them in the post-condition. The heap access syntax $s[a]$ is used to select the split heap *heap-w32* from state s at pointer a. [3]

lemma $[\![IS\text{-}VALID(32\ word)\ s\ a;\ s[a] = x;\ IS\text{-}VALID(32\ word)\ s\ b;\ s[b] = y]\!] \Longrightarrow$
$\quad swap'\ a\ b \cdot s$
$\quad \{\!|\ \lambda\text{-}\ s.\ s[a] = y \wedge s[b] = x\ |\!\}$
$\langle proof \rangle$

Note that we have "only" proved that the function swaps its arguments. We have not proved that it does *not* change any other state. This is a typical *frame problem* with pointer reasoning. We can prove a more complete specification of `swap`:

lemma $[\![(\bigwedge x\ y\ s.\ P\ (s[a := x][b := y]) = P\ s);$
$\quad IS\text{-}VALID(32\ word)\ s\ a;\ s[a] = x;\ IS\text{-}VALID(32\ word)\ s\ b;\ s[b] = y\ ;\ P\ s\ ]\!]$
$\Longrightarrow$
$\quad (swap'\ a\ b) \cdot s$
$\quad \{\!|\ \lambda\text{-}\ s.\ s[a] = y \wedge s[b] = x \wedge P\ s\ |\!\}$
$\langle proof \rangle$

In other words, if predicate P does not depend on the inputs to `swap`, it will continue to hold.

Separation logic provides a more structured approach to this problem.

[3] For more details on the split heap model in autocorres see section 26.21.

25.4 Command Options and Invocation

25.4.1 Session Structure

The supplied session structure has some peculiarities to accomodate the AFP. The AFP presents the documentation of the session that is named like the AFP-entry. This is `AutoCorres2`. This session also includes this documentation and other example application of AutoCorres. When you want to use AutoCorres for your own projects you do not have to import these examples. Thats why we also supply the leaner `AutoCorres2_Main` as entry point, which we recommend as parent session for your applications.

25.4.2 C-Parser

Options for **install-C-file**,
syntax `install_C_file <filename> [<option> = value, ...]`:

- `memsafe` (default false): add additional guards to ensure that well-typedness of pointers
- `c_types` (default true): import types to UMM model
- `c_defs` (default true): import function to SIMPL
- `roots`: (default all functions): List of C functions that are the root functions to import
- `prune_types` (default true): only import those types that are actually used in the program
- `machinety`: HOL type for ghost state modelling the machine
- `gostty`: HOL type for some additional ghost state
- `skip_asm` (default false): Skip inline assembler

Moreover there are some Isabelle options for more feedback / tracing:

- `c_parser_feedback_level` (default 0), higher number means more tracing
- `option c_parser_verbose` (default false), enable for verbose messages

Multiple C Files

Command **install-C-file** only has a single C file as argument. When your project consists of several `.c` and `.h` you can register those dependencies first, with command **include-C-file**. The main C file, which is the argument to **install-C-file**, can then include all these files. If there is no such C file in your project you can create one to accomodate **install-C-file**.

Target Architecture Selection `L4V_ARCH`

The Environment variable `L4V_ARCH` can be used to determine the target architecture which influences the sizes of C integral types and pointer types. Supported platforms are `ARM` (default), `ARM64`, `ARM_HYP`, `RISCV64` and `X64`.

For `ARM`, the sizes are:

- 128 bits: `int128`
- 64 bits: `long long`
- 32 bits: pointers, `long`, `int`
- 16 bits: `short`

For `X64`, `ARM64`, `ARM_HYP`:

- 128 bits: `int128`
- 64 bits: pointers, `long long`, `long`
- 32 bits: `int`
- 16 bits: `short`

For `ARM64` `char` is signed.

For example to build AutoCorres for ARM64:

```
L4V_ARCH=ARM64 isabelle build -d . AutoCorres_Main
```

25.4.3 AutoCorres

Options for **init−autocorres** / **autocorres**,
syntax `autocorres [option = value, ...] <filename>`:

- `phase`: perform autocorres up to specified phase only (L1, L2, HL, WA, IO, TS)
- `scope`: space separated list of functions to perform autocorres on. (Default all).

- `scope_depth`: depth of callees to also also include
- `single_threaded`: flag to disable parallel processing (e.g. to make more sense out of tracing messages)
- `no_heap_abs`: space separated list of functions that should be excluded from heap abstraction
- `in_out_parameters`: 'and' separated list of function specs e.g. `inc(y:in_out)`, cf. `../In_Out_Parameters_Ex.thy`
- `in_out_globals`: 'and' separated list of functions which modify global variables via pointers
- `skip_io_abs`: flag to disable IO abstraction
- `addressable_fields`: space separated list of paths to struct fields that should be addressable in the split heap, cf. `../open_struct.thy`
- `ignore_addressable_fields_error`: option to downgrade error to a warning
- `skip_heap_abs`: flag to disable heap abstraction (into split heap)
- `unsigned_word_abs`: space separated list of functions where unsigned words should be abstracted to *nat*.
- `unsigned_word_abs_known_functions`: assume unsigned word abstraction for function pointers
- `no_signed_word_abs`: space separated list of functions where abstraction of unsigned word to *int* should be disabled.
- `no_signed_word_abs_known_functions`: don't assume unsigned word abstraction for function pointers
- `skip_word_abs`: flag to disable word abstraction
- `ts_rules`: space separated list of rules to consider during TS phase (pure, gets, option, nondet, exit).
- `ts_force <rule-name>`: space separated list of function names to put in the specified monad (pure, gets, option, nondet, exit).
- `ts_force_known_functions`: assume function pointers live in the specified monad (pure, gets, option, nondet, exit)
- `heap_abs_syntax`: enable some additional syntax for heap accesses
- `do_polish` (default true): flag for polish phase

- `L1_opt` (RAW | PEEP (default)): level for L1 optimisation
- `L2_opt` (RAW | PEEP (default)): level for L2 optimisation
- `HL_opt` (RAW | PEEP (default)): level for HL optimisation
- `WA_opt` (RAW | PEEP (default)): level for WA optimisation
- `trace_opt`: flag to enable some tracing
- `gen_word_heaps`: flag to generate word heaps even if the program does not use them
- `keep_going`: continue despite some errors
- `lifted_globals_field_prefix`: custom prefix for split heap naming
- `lifted_globals_field_suffix`: custom suffix for split heap naming
- `function_name_prefix`: custom prefix for generated function names
- `function_name_suffix`: custom suffix for generated function names
- `no_c_termination`: flag to disable termination precondition for correspondence proofs
- `unfold_constructor_bind`: (Selectors (default) | Always | Never) to give some user level control to unfold certain "simple" binds. cf.: `../../tests/proof-tests/unfold_bind_options.thy`
- `base_locale`: custom base locale for all autocorres locales

Moreover there are some Isabelle options for more feedback / tracing:

- `option verbose` (default 0), higher value means more verbosity
- `verbose_timing` (default 0), higher value means more timing messages
- `timing_threshold` (default 3), threshold in milliseconds for timing messages

Chapter 26

Overview of AutoCorres

theory *AutoCorresInfrastructure*
imports *AutoCorres*
begin

This theory elaborates on some of the (internal) AutoCorres infrastructure and is also supposed to act as a testbench for this infrastructure, e.g. simplifier setup.

Following the Isabelle tradition user relevant changes to AutoCorres are described in the file `../NEWS`.

26.1 Building Blocks

'AutoCorres' has the following major building blocks and contributions.

- The 'C-parser' (by Michael Norrish) takes C-Input files and parses them to 'SIMPL' (by Norbert Schirmer) programs within Isabelle/HOL:
  - C-parser: chapter 28
  - SIMPL: https://www-wjp.cs.uni-saarland.de/leute/private_homepages/nschirmer/pub/schirmer_phd.pdf
- The unified memory model (UMM) (by Harvey Tuch) models C-types as HOL-types with conversions between the typed view and the raw-byte-level view. Moreover it provides a separation logic framework.
  - UMM: https://trustworthy.systems/publications/papers/Tuch%3Aphd.pdf
- AutoCorres (by David Greenaway): https://trustworthy.systems/publications/nicta_full_text/8758.pdf

Some historic remarks, that might give some insight why things are as they are. The motivation for AutoCorres was the C-verification tasks coming from the sel4 Projects https://sel4.systems, a verified microkernel, written in C.

The project started with high level HOL-specifications, refining them to monadic functions (within HOL and Haskell) and then refining them to C-code. The nondetermistic state-monad goes back to Cook. et al http://www.cse.unsw.edu.au/~kleing/papers/Cock_KS_08.pdf. To link the C-code with the monadic functions within Isabelle / HOL the C-parser was written. The C-parser produces SIMPL programs, and SIMPL is equipped with various semantics (big and small-step) and a Hoare-logic framework including a verification condition generator. The correspondence of the C-functions represented in SIMPL and the monadic functions was manually expressed within this framework.

AutoCorres was then built after (or in parallel) to this verification effort in order to speed up the process for future projects.

While the main concepts described in the publications above are still valid, a lot of things have evolved and were extended. Especially the implementation details have changed. In this document we also give some notes on these differences. Also see `../NEWS`.

26.2 C Parser

Some remarks on the C-Parser

- Lexical and syntactical analysis is implemented using ML-Lex / ML-Yacc coming from the MLton project http://mlton.org/. Originally mlton was used to generate the ML files from the grammers. Meanwhile this is all integrated into Isabelle/ML.
- The C parser first generates an ML-data structure representing the program. It then does some transformations on that data structure (e.g. storing results of nested function calls into temporary variables), before translating it to SIMPL. In roughly the following steps:

  1. Generating HOL-types for the C-types.
     - The C parser performs a complete program analysis to determine the types used.
     - Defines the types and performs the UMM proofs, in particular conversions and properties to and from raw byte lists.
  2. Defining the state-space type: global variables, local variables, heap variables. The C parser does a complete program analysis to determine how global and heap variables are used to decide

whether to model them as part of the globals-record or the heap. When no 'address-of' a variable is calculated it is stored as part of the global variable record, otherwise it is stored within the heap. Global variables are more abstract to deal with compared to heap variables.

3. Defining the SIMPL procedures
4. Proving 'modifies' specifications for the procedures, i.e. frame conditions on global variables.

26.3 AutoCorres

AutoCorres abstracts the SIMPL program to a monadic HOL function in various phases, producing correspondence (a.k.a. simulation, a.k.a. refinement) theorems along the way. Note that the translation (and the correspondence theorems) might only be partial in the following sense: A phase might introduce additional guards into the code. Simulation only holds for programs that do not fail on the abstract execution. Note that nontermination is currently also modelled as failure. So simulation only holds for terminating abstract programs:

- Correspondence relation: *ac-corres*

Each phase is processed in similar stages:

- Raw translation and refinement-proof
- Optimisation of the output program, sometimes distinguished between a more lightweight "peephole" optimisation and a more heavyweight "flow" optimisation
- Define a constant for the output of the translation for that phase.

  The phases are:

- From SIMPL (deep embedding of statements, shallow embedding of expressions) to L1 (shallow embedding of statements and expressions). The transformation preserves the state-space representation, and merely focuses on translating SIMPL statements to monadic functions:
  - Definition of L1: `../L1Defs.thy`
  - Correspondence relation: *L1corres*
  - Peephole optimisation: `../L1Peephole.thy`
  - Flow optimisation: `../ExceptionRewrite.thy`

  - Main ML file: `../simpl_conv.ML`

- Local variable abstraction, from L1 to L2. In L2, local variables are represented as lambda abstractions. So the underlying state-space type changes, getting rid of the local variable record. As local variables are now treated as lambda abstractions, and thus names become meaningless, autocorres keeps the original names around as part of (logically redundant) name annotations within the L2 constants, e.g. *L2-gets* (λ-. x) [*c-source-name-hint*]. Lists of names are used, as variables might pile up in tuples. These annotations are removed in the final polishing phase of autocorres, attempting to rename the bound variables accordingly on the fly.

  - Definition of L2: `../L2Defs.thy`
  - Correspondence relation: *L2corres*
  - Exception rewriting, introducing nested exceptions: `../L2ExceptionRewrite.thy`
  - Peephole optimisation: `../L2Peephole.thy`
  - Main ML files: `../local_var_extract.ML`, `../l2_opt.ML` From now on we stay in the language L2, and also the optimisation stages are reused.

- In/Out parameter abstraction (IO). Pointer parameters are replaced by passing value parameters. This step may eliminate pointers to local variables.

  - Theory: `../In_Out_Parameters.thy`
  - Correspondence relation: *IOcorres*
  - Documentation: `In_Out_Parameters_Ex.thy`

- Heap abstraction, referred to as heap-lifting (HL). The monolithic-byte-oriented heap is abstracted to a typed-split-heap model. For that the new type *lifted-globals* is introduced. Note, that the separation logic of Tuch also attempts to give a typed view on top of the byte-level-heap. In case of nested structure types it even is capable to express the various levels of typed-heap views from bytes to individual structure fields to (nested) structures. AutoCorres takes a simpler, more pragmatic approach here and only provides a single split-heap abstraction on the level of complete types. This means there is a heap for every compound type and for every primitive type but not for nested structures. However, autocorres allows to mix between the abstractions layers (split-heap vs. byte-heap) on the granularity of functions. So you can model low-level functions like memory-allocators on the

untyped byte-heap and then still use these functions within other functions in the split-heap abstraction. Meanwhile, genuine support for a split-heap approach with addressable nested structures was added. It is described in `open_struct.thy`.

  - Main theory: `../HeapLift.thy`
  - Correspondence relation: *L2Tcorres*
  - Main ML files: `../heap_lift_base.ML`, `../heap_lift.ML`

- Word abstraction (WA): (un)signed n-bit words are converted to *int* or *nat*. This conversion only affects local variable (lambda abstraction), so the underlying state-space type is maintained. When reading and storing to memory the values are converted accordingly.

  - Main theory: `../WordAbstract.thy`
  - Correspondence relation: *corresTA*
  - Main ML file: `../word_abstract.ML`

- Type strengthening (TS), sometimes also referred to as type lifting or type specialisation: Best effort approach to simplify the structure of the monad as far as possible:

  - Pure functions
  - Reader monad, read-only state-dependant functions.
  - Option (reader) monad, in particular for potential non-terminating functions (recursion / while). Nontermination is treated as failure in autocorres and failure of guards are represented as a result of *None*
  - Nondeterministic state monad (exceptional control flow confined within function boundaries). Failure (of guards) and nontermination are represented by *Failure* as outcome.
  - Nondeterministic state monad with exit (exceptional control flow crossing function boundaries). New monad types can be registered within Isabelle / ML.
  - Main theories: `../TypeStrengthen.thy`, `../Refines_Spec.thy`
  - Correspondence relation: *refines*. Originally this was actually based on equality. As we now implement (mutually) recursive functions by a CCPO **fixed-point** instead of introducing an explicit measure parameter we changed to simulation. Equality is not *ccpo.admissible*, whereas simulation is.
  - Main ML files: `../type_strengthen.ML`, `../monad_types.ML`, `../monad_convert.ML`

- Polish. Performed as a part of type strengthening. Here remaining special and limited L2 definitions are expanded to 'ordinary' monadic functions. Also annotations of original c variable names are removed and bound variables are named accordingly.

26.4 AutoCorres Flow

The original AutoCorres implementation of David Greenaway was refined in several ways. In particular from a high level perspective the concepts of incremental build and parallel processing where unified:

- Parallel processing is moved to the outermost invocation of autocorres. All tasks are calculated respecting the dependencies of the call graph and the various autocorres phases, cf. `AutoCorres.parallel_autocorres`.
- Every task then invokes a distinct call to `AutoCorres.do_autocorres` exactly specifying the scope (which function clique) and which phase via the options.
- The results of an invocation are maintained in generic data, such that they are available to subsequent dependent calls.

  The common parts of each phase are implemented in `AutoCorresUtil`. The last phase, Type strengthening (TS), historically played a special role and still does not make much use of `AutoCorresUtil`, but conceptually it follows the same idea:

  Functions are processed in a sequence of cliques, from the bottom of the call graph to the top. Here a clique is either a single function or a group of strongly connected recursive calls. After processing of a phase the clique might get splitted due to dead code elimination. This general idea is implemented in `AutoCorresUtil.convert_and_define_cliques`.

26.4.1 General Remarks

- The main autocorres files, putting everything together: `../AutoCorres.thy`, `../autocorres.ML`
- There are two main approaches in a single phase to come up with an abstract program (output of the phase) and the correspondence proof to the concrete program (input of the phase):
  - Implicit synthesis of the abstract program from the concrete program by applying "intro" rules, where the abstract program is initialised with a schematic variable. Prototypical examples are the heap abstraction, word abstraction and type strengthening.

  - First explicitly calculate the abstract program (fragment) within ML from the concrete program (fragment), and then perform the correspondence proof. An prototypical example is the local variable abstraction.

- The common aspects of the various stages within a single phase is (partially) generalised into library functions: `../autocorres_util.ML`

26.4.2 Links to more documentation / examples

- Pointers to local variables: `pointers_to_locals.thy`
- In-Out parameters: `In_Out_Parameters_Ex.thy`
- Addressable fields and open types: `open_struct.thy`
- Function pointers: `fnptr.thy`
- Unions: `union_ac.thy`

26.5 Overview on the Locales

The representation of local variables in SIMPL and L1 where changed from records to 'Statespaces' as implemented by **statespace** and finally to positional variables as in *locals* in `../c-parser/CLocals.thy`. This allows for an uniform representation of local variables as a function from 'nat to byte list', Typing is achieved by ML data organised in locales and bundles, cf. `../c-parser/CTranslationInfrastructure.thy`. These are also used to do the L1 and the L2 correspondence proofs. Starting from L2 on the local variables are represented as lambda bindings.

We describe the various locales (and bundles) later on with our running examples.

To support function pointers even more locales where introduced. See `fnptr.thy`.

26.6 Example Program

declare [[*c-parser-feedback-level=1*]]
install-C-file *autocorres-infrastructure-ex.c*
print-theorems
thm *upd-lift-simps*
thm *fl-ti-simps*

The variables consist of parameters (input and output) and the local variables. Note that this is merely a ML-declaration of the scope which can be accesses via a bundle, cf. `../c-parser/CTranslationInfrastructure.thy`.

context includes *add-variables*
begin
term *´n :== 42*
end

Adresses of global variables or function pointers are kept in the *global-addresses* locale.

print-locale *autocorres-infrastructure-ex-global-addresses*

This is everything necessary to define the body of a function. All bodies are defined in that locale.

context *autocorres-infrastructure-ex-global-addresses*
  opening *add-variables*
begin
term *add-body*
thm *add-body-def*
end

The implementation locale holds the defining equation of the function in SIMPL and is closed under callees.

context *call-add-impl*
begin
thm *add-impl*
thm *call-add-impl*
end

In case of a clique there is a clique locale and aliases for each function.

print-locale *even-odd-impl*
print-locale *even-impl*
print-locale *odd-impl*

The locale importing all implementations is named like the file with suffix *-simpl*.

print-locale *autocorres-infrastructure-ex-simpl*

26.6.1 Incremental Build

autocorres [*phase=L1*, *scope=add*] *autocorres-infrastructure-ex.c*
autocorres [*phase=L2*, *scope=add*] *autocorres-infrastructure-ex.c*
autocorres [*phase=HL*, *scope=add*] *autocorres-infrastructure-ex.c*
autocorres [*phase=WA*, *scope=add*] *autocorres-infrastructure-ex.c*
autocorres [*phase=TS*, *scope=add*] *autocorres-infrastructure-ex.c*

26.6.2 All the rest

autocorres *autocorres-infrastructure-ex.c*

context *unsigned-to-signed-impl*
begin
thm *unsigned-to-signed-body-def*
end
context *autocorres-infrastructure-ex-all-corres*
begin
thm *unsigned-to-signed′-def*

The SIMPL versions produced by c-parser

declare [[*clocals-short-names*]]
thm *max-body-def*
thm *add-body-def*
term *add-body*::((*globals*, *locals*, *32 signed word*) *CProof.state*, *unit ptr*, *strictc-errortype*) *com*
thm *call-add-body-def*
thm *seq-assign-body-def*
thm *call-seq-assign-body-def*
thm *inc-g-body-def*
thm *deref-body-def*
thm *deref-g-body-def*
thm *factorial-body-def*
thm *call-factorial-body-def*
thm *odd-body-def*
thm *even-body-def*
thm *dead-f-body-def*
thm *dead-h-body-def*
thm *dead-g-body-def*

L1 versions. Monadic with same state. Note that recursive procedures are defined with the **fixed-point** package and hence the *.simps* instead of *-def* makes more sense to look at. In this layer a first exception elimination optimisation takes places. The optimized definition has the extension "opt".

declare [[*show-types=false*]]
thm *l1-max′-def*
thm *l1-opt-max′-def*

term *l1-add′*::(*unit*, *unit*, (*globals*, *locals*, *32 signed word*) *CProof.state*) *exn-monad*
thm *l1-add′-def*
thm *l1-opt-add′-def*

thm *l1-call-add′-def*
thm *l1-opt-call-add′-def*

thm *l1-seq-assign′-def*
thm *l1-opt-seq-assign′-def*

thm *l1-call-seq-assign'-def*
thm *l1-opt-call-seq-assign'-def*

thm *l1-inc-g'-def*
thm *l1-opt-inc-g'-def*

thm *l1-deref'-def*
thm *l1-opt-deref'-def*

thm *l1-deref-g'-def*
thm *l1-opt-deref-g'-def*

thm *l1-factorial'.simps*
thm *l1-opt-factorial'-def*

thm *l1-call-factorial'-def*
thm *l1-opt-call-factorial'-def*

thm *l1-odd'.simps*
thm *l1-opt-odd'-def*

thm *l1-even'.simps*
thm *l1-opt-even'-def*

thm *l1-dead-f'.simps*
thm *l1-opt-dead-f'-def*

thm *l1-dead-h'.simps*
thm *l1-opt-dead-h'-def*

thm *l1-dead-g'.simps*
thm *l1-opt-dead-g'-def*

L2 version: lambda bindings for local variables

thm *l2-max'-def*
term *l2-add'*::*32 signed word* ⇒ *32 signed word*
⇒ (*32 signed word c-exntype*, *32 signed word*, *globals*) *exn-monad*
thm *l2-add'-def*
thm *l2-call-add'-def*
thm *l2-seq-assign'-def*
thm *l2-call-seq-assign'-def*
thm *l2-inc-g'-def*
thm *l2-deref'-def*
thm *l2-deref-g'-def*
thm *l2-factorial'.simps*
thm *l2-call-factorial'-def*
thm *l2-odd'.simps*
thm *l2-even'.simps*
thm *l2-dead-f'.simps*

thm *l2-dead-h'.simps*
thm *l2-dead-g'-def* — Note that *l2-dead-g'* is no longer part of the clique, because of dead code elimination.

Heap abstraction. Note that the change in the heap component of the global variables. The type changes to *lifted-globals* instead of *globals*.

print-record *globals* — contains byte level tagged heap: *t-hrs-'*
print-record *lifted-globals* — contains split heap with single component *heap-w32*, as we only have a pointers to unsigend.

thm *hl-max'-def*
term *hl-add'*:: *32 signed word* ⇒ *32 signed word*
⇒ (*32 signed word c-exntype*, *32 signed word*, *lifted-globals*) *exn-monad*
thm *hl-add'-def*
thm *hl-call-add'-def*
thm *hl-seq-assign'-def*
thm *hl-call-seq-assign'-def*
thm *hl-inc-g'-def*
thm *hl-deref'-def*
thm *hl-deref-g'-def*
thm *hl-factorial'.simps*
thm *hl-call-factorial'-def*
thm *hl-odd'.simps*
thm *hl-even'.simps*
thm *hl-dead-f'.simps*
thm *hl-dead-h'.simps*
thm *hl-dead-g'-def*

Word abstraction.

thm *wa-max'-def*
term *wa-add'*::*int* ⇒ *int* ⇒ (*32 signed word c-exntype*, *int*, *lifted-globals*) *exn-monad*
thm *wa-add'-def*
thm *wa-call-add'-def*
thm *wa-seq-assign'-def*
thm *wa-call-seq-assign'-def*
thm *wa-inc-g'-def*
thm *wa-deref'-def*
thm *wa-deref-g'-def*
thm *wa-factorial'.simps*
thm *wa-call-factorial'-def*
thm *wa-odd'.simps*
thm *wa-even'.simps*
thm *wa-dead-f'.simps*
thm *wa-dead-h'.simps*
thm *wa-dead-g'-def*

Final definition.

term *max'*::*int* ⇒ *int* ⇒ *int*
thm *max'-def*

term *add′*:: *int* ⇒ *int* ⇒ *lifted-globals* ⇒ *int option*
thm *add′-def*
thm *wa-call-add′-def*
thm *wa-seq-assign′-def*
thm *wa-call-seq-assign′-def*
thm *wa-inc-g′-def*
thm *wa-deref′-def*
thm *wa-deref-g′-def*
thm *wa-factorial′.simps*
thm *wa-call-factorial′-def*
thm *wa-odd′.simps*
thm *wa-even′.simps*
thm *wa-dead-f′.simps*
thm *wa-dead-h′.simps*
thm *wa-dead-g′-def*

Correspondence theorem.

thm *factorial′-ac-corres*
thm *call-factorial′-ac-corres*

end

26.7 Simplification strategy and dealing with tuples in L2-optimization phases

Consider a congruence rule from AutoCorres

lemma
assumes *c-eq*: $\bigwedge r\ s.\ c\ r\ s = c'\ r\ s$
assumes *bdy-eq*: $\bigwedge r\ s.\ c'\ r\ s \Longrightarrow run\ (A\ r)\ s = run\ (A'\ r)\ s$
shows *L2-while c A* = *L2-while c′ A′*
⟨*proof*⟩

Note that r could be a tuple and and we would like to "split" it. The condition c will already be formulated with *case-prod*, e.g. $\lambda(x,\ y,\ z)\ s.\ x < y$.

When the congruence rule above is applied, variable *?c′* has to be instantiated at some point. First the congruence rule is applied as is. Then we use *split-paired-all* to split up the r. For the example we will end up with the subgoal:

$\bigwedge a\ b\ c\ s.\ (a < b) = ?c'\ (a,\ b,\ c)\ s$

Mere Unification will fail here as it does not work "modulo" *case-prod*. To find the proper instantiation we developed `Tuple_Tools.tuple_inst_tac` which can be added to the simplifier as a looper. It will instantiate *?c′* with the respective *case-prod* instance introducing a new schematic variable: *?c′* = $(\lambda(a,\ b,\ c).\ ?f\ a\ b\ c)$. Now the unification will kick in and do the rest of the work.

However, allthough an instantiation is now found, it does not have the expected effect on `body_eq`. The problem is, that the premise of the implication is not simplified as it is added to the "prems" of the simplifier so it will be (*case* (*a*, *b*, *c*) *of* (*a*, *b*, *c*) ⇒ λ*s*. *a* < *b*) *s* ≡ *True* and not just *a* < *b* ≡ *True*". The former format is unfortunately ineffective in the simplification of the while body.

Fortunately the simpset can be instructed to apply a function to the premise before it is added. So adding `Tuple_Tools.mksimps` with `Simplifier.set_mksimps` helps with that respect. As the simplifier descends into the term it calls the function to do "beta-reduction" of tuple application before adding the premise.

lemma *L2-while* (λ(*x*,*y*,*z*) *s*. *0* < (*y*::*nat*)) (λ(*x*,*y*,*z*). *L2-seq* (*L2-guard* (λ-. *0* < *y*)) (λ-. *X*)) *x ns*
=
L2-while (λ(*x*,*y*,*z*) *s*. *0* < *y*) (λ(*x*,*y*,*z*). *L2-seq* (*L2-guard* (λ-. *True*)) (λ-. *X*)) *x ns*
⟨*proof*⟩

Unfortunately there is still an issue. Linear arithmetic is applied as a simproc by the simplifier, e.g. *0* < *n* ⟹ *Suc 0* ≤ *n*

Unfortunately this does not work as expected in the setup above:

unbundle *clocals-string-embedding*

lemma *L2-while* (λ(*x*,*y*,*z*) *s*. *0* < (*y*::*nat*)) (λ(*x*,*y*,*z*). *L2-seq* (*L2-guard* (λ-. *Suc 0* ≤ *y* ∧ *z* = *a*)) (λ-. *X*)) *x ns*
=
L2-while (λ(*x*,*y*,*z*) *s*. *0* < *y*) (λ(*x*,*y*,*z*). *L2-seq* (*L2-guard* (λ-. *z* = *a*)) (λ-. *X*)) *x ns*
⟨*proof*⟩

After some investigation in the interplay between the simplifier and the linear arithmetic simproc it seems that the linear arithmetic somehow ignores the transformed theorem that `Tuple_Tools.mksimps` yields. Without understanding all the details I guess the reason is the missmatch between the term in the hyps and the prop of the resulting theorem. As the simplifier adds descends into an implication *P* ⟹ *Q* it generates a theorem from *P*, by also adding P to the hyps, so we have *P* [*P*] as a theorem. When `Tuple_Tools.mksimps` kicks in it only simplifies the prop not the hyps. So we end up with *a* < *b* ≡ *True* [(λ(*a*, *b*, *c*) *s*. *a* < *b*) (*a*, *b*, *c*) *s* ≡ *True*] instead of *a* < *b* ≡ *True* [*a* < *b* ≡ *True*]. This seems to irritate the linear arithmetic. One solution could be to also simplify the hyps. We did not explore this path, as I would expect some issues when the hyps are eventually discharged. Nevertheless there could be a solution following that path.

We came up with a different solution. We get rid of the dependency of `Tuple_Tools.mksimps` by using *P* =*simp*=> *Q* to trigger simplification of P.

Note the various options to trace the simplification process.

declare [[*linarith-trace=false*]]
declare [[*simp-trace=false*, *simp-trace-depth-limit=100*]]
declare [[*simp-debug=false*]]
declare [[*show-hyps=false*]]

lemma
assumes *c-eq*: $\bigwedge r\ s.\ c\ r\ s = c'\ r\ s$
assumes *bdy-eq*: $\bigwedge r\ s.\ c'\ r\ s$ =*simp*=> *run* $(A\ r)\ s = run\ (A'\ r)\ s$
shows *L2-while* $c\ A =$ *L2-while* $c'\ A'$
⟨*proof*⟩

lemma *L2-while* $(\lambda(x,y,z).\ 0 < (y::nat))$ $(\lambda(x,y,z).$ *L2-seq* (*L2-guard* $(\lambda\text{-}.$ *Suc* $0 \le y \wedge z = a))$ $(\lambda\text{-}.\ X))\ x\ ns$
=
L2-while $(\lambda(x,y,z).\ 0 < y)$ $(\lambda(x,y,z).$ *L2-seq* (*L2-guard* $(\lambda\text{-}.\ z = a))$ $(\lambda\text{-}.\ X))\ x\ ns$
⟨*proof*⟩

The more standard congruence rule also works fine.

lemma
assumes *c-eq*: $c = c'$
assumes *bdy-eq*: $\bigwedge r\ s.\ c'\ r\ s$ =*simp*=> *run* $(A\ r)\ s = run\ (A'\ r)\ s$
shows *L2-while* $c\ A =$ *L2-while* $c'\ A'$
⟨*proof*⟩

lemma *L2-while* $(\lambda(x,y,z).\ 0 < (y::nat))$ $(\lambda(x,y,z).$ *L2-seq* (*L2-guard* $(\lambda\text{-}.$ *Suc* $0 \le y \wedge z = a))$ $(\lambda\text{-}.\ X))\ x\ ns$
=
L2-while $(\lambda(x,y,z).\ 0 < y)$ $(\lambda(x,y,z).$ *L2-seq* (*L2-guard* $(\lambda\text{-}.\ z = a))$ $(\lambda\text{-}.\ X))\ x\ ns$
⟨*proof*⟩

To avoid repeated splitting (and diving into the subterms) of tuples with $(\bigwedge x.\ PROP\ ?P\ x) \equiv (\bigwedge a\ b.\ PROP\ ?P\ (a, b))$ consider the follwing simproc setup.

lemma *L2-while* $(\lambda(x,y,z).\ 0 < (y::nat))$ $(\lambda(x,y,z).$ *L2-seq* (*L2-guard* $(\lambda\text{-}.$ *Suc* $0 \le y \wedge z = a))$ $(\lambda\text{-}.\ X))\ x\ ns$
=
L2-while $(\lambda(x,y,z).\ 0 < y)$ $(\lambda(x,y,z).$ *L2-seq* (*L2-guard* $(\lambda\text{-}.\ z = a))$ $(\lambda\text{-}.\ X))\ x\ ns$
⟨*proof*⟩

This is also the setup of `L2Opt.cleanup_ss`

lemma *L2-while* $(\lambda(x,y,z).\ 0 < (y::nat))$ $(\lambda(x,y,z).$ *L2-seq* (*L2-guard* $(\lambda\text{-}.$ *Suc* $0 \le y \wedge z = a))$ $(\lambda\text{-}.\ X))\ x\ ns$
=

L2-while (λ(x,y,z) *s. 0 < y*) (λ(x,y,z). *L2-seq* (*L2-guard* (λ-. *z = a*)) (λ-. *X*)) *x ns*
⟨*proof*⟩

lemma
assumes *XX-def*: *XX = L2-while* (λ(*x*, *y*, *z*) *s. 0 < y*)
(λ(*x1*, *x2*, *x3*). *L2-seq* (*L2-guard* (λ-. *Suc 0 ≤ y1 ∧ z1 = a*)) (λ-. *X*)) *x ns*
shows *L2-while* (λ(x,y,z) *s. 0 < (y::nat)*) (λ(x,y,z). *L2-seq* (*L2-guard* (λ-. *Suc 0 ≤ y1 ∧ z1 = a*)) (λ-. *X*)) *x ns*
=
XX
⟨*proof*⟩

lemma *L2-while* (λ(*x*, *y*, *z*) *s. y = 0*)
(λ(*x*, *y*, *z*).
L2-seq (*L2-gets* (λs. *y*) [$\mathcal{S}$ ''*ret*'']) (λr. *XXX21 r x*))
names =
L2-while (λ(*x*, *y*, *z*) *s. y = 0*) (λ(*x*, *y*, *z*). *L2-seq* (*L2-gets* (λs. *0*) [$\mathcal{S}$ ''*ret*'']) (λr. *XXX21 r x*)) *names*
⟨*proof*⟩

lemma *PROP SPLIT* (⋀r. ((λ(x,y,z). *y < z ∧ z=s*) *r*) ⟹ *P r*)
≡ (⋀*x y z*. *y < z ∧ z = s* ⟹ *P* (*x*, *y*, *s*))
⟨*proof*⟩

experiment
begin

schematic-goal *foo*:
⋀*x1 x2 x3 s*. (*case* (*x1*, *x2*, *x3*) *of* (*x*, *y*, *z*) ⇒ λs. *0 < y*) *s* ⟹
(*case* (*x1*, *x2*, *x3*) *of* (*x*, *y*, *z*) ⇒ *L2-seq* (*L2-guard* (λ-. *Suc 0 ≤ y ∧ z = a*)) *X*) = *?A′* (*x1*, *x2*, *x3*) *s*
⟨*proof*⟩

?A′ should be properly instantiated.

thm *foo*

end

declare [[*simp-trace=true*, *simp-trace-depth-limit=100*]]
lemma *L2-while* (λ(x,y,z) *s. 0 < (y::nat)*) (λ(x,y,z). *L2-seq* (*L2-guard* (λ-. *Suc 0 ≤ y ∧ z = a*)) *X*) *x ns*
=
L2-while (λ(x,y,z) *s. 0 < y*) (λ(x,y,z). *L2-seq* (*L2-guard* (λ-. *z = a*)) *X*) *x ns*
⟨*proof*⟩

Finally the following setup implements a very controlled tuple esplitting on the While body.

lemma
assumes *c-eq*: $c = c'$
assumes *bdy-eq*: *PROP SPLIT* $(\bigwedge r\ s.\ c'\ r\ s \Longrightarrow run\ (A\ r)\ s = run\ (A'\ r)\ s)$
shows *L2-while* $c\ A =$ *L2-while* $c'\ A'$
⟨*proof*⟩

With *SPLIT* we mark a position in the term that will trigger `Tuple_Tools.SPLIT_simproc`. By adding *SPLIT PROP ?P* ≡ *SPLIT PROP ?P* as congruence rule we prohibit the simplifier from diving into the loop body before we actually split the result variable. Note that the simplifier usually works bottom up, only congruence rules are applied top down.

lemma *L2-while* (λ(*x*,*y*,(*z*::*nat*)) *s*. *0* < (*y*::*nat*) ∧ *y*=*x*) (λ(*x*,*y*,*z*). *L2-seq* (*L2-guard* (λ-. *Suc 0* ≤ *y* ∧ *z* = *a*)) *X*) *x ns*
=
L2-while (λ(*x*, *y*, *z*) *s*. *0* < *y* ∧ *y* = *x*) (λ(*x*, *x*, *z*). *L2-seq* (*L2-guard* (λ-. *z* = *a*)) *X*) *x ns*
⟨*proof*⟩

26.8 Simplification of conditions (guards, loops, conditionals)

The basic peephole optimization for conditions tries to simplify "trivial" guards / conditions by propagating the information of guards / conditions to subsequent guards / conditions. It is called "peephole" optimisation (in contrast to "flow"-sensitive), because it only propagates the state independent information of gaurds, i.e. constraints on constant expressions or local variables, not constraints on the state itself.

To facilitate this propagation with congruence rules and simprocs, we introduce two special constants *L2-seq-gets* and *L2-seq-guard* that are introduced as intermediate step by a conversion. With this marking we can distinguish the cases by congruence rules, without the marking both cases would be subject to a single congruence rule for *L2-seq*. (Note that the simplifier only considers the head term of a congruence rules.)

We add the congruence rules *?c* = *?c'* ⟹ *L2-seq-gets ?c ?n ?A* ≡ *L2-seq-gets ?c' ?n ?A* and ⟦*?P* = *?P'*; ⋀*s*. *?P' s* ⟹ *run* (*?X* ()) *s* = *run ?X' s*⟧ ⟹ *L2-seq-guard ?P ?X* = *L2-seq-guard ?P'* (λ-. *?X'*). The congruence rules for the guard is a standard congruence rules that adds the gaurd as a precondition for simplifying the second statement in the sequential composition. The congruence rule for *?c* = *?c'* ⟹ *L2-seq-gets ?c ?n ?A* ≡ *L2-seq-gets ?c' ?n ?A* stops the simplifier from descending into the second term of the sequential composition. Instead the simproc `L2Opt.l2_marked_gets_bind_simproc` is triggered to analyse the situation. It does the following:

- If the returned value in the first statement is simple, only occurs once in

second statement, or is a structure-constructor or update that is only appied to structure-selections then the value is directly propagated to the second statement and the sequential composition is removed.

- It peeks into the prems of the simplifier to see if there are already "interesting" facts collected from the congruence rules of guards / conditions. If not, it unfolds the marking and continues with the ordinary sequential composition by simplifying the second statement. "Interesting" means there is at least one premise that has impact on the new return value. E.g. if we have $x + y < (5::'a)$ in the prems, and we now we return $x + y$, than the simproc introduces a new variable r for return value, augments the context with the equation $r = x + y$ and derives the new premise $r < (5::'a)$ which is put to the premises of the simplifier as well as the simpset. Then the simplifier is called recursively on the second statement of the sequential composition with the augmented context. When it is finished it uses the rule *?f ?c ≡ ?g ⟹ L2-seq-gets ?c ?n ?f ≡ STOP* (*L2-seq-gets ?c ?n* (λ-. *?g*)) to introduce the constant *STOP*. The congruence rule *STOP ?P ≡ STOP ?P* prohibits the simplifier to descend down into the just simplified second statement again.

  Note that this setup works around the limitation of the simplifier that does not allow us to modify the context by something like a "congproc", similar to a "simproc". Keep in mind that a congruence rule is applied top-down as the simplifier works its way down into a term, whereas a simpproc (or any other simplification rule) is applied buttom up. So a simproc / simplification rule can build on the fact that the subterms are already normalised. The simplifier makes use of this by doing some bookkeeping with "term-skeletons" to avoid resimplification of subterms. This mechanism fails short in our setup of using a congruence rule to stop simplification of the second statement and calling the simproc instead. That is why we explicitly introduce *STOP*. It will be removed after simplification by an additional traversal of the term.

declare [[*simp-trace=false*, *ML-print-depth=1000*]]

Propagation of simple constant by unfolding.

lemma *L2-seq-gets c* [$\mathcal{S}$ $''r''$] (λ*r*. (*L2-guard* (λ-. *P r*))) =
L2-guard (λ-. *P c*)
⟨*proof*⟩

Propagation of term, as it only appears once in the second statement.

lemma *L2-seq-gets* (*c + d*) [$\mathcal{S}$ $''r''$] (λ*r*. (*L2-guard* (λ-. *P r*))) =
L2-guard (λ-. *P* (*c + d*))
⟨*proof*⟩

As nothing is in the prems yet, marking is just removed and second statement is thus simplified

lemma *L2-seq-gets* $(c + d)$ $[\mathcal{S}\ ''r'']$ $(\lambda r.\ (L2\text{-}guard\ (\lambda\text{-}.\ P\ r \wedge (P\ r \longrightarrow Q\ r))))$
$=$
$STOP\ (L2\text{-}seq\text{-}gets\ (c + d)\ [\mathcal{S}\ ''r'']\ (\lambda r.\ L2\text{-}guard\ (\lambda\text{-}.\ P\ r \wedge Q\ r)))$
⟨*proof*⟩

The first guard condition is propagated to the second guard, via the intermediate assignment $r = a + b$.

lemma *L2-seq-guard* $(\lambda\text{-}.\ a + b < 5)$
$(\lambda\text{-}.\ L2\text{-}seq\text{-}gets\ (a + b)\ [\mathcal{S}\ ''r'']$
$(\lambda r.\ L2\text{-}guard\ (\lambda\text{-}.\ r < 5 \wedge P))) =$
$L2\text{-}seq\text{-}guard\ (\lambda\text{-}.\ a + b < 5)$
$(\lambda\text{-}.\ L2\text{-}guard\ (\lambda\text{-}.\ P))$
⟨*proof*⟩

⟨*ML*⟩

lemma *L2-seq-guard* $(\lambda s.\ V1 + a - (r::int) \leq 2)$
$(\lambda\text{-}.\ L2\text{-}seq\ (L2\text{-}condition\ (\lambda s.\ CC)\ (L2\text{-}seq\text{-}guard\ (\lambda s.\ V1 + a - r \leq 3)$
$(\lambda\text{-}.\ X1))\ X2)\ (\lambda\text{-}.\ X3)) =$
$L2\text{-}seq\text{-}guard\ (\lambda s.\ V1 + a - r \leq 2)$
$(\lambda\text{-}.\ L2\text{-}seq\ (L2\text{-}condition\ (\lambda s.\ CC)\ (L2\text{-}seq\text{-}guard\ (\lambda s.\ True)\ (\lambda\text{-}.\ X1))\ X2)$
$(\lambda\text{-}.\ X3))$
⟨*proof*⟩

⟨*ML*⟩

declare [[*simp-trace=false*, *linarith-trace=false*]]

26.9 Tricks to enforce first-order unification

⟨*ML*⟩

In heap lifting phase:

thm *heap-abs-fo*

In word abstraction phase:

thm *abstract-val-fun-app*

lemma *abstract-val-fun-app'*:
$[\![abstract\text{-}val\ Q\ x\ id\ x';\ abstract\text{-}val\ P\ f\ id\ f'\ ]\!] \Longrightarrow$
$abstract\text{-}val\ (P \wedge Q)\ (g\ (f\ x))\ g\ (f'\ x')$
⟨*proof*⟩

schematic-goal *abstract-val ?Q ?f g* (*f′ a′*)
⟨*proof*⟩

`Utils.fo_arg_resolve_tac` does the trick for us.

schematic-goal *abstract-val ?Q ?f g* (*f′ a′*)
⟨*proof*⟩

It turned out that `Utils.fo_arg_resolve_tac` and other tactics like simplification can become quite slow when operating under a lot of bound variables. We introduced a `WordAbstract.thin_abs_var_tac` to remove unused premises and bound variables. It is triggered by *THIN* in the rules, e.g. ⟦*introduce-typ-abs-fn ?rx1.0*; *THIN* (*corresTA ?P ?rx1.0 ?ex ?L ?L′*); *THIN* (⋀*r r′*. *abs-var r ?rx1.0 r′* ⟹ *corresTA* (*?Q r*) *?rx2.0 ?ex* (*?R r*) (*?R′ r′*))⟧ ⟹ *corresTA ?P ?rx2.0 ?ex* (*L2-seq ?L* (λ*r*. *L2-seq* (*L2-guard* (*?Q r*)) (λ-. *?R r*))) (*L2-seq ?L′ ?R′*).

schematic-goal
⋀*n′ s r r′ ra r′a sa*.

abs-var r id r′ ⟹
abs-var ra id r′a ⟹ *abstract-val* (*?Q57 r ra sa*) (*?b60 r ra sa*) *id* (*6* ∗ *r′a*)

⟨*proof*⟩

schematic-goal
⋀*n′ s r r′ ra r′a sa*.
abs-var n id n′ ⟹
abs-var r id r′ ⟹
abs-var ra id r′a ⟹ *abstract-val* (*?Q57 r ra sa*) (*?b60 r ra sa*) *id* (*6* ∗ *r′a*)

⟨*proof*⟩

26.10 Exception Rewriting

This section highlights some of the ideas to rewrite and optimise exceptions.

In SIMPL and L1 the cause for an exception is stored in the state record in field *global-exn-var′-′*. The handler then inspects the state to decide whether it is responsible or not, and either handles the exception or re-raises it. In L2 this is refined in several steps.

- The cause of the exception is moved out of the state and represented as an error result in the L2 monad.
- The check for the cause in the handler of *L2-catch* is resolved to a static nesting of *L2-try* instead. The static nesting depth is directly

reflected in the depth of the sum that reflects the error value *Inl* (*Inl* ...). For local exceptions this sequence of *Inl* ends with an *Inr*, global exception (crossing function boundaries) end in an *Inl*.

- The tuple arity of intermediate bindings in statements like *L2-seq*, *L2-while* is optimised to only propagate values that are actually used later on. As a side effect of this step, unnecessary nondeterministic initialisation steps for local variables (especially the technical return variable) are removed. The initialisation becomes unnecessary if the variable is strictly assigned to before used. This is always the case for the return variable, as *return x* is translated to an assignment to the return variable followed by raising the *Return* exception.

  Note that the field *global-exn-var'-'* has a special status, it is neither part of *globals* nor *locals*. In function calls it is treated like a global variable to ensure that the exception passes function boundaries. However, in the local variable abstraction of phase L2 it is treated similar to local variable and abstracted to lambda bindings.

declare [[*verbose=3*]]
declare [[*verbose-timing = 3*]]

26.10.1 Preliminary examples illustrating the usage of *rel-spec-monad*

schematic-goal *rel-spec-monad* (=) (=)
(*L2-catch*
(*L2-seq*
(*L2-catch*

(*L2-while* (λ*r s*. *True*)
(λ*r*. *L2-condition* (λ*s*. *3* < *a*) (*L2-throw* (*Return*, (*1*::*int*)) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'']) (*L2-seq*
(*L2-condition* (λ*s*. *1* < *a*)
((*L2-throw* (*Nonlocal* (*1*::*nat*), *r*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'']))::(*nat c-exntype* × *int*, *int*, '*a*) *exn-monad*)
(*L2-throw* (*Break*, *r*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'']))
(λ*ra*. *L2-gets* (λ*s*. *r*) [$\mathcal{S}$ ''*ret*'']))))
r [$\mathcal{S}$ ''*ret*'']))

(λ(*a*, *b*).
L2-condition (λ*s*. *a* = *Break*) (*L2-gets* (λ-. *b*) [$\mathcal{S}$ ''*ret*'']) (*L2-throw* (*a*, *b*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'']))))
(λ*r*. *L2-throw* (*Return*, *2*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'']))
(λ(*a*, *b*).
L2-condition (λ*s*. *is-local a*) (*L2-gets* (λ*s*. *b*) [$\mathcal{S}$ ''*ret*'']) (*L2-throw* (*the-Nonlocal a*) [$\mathcal{S}$ ''*global-exn-var*'']))))

?A

⟨*proof*⟩

schematic-goal *rel-spec-monad* (=) (=)
(*L2-seq* (*L2-unknown* [$\mathcal{S}$ ''*ret--int*''])
(λ*r*. *L2-catch*
(*L2-seq*
(*L2-catch*
(*L2-seq*
(*L2-while* (λ*r s*. *True*)
(λ*r*. *L2-condition* (λ*s*. *3* < *a*) (*L2-throw* (*Return*, *1*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'']) (*L2-seq*
(*L2-condition* (λ*s*. *1* < *a*)
(*L2-throw* (*Nonlocal 1*, *r*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*''])
(*L2-throw* (*Break*, *r*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'']))
(λ*ra*. *L2-gets* (λ*s*. *r*) [$\mathcal{S}$ ''*ret*'']))))
r [$\mathcal{S}$ ''*ret*'']))
(λ*r*. *L2-gets* (λ*s*. ()) [$\mathcal{S}$ ''*ret*'']))
(λ(*a*, *b*).
L2-condition (λ*s*. *a* = *Break*) (*L2-gets* (λ-. ()) [$\mathcal{S}$ ''*ret*'']) (*L2-throw* (*a*, *b*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'']))))
(λ*r*. *L2-throw* (*Return*, *2*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'']))
(λ(*a*, *b*).
L2-condition (λ*s*. *is-local a*) (*L2-gets* (λ*s*. *b*) [$\mathcal{S}$ ''*ret*'']) (*L2-throw a* [$\mathcal{S}$ ''*global-exn-var*''])))))

?A

⟨*proof*⟩

26.10.2 From *L2-catch* and flat exceptions to *L2-try* and nested exceptions

schematic-goal *rel-spec-monad* (=) (=)
(*L2-seq* (*L2-unknown* [$\mathcal{S}$ ''*ret--int*''])
(λ*r*. *L2-catch*
(*L2-seq*
(*L2-catch*
(*L2-seq*
(*L2-while* (λ*r s*. *True*)
(λ*r*. *L2-condition* (λ*s*. *3* < *a*) (*L2-throw* (*Return*, *1*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'']) (*L2-seq*
(*L2-condition* (λ*s*. *1* < *a*)
(*L2-throw* (*Nonlocal 1*, *r*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*''])
(*L2-throw* (*Break*, *r*) [$\mathcal{S}$ ''*global-exn-var*'', $\mathcal{S}$ ''*ret*'']))

```
                    (λra. L2-gets (λs. r) [S ''ret'']))
              r [S ''ret''])
            (λr. L2-gets (λs. ()) [S ''ret'']))
          (λ(a, b).
             L2-condition (λs. a = Break) (L2-gets (λ-. ()) [S ''ret''])
              (L2-throw (a, b) [S ''global-exn-var'', S ''ret'']))))
        (λr. L2-throw (Return, 2) [S ''global-exn-var'', S ''ret'']))
      (λ(a, b).
         L2-condition (λs. is-local a) (L2-gets (λs. b) [S ''ret''])
          (L2-throw a [S ''global-exn-var''])))))

?A

  ⟨proof⟩


schematic-goal rel-spec-monad (=) (=)
(L2-seq (L2-unknown [S ''ret--int''])
 (λr. L2-catch
       (L2-seq
         (L2-catch
           (L2-seq
             (L2-while (λr s. True)
             (λr. L2-condition (λs. 3 < a) (L2-throw (Return, 1) [S ''global-exn-var'',
S ''ret''])
                       (L2-seq
                         (L2-condition (λs. 1 < a)
                           (L2-throw (Nonlocal 1, r) [S ''global-exn-var'', S ''ret''])
                           (L2-throw (Break, r) [S ''global-exn-var'', S ''ret'']))
                         (λra. L2-gets (λs. r) [S ''ret'']))))
              r [S ''ret''])
            (λr. L2-gets (λs. ()) [S ''ret'']))
          (λ(a, b).
             L2-condition (λs. a = Break) (L2-gets (λ-. ()) [S ''ret''])
              (L2-throw (a, b) [S ''global-exn-var'', S ''ret'']))))
        (λr. L2-throw (Return, 2) [S ''global-exn-var'', S ''ret'']))
      (λ(a, b).
         L2-condition (λs. is-local a) (L2-gets (λs. b) [S ''ret''])
          (L2-throw a [S ''global-exn-var''])))))

?A

  ⟨proof⟩

schematic-goal rel-spec-monad (=) (=)
 (L2-seq (L2-unknown [S ''ret--int''])
                                (λret--int.
                                   L2-catch
                                    (L2-seq
```

```
                                        (L2-condition (λs. p = 0x20 ∨ 2 <s p)
                                       (L2-condition (λs. p ≠ 0x1E ∧ p ≠ 2) (L2-throw
(Return, p) [S ''global-exn-var'', S ''ret'']
                                            (L2-gets (λ-. ret--int) [S ''ret'']))
                                         (L2-gets (λ-. ret--int) [S ''ret'']))
                                        (λret--int.
                                           L2-seq
                                       (L2-call (l2-add' undefined p p) (λglobal-exn-var.
(Nonlocal (the-Nonlocal global-exn-var), ret--int))
                                             [S ''global-exn-var'', S ''ret''])
                                                 (λp---int. L2-throw (Return, p) [S
''global-exn-var'', S ''ret'']))))
                                      (λ(global-exn-var, ret--int).
                                      L2-condition (λs. is-local global-exn-var) (L2-gets
(λ-. ret--int) [S ''ret''])
                                       (L2-throw global-exn-var [S ''global-exn-var'']))))
?A

  ⟨proof⟩

⟨ML⟩

schematic-goal rel-spec-monad (=) (rel-xval (=) (=))
(L2-seq (L2-unknown [S ''ret--int''])
 (λr. L2-catch
       (L2-seq
         (L2-catch
           (L2-seq
             (L2-while (λr s. True)
           (λr. L2-condition (λs. 3 < a) (L2-throw (Return, 1) [S ''global-exn-var'',
S ''ret''])
                    (L2-condition (λs. 1 < a) (L2-gets (λs. r) [S ''ret''])
                      (L2-throw (Break, r) [S ''global-exn-var'', S ''ret'']))))
               r [S ''ret''])
             (λr. L2-gets (λs. ()) [S ''ret'']))
           (λ(a, b).
              L2-condition (λs. a = Break) (L2-gets (λ-. ()) [S ''ret''])
               (L2-throw (a, b) [S ''global-exn-var'', S ''ret'']))))
         (λr. L2-throw (Return, 2) [S ''global-exn-var'', S ''ret'']))
       (λ(a, b). L2-condition (λs. is-local a) (L2-gets (λs. b) [S ''ret'']) (L2-throw
a [S ''global-exn-var''])))))
?A
  ⟨proof⟩

schematic-goal
rel-spec-monad (=) (rel-xval (=) (=))
(L2-seq (L2-unknown [S ''ret--int''])
 (λr. L2-catch
       (L2-seq
```

```
        (L2-catch
          (L2-seq
            (L2-while (λ(a, b) a. True)
              (λ(a, b).
                 L2-seq
                  (L2-condition (λs. 3 < a)
                    (L2-throw (a, Return, 1) [S ''a'', S ''global-exn-var'', S ''ret'']) 
                    (L2-seq
                      (L2-condition (λs. 1 < a) (L2-gets (λs. a + 1) [S ''a''])
                      (L2-throw (a, Break, b) [S ''a'', S ''global-exn-var'', S ''ret'']))
                      (λr. L2-gets (λs. (r, b)) [S ''a'', S ''ret'']))))
                  (λ(a, b). L2-gets (λs. (a, b)) [S ''a'', S ''ret'']))
              (a, r) [S ''a'', S ''ret''])
            (λ(a, b). L2-gets (λs. a) [S ''a'']))
          (λ(aa, a, b).
             L2-condition (λs. a = Break) (L2-gets (λs. aa) [S ''a''])
              (L2-throw (a, b) [S ''global-exn-var'', S ''ret'']))) 
        (λr. L2-throw (Return, r) [S ''global-exn-var'', S ''ret'']))
      (λ(a, b).
         L2-condition (λs. is-local a) (L2-gets (λs. b) [S ''ret''])
          (L2-throw a [S ''global-exn-var''])))) 
?A
  ⟨proof⟩
```

declare [[*show-main-goal*=*true*]]

26.10.3 Flatten the error type of calls, aka get rid of constructor *Nonlocal*

```
schematic-goal rel-spec-monad (=) (rel-xval rel-Nonlocal (=))
(L2-seq (L2-unknown [S ''ret--int''])
 (λr. L2-catch
       (L2-seq
         (L2-catch
           (L2-seq
             (L2-while (λr s. True)
            (λr. L2-condition (λs. 3 < a) (L2-throw (Return, 1) [S ''global-exn-var'',
S ''ret'']) 
                    (L2-seq
                      (L2-condition (λs. 1 < a)
                        (L2-throw (Nonlocal 1, r) [S ''global-exn-var'', S ''ret''])
                        (L2-throw (Break, r) [S ''global-exn-var'', S ''ret'']))
                      (λra. L2-gets (λs. r) [S ''ret'']))) 
              r [S ''ret''])
            (λr. L2-gets (λs. ()) [S ''ret'']))
          (λ(a, b).
             L2-condition (λs. a = Break) (L2-gets (λ-. ()) [S ''ret''])
              (L2-throw (a, b) [S ''global-exn-var'', S ''ret''])))
```

(λr. L2-throw (Return, 2) [𝒮 ''global-exn-var'', 𝒮 ''ret'']))
(λ(a, b).
L2-condition (λs. is-local a) (L2-gets (λs. b) [𝒮 ''ret''])
(L2-throw a [𝒮 ''global-exn-var'']))))

?A

⟨proof⟩

Some statistics on the caches involved.

⟨ML⟩

26.11 Tuple optimization by analysing variable use and removing unused variables.

The cases for *L2-seq* are the points were the uses-analysis is invoked: `Rel_Spec_Monad_Synthesize_Ru`
The analysis follows the usage of the bound variable within the term.

schematic-goal
rel-spec-monad (=) (rel-xval (=) (rel-project (λv. ())))
(L2-seq (L2-gets (λ-. x) [𝒮 ''x''])
(λx. L2-seq
(L2-gets (λ-. (x, y)) [𝒮 ''x'',𝒮 ''y''])
(λ(x, y). L2-gets (λ-. y) [𝒮 ''y'']))) ?X

⟨proof⟩

schematic-goal
rel-spec-monad (=) (rel-xval (=) (rel-project (λv. v)))
(L2-seq (L2-gets (λ-. x) [𝒮 ''x''])
(λx. L2-seq
(L2-gets (λ-. (x, y)) [𝒮 ''x'',𝒮 ''y''])
(λ(x, y). L2-gets (λ-. y) [𝒮 ''y'']))) ?X

⟨proof⟩

schematic-goal
rel-spec-monad (=) (rel-xval (=) (rel-project (λv. v)))
(L2-seq (L2-gets (λ-. (l,m)) [𝒮 ''l'', 𝒮 ''m''])
(λ(x, y). L2-seq
(L2-gets (λ-. (x, k)) [𝒮 ''x'', 𝒮 ''y''])

(λ(a, b).
L2-seq
(L2-while (λ(x,y) -. x < 2)

```
          (λ(x,y). (L2-gets (λ-. (y, k)) [S ''x'', S ''y'']) )
          (a, b)  [S ''x'', S ''y''])
        (λ(x, y). L2-gets (λ-. y) [S ''y'']) )))
?X
 ⟨proof⟩
```

```
declare [[show-main-goal=true]]
schematic-goal
 rel-spec-monad (=) (rel-xval(=) (rel-project (λv. v)))
   (L2-seq (L2-gets (λ-. (l,m)) [S ''l'', S ''m''])
     (λ(x, y). L2-seq
       (L2-gets (λ-. (x, k)) [S ''x'', S ''y''])

       (λ(a, b).
        L2-seq
         (L2-while (λ(x,y) -. y < 2)
           (λ(x,y). (L2-gets (λ-. (y, k)) [S ''x'', S ''y'']) )
           (a, b)  [S ''x'', S ''y''])
         (λ(x, y). L2-gets (λ-. y) [S ''y'']) )))
?X
 ⟨proof⟩
```

```
schematic-goal
 rel-spec-monad (=) (rel-xval (=) (rel-project (λv. v)))
(L2-seq (L2-unknown [S ''ret--unsigned''])
 (λx. L2-try
       (L2-seq
         (L2-while (λ(n---int, ret--unsigned) s. True)
           (λ(x1, x2).
                 L2-seq (L2-guard (λs. 0 ≤ 2147483649 + sint x1 ∧ sint x1 ≤
2147483646))
               (λxa. L2-seq (L2-gets (λ-. x1 + 1) [S ''n''])
                     (λxb. L2-seq
                           (L2-condition (λs. xb <s 2) (L2-throw (Inr 1) [S ''ret''])
                              (L2-throw (Inr 2) [S ''ret'']))
                             (λxc. L2-gets (λ-. (xb, xc)) [S ''n'', S ''ret'']))))
           (n, x) [S ''n'', S ''ret''])
         (λ(x1, x2). L2-fail))))
?X

 ⟨proof⟩

schematic-goal rel-spec-monad (=) (rel-xval (=) (rel-project (λv. v)))
 (L2-seq (L2-call (l2-plus' undefined 1 2) (λe. e) [])
   (λx. L2-seq (L2-call (l2-plus' undefined 2 3) (λe. e) [])
```

```
        (λxa. L2-gets (λs. if xa = 0 then 1 else 0) [S ''ret'']))))
?XX
  ⟨proof⟩


schematic-goal rel-spec-monad (=) (rel-xval (=) (rel-project (λv. v)))
(L2-seq (L2-unknown [S ''ret--int''])
  (λx. L2-seq (L2-gets (λ-. 2) [S ''x''])
        (λxa. L2-try
              (L2-seq
                (L2-try
                  (L2-seq
                    (L2-while (λ(a---int, ret--int, y---int) s. True)
                      (λ(x1, x2, x3).
                          L2-seq (L2-gets (λ-. 4) [S ''y''])
                           (λxb. L2-seq
                                 (L2-condition (λs. 3 <s x1) (L2-throw (Inl (Inr 1))
[S ''ret'', S ''y''])
                                    (L2-seq
                                      (L2-condition (λs. 1 <s x1)
                                       (L2-seq (L2-guard (λs. 0 ≤ 2147483649 + sint
x1 ∧ sint x1 ≤ 2147483646))
                                           (λxc. L2-gets (λs. x1 + 1) [S ''a'']))
                                       (L2-try
                                         (L2-seq (L2-try (L2-while (λ- s. True) (λxc.
L2-throw (Inr xc) []) () [])) (λxc. L2-throw (Inl (Inr xb)) [])))))
                                      (λxc. L2-gets (λ-. (xc, x2)) [S ''a'', S ''ret'']))))
                                     (λ(x1, x2). L2-gets (λ-. (x1, x2, xb)) [S ''a'', S
''ret'', S ''y'']))))
                    (a, x, 3) [S ''a'', S ''ret'', S ''y'']
                   (λ(x1, x2, x3). L2-gets (λ-. x3) [S ''y'']))))
                (λxb. L2-seq (L2-guard (λs. − 2147483648 ≤ sint xa + sint xb ∧
sint xa + sint xb ≤ 2147483647))
                     (λxc. L2-throw (Inr (xa + xb)) [S ''ret'']))))))
?X
  ⟨proof⟩


schematic-goal rel-spec-monad (=) (rel-xval (=) (rel-project (λv. v)))
  (L2-condition (λs. n = 0) (L2-gets (λs. 0) [S ''ret''])
    (L2-seq (L2-call (l2-even' (recguard-dec rec-measure') (n − 1)) (λe. e) [])
      (λx. L2-gets (λs. SCAST(32 signed → 32) (if x = 0 then 1 else 0))
            [S ''ret'']))))
?X
  ⟨proof⟩



schematic-goal rel-spec-monad (=) (rel-xval (=) (rel-project (λv. v)))
```

```
(L2-seq (L2-unknown [S ''ret--unsigned'']) 
  (λx. L2-condition (λs. n = 0) (L2-gets (λs. 0) [S ''ret'']) 
        (L2-try 
          (L2-seq 
            (L2-call 
              (l2-fac-exit' (recguard-dec rec-measure') (n − 1)) 
               (λe. Inl (Nonlocal (the-Nonlocal e))) [S ''ret'']) 
            (λxa. L2-throw (Inr xa) [S ''ret'']))))))
?X
 ⟨proof⟩
```

```
schematic-goal rel-spec-monad (=) (rel-xval (=) (rel-project (λv. v)))
(L2-try
 (L2-call (l2-just-exit' undefined)
  (λe. Inl (Nonlocal (the-Nonlocal e))) [])
  )
?X
 ⟨proof⟩
```

```
declare [[show-main-goal=false]]
declare [[verbose=0]]
declare [[verbose-timing=0]]
```

26.12 Locales, Local-Theories, Named-Targets, Morphisms and Declarations...

We explore some of the concepts of Locales, Local-Theories, Named-Targets, Declarations etc. Besided the corresponding chapters in the Isabelle/Isar Reference Manual as well as the Isabelle/Isar Implementation Manual, the following references give a solid theoretical background:

- Local Theories: http://isabelle.in.tum.de/~haftmann/pdf/local_theory_specifications_haftmann_wenzel.pdf
- Locales: http://www21.in.tum.de/~ballarin/publications/jar2013.pdf
- Morphisms and Declarations: https://www21.in.tum.de/~wenzelm/papers/context-methods.pdf

  Roughly speaking, locales provide the infrastructure to manage the dependency graph of locale declarations. When entering a locale this dependency graph is traversed and results in a blue-print to build a local context by expanding and evaluating the locale expressions in a canonical order. The result is presented to the user as a local theory. A local theory consists of an axiomatic part (think of fixes and assumes)

and a derived part, consisting of definitions and theorems within the axiomatic part.

locale *Foo* =
fixes *N*::*nat*
fixes *M*::*nat*
assumes *N-le-M*: $N < M$

Locale *Foo* is blue-print to build a local context. We can enter this context with **context**.

context *Foo*
begin

Within the context we can explore it. As an example the locale assumption becomes a theorem within the context

thm *N-le-M*

Within the context the axiomatic part (fixes and assumes) are often accessed implicitly. For example the fixes are presented as free Variables with the declared type fixed.

⟨*ML*⟩

The locale assumptions are wrapped up in the locale predicate *Foo*, and internally become part of the hypothesis of a (local theory) theorem. Theorems derived within the local theory will all carry around this implicit hypothesis.

declare [[*show-hyps*]]
thm *N-le-M*
⟨*ML*⟩
thm *Foo-def*

There is also an exported version of the theorem that holds on the theory level. Here the implicit hypothesis becomes an explicit precondition.

thm *Foo.N-le-M*

A definition in a local theory has multiple effects. A generalised definition, where the locale fixes become explicit parameters is issued in the background theory. Moreover, an abbreviation that hides these parameters in introduced.

definition *G-def*: $G = M + N$

lemma $G = Foo.G\ N\ M$⟨*proof*⟩

⟨*ML*⟩

When working within ML we also deal with another view, the *auxiliary context* as referred to in the Isabelle documentation. On the Isar top-level

we are usually only exposed to the *background theory*, e.g. *Foo.G* and the *target context*, e.g. *G*.

⟨*ML*⟩

Here we have the target view of the freshly defined *F*.

⟨*ML*⟩

One notable logical difference of the view on a definition within a auxiliary context and the target context is polymorphism. Within the auxiliary context the definition is always monomorphic, maybe referring to implicitly fixed types 'variables', or better frees. Within the target context it might be polymorphic (depending on the definition), as it is just an abbreviation for the global constant applied to the locale parameters.

The fixed view of the auxiliary context can be quite handy when continuing work with the freshly defined function, do some proofs etc. in ML.

Working with nested contexts, like fixing and assuming new stuff opening and closing them and exporting results is supported within ML by `Local_Theory.begin_nested` and `Local_Theory.end_nested`. The term might be a little bit misleading as you don't actually open an close a target but open and close a context block within the same target.

The following example illustrates this first with toplevel commands and then with ML.

context
fixes *K*
assumes *M-le-K*: $M < K$
begin
lemma *N-le-K*: $N < K$
⟨*proof*⟩

Here we have the local view on the theorem within the nested context.

thm *N-le-K*
end

Now we have the exported view of the theorem within the target context.

thm *N-le-K*

⟨*ML*⟩

Here we have the exported view of the theorem within the target context.

thm *N-le-K'*

end

After a definition in a local theory the context is extended with a fixed variable for the lhs and an equation that relates it to the global definition. So everything what happens thereafter happens within that extended context.

⟨*ML*⟩

By surrounding the definition with a `Local_Theory.begin_nested` and `Local_Theory.end_nested` you can step out of the extended context again. Don't forget to export the results as well.

⟨*ML*⟩

26.13 Morphisms and Declarations

As we have seen before, when working with locales and local theories, we deal with different views on the same results (like theorems). The locale infrastructure takes care to provide the results in the 'expected' way by a careful naming policy for facts and (local) constants.

When maintaining results in custom (generic) context data, Isabelle provides mechanisms around the notion of a *declaration*, which comes in various flavours. In particular attached to (local) theorems / facts in the form of attributes (rule and declaration attributes) or as local theory declarations. In the case of attributes the morphism is implicit, as the theorem / facts they are attached to are already in the expected localized version. So the function can refer to these theorem / facts. In case of local theory declarations the morphism is explicitly passed as an argument to the function.

In ML, attributes are functions from `thm * Proof.context -> thm * Proof.context`, in practice presented as either a declaration attribute (modifying the context) `thm -> Proof.context -> Proof.context` or as a rule attribute (modifying the theorem) `Proof.context -> thm -> thm`. Local theory declarations are functions, taking a morphism and modifying the `local_theory`: `Morphism.morphism -> local_theory -> local_theory`.

All these declarations are part of the context-building-infrastructure of locales and local theories. In that sense a local theory declaration can be thought of as declaration attribute attached to a dummy fact. Whenever a context is entered and the fact is processed the declarations are reevaluated on the that context. As we have seen, at each point there are three contexts simultaneously, representing the background theory, the target context and the auxiliary context. So at each point there are also three views on the data, 'viewed' via the application of the explicit or implicit morphisms.

Note that when implementing context data and tools, morphisms come in two variants. Import and export morphisms. Exporting results or data out of a context into a more general context can often be treated explicit by `Proof_Context.export` or `Proof_Context.export_morphism` or variants of those functions. This is because going out of a context is unambigous and therefor a single morphism is enough.

Whereas when entering into a context (like entering a locale) the morphisms are internally generated and provided by the locale infrastructure. As a locale might be imported several times in different variants into the same local theory there is no unambigous morphism. Each import provides

a different morphism.

When issuing a local theory declaration via `Local_Theory.declaration` there is also two flags to define, represented as a record `{syntax: bool, pervasive: bool}`.

- A pervasive declaration is also applied to the background-theory (besides the target and auxiliary context)
- A syntax declaration is activated already in the syntax phase of the locale roundup. This means, when your context data already has to be ready to enabling parsing of new assumptions declaring a new locale you have to set the syntax flag.

  Note that theorem attributes are treated like `{pervasive = false, syntax = false}`. That means that they are not evaluated on the background-theory version of the facts. This makes perfect sense for typical attributes like *simp*. The theorem is only added to the simpset within the locale. The background-theorem is not added to the simpset of the theory. Keep in mind that the background-theorem is typically an implication with the locale predicate as precondition. However, when you interprete the locale on the theory level the interpreted version of the theorem is added to the simpset of the theory, here the locale-predicate precondition is discharged.

  In the following we have some examples illustrating the described behaviour.

⟨*ML*⟩

context *Foo*
begin
⟨*ML*⟩

Manipulating data by functions like `Context.proof_map` will only have very limited and especially temporary effect. It is applied to the auxiliary context only. When exiting the auxiliary context it will be away. Also it has no effect on the target context.

⟨*ML*⟩

You might be surprised to see that the value is already reset here. The reason is that every toplevel command resets the context and is treated like a block. Think of a `Local_Theory.begin_nested` / `Local_Theory.end_nested`.

⟨*ML*⟩

A non pervasive declaration takes effect on the auxiliary and the target context but not the theory context.

⟨*ML*⟩

As expected the effect is still there at this point.

⟨*ML*⟩
end

Note that the declaration is persistent in the sense that when reentering the locale it will be evaluated again. So when designing custom data and implementing declarations or attributes one has to be clear about the order of things and that they might be applied in many different situations. Using `MyData.map` instead of `MyData.put` might be handy to design robust data management.

context *Foo*
begin
⟨*ML*⟩

A pervasive declaration also effects the theory.

⟨*ML*⟩

end

⟨*ML*⟩

26.13.1 Excurse on Proof Context vs. Local Theory.

A local theory is represented as a proof context. So a local theory is a semantic subtype of a proof context. Every local theory is a proof context but not every proof context is a local theory. In particular a local theory is a proof context that is ready to accept (local theory) definitions (`Local_Theory.define`) and notes (`Local_Theory.note`).

The Isabelle sources try to be consistent in the naming, i.e. 'ctxt' vs 'lthy' in parameters and variables and `Proof.context` vs. `local_theory` in signatures. But as `local_theory` is only a type synonym for `Proof.context`, there is no type check for this convention. The nesting level that you get with `Local_Theory.level` is an indicator. A Level of *0* is not a proper local theory.

With `Proof_Context.init_global` you get a proof context not a local theory.

⟨*ML*⟩

With `Named_Target.theory_init` you get a local theory.

⟨*ML*⟩

With `Locale.init` you get a proof context not a local theory.

⟨*ML*⟩

With `Named_Target.init` you get a local theory.

⟨*ML*⟩

26.13.2 Attributes vs. Local Theory Declarations

Attributes on local facts, are not applied on the underlying theory level foundation. Only when you interpret a locale on the theory level the attributes get activated. This corresponds to general `Local_Theory.declaration` where you have the flag *pervasive=false*.

To demonstrate this we define an artificial tracing attribute.

⟨*ML*⟩

First we try the attribute on a theory level theorem. When you put the cursor on or after the *simp* you see the tracing output of our attribute.

lemma *foo*[*trace-attr*]: *x* = *x*
⟨*proof*⟩

Interestingly we see four tracing outputs. Here the outputs and my interpretation:

- *trace-attr* (*local-theory*, *1*, *true*, *NONE*, *NONE*): *?x* = *?x* This seems to be the immediate auxiliary context of the lemma.
- *trace-attr* (*theory*, *0*, *true*, *NONE*, *NONE*): *?x* = *?x* This is the theory level.
- *trace-attr* (*context*, *0*, *false*, *NONE*, *NONE*): *?x* = *?x* This seems to be a theory-context that might be generated with `Proof_Context.init_global`.
- *trace-attr* (*local-theory*, *1*, *true*, *NONE*, *NONE*): *?x* = *?x* This seems to be from reentering the original lemma context.

Now let us look at a nested context.

context *Foo*
begin
context
begin
lemma *silly*[*trace-attr*]: *M* = *M* ⟨*proof*⟩
end
end

As you see there is no theory level trace:

- *trace-attr* (*local-theory*, *2*, *false*, *NONE*, *SOME Scratch.Foo*): *M* = *M* [*Foo N M*] This is the original context of the lemma.
- *trace-attr* (*context*, *0*, *false*, *NONE*, *SOME Scratch.Foo*): *M* = *M* [*Foo N M*] This seems to be the target context but not as a local theory.

- *trace-attr* (*local-theory*, *1*, *false*, *SOME Scratch.Foo*, *SOME Scratch.Foo*): $M = M$ [*Foo N M*] This seems to be the target context as a local theory.
- *trace-attr* (*local-theory*, *2*, *false*, *NONE*, *SOME Scratch.Foo*): $M = M$ [*Foo N M*] This seems to be from reentering the original lemma context.

Let us now try an interpretation of the locale on the theory level.

definition *NN*::*nat* **where** *NN = 2*
definition *MM*::*nat* **where** *MM = 3*

interpretation *Foo NN MM*
⟨*proof*⟩

We can see the trace of the activation of the attribute: *trace-attr* (*theory*, *0*, *true*, *NONE*, *NONE*): *MM = MM*

26.14 ML Antiquotations

The notion of quote / antiquote is related to lisps quotation and quasi-quotation mechanisms or the notion of 'interpolation' that is used in the context of macros in other programming languages. In lisp, "everything is a list", in particular the program itself and you can dynamically evaluate a list. With quotation you can express that a list is meant to be plain data and should not be immediately evaluated. With quasi-quotation you can build data which inside uses a lisp-expression that evaluates / expands to some data that is inserted at that point.

This is a first intuition to understand ML antiquotations in Isabele/ML. An ML antiquotation is something embedded into the Isabelle/ML program text, that is not itself plain ML, but something that evaluates to a piece of ML text that is inserted in that position. A prominent example is the term antiquotation, which transforms a logical term (presented in the inner-syntax) of the logic, to the ML representation of that term. `term`

⟨*ML*⟩

The expansion takes place during compile time of the piece of Isabelle/ML. To be more precise it is expanded already during lexing of Isabelle/ML. This is important to keep in mind when writing your own antiquotations. The context you can refer to during the expansion of the antiquotation is the context you have during the lexing phase. This design decision facilitates robustness and predictability of the resulting ML code. Note that the expanded ML-code itself might of course be dynamically used later on in various contexts, but it is always the same code.

The main interface to write your own ML antiquotations is `ML_Antiquotation`. The more low-level (and more flexible) interface it is based on is `ML_Context.add_antiquotation`. The interface in `ML_Antiquotation` distinguishes the antiquotations into the variants *inline*, *value* and *special−form*.

The *inline* variant is the closest to the intuition of macro expansion. The inline-antiquotaton yields a piece of ML-text that is literally inserted at the position of the antiquotation. The term antiquotation is an example of this. `\<^term>\<open>x + y\<close>`. It is implemented with `ML_Antiquotation.inline_embedded`.
[1]

For the *value* variant the intuition is that the expanded antiquotation is immediatly evaluated in the compile time context and the resulting value is what is inserted at the position of the antiquotation. An example is the cterm antiquotation, e.g. `@{cterm "a = b"}`.

Following the static nature of antiquotations this abstract value is produced statically during compile time and is bound to some value as in *val x = ...compile−time−context...*, and then this value x is what appears in the expanded ML text. So to implement such an antiquotation means to provide two main ingredients: the code for the value binding (referred to as environment) and the code to reference that value (referred to as body). This is what the interface `ML_Antiquotation.declaration` offers. The argument `Proof.context -> string * string` is the central point. The pair of strings denotes the ML-text for the environment and the body respectively. See for example the theorem antiquotation: `@{thm refl}`. This can also be considered a *value* antiquotation, albeit being implemented by the more low-level interface.

Digging even deeper into `ML_Context.add_antiquotation` the `ML_Context.decl` also refers to a pair of ML-code, denoting an environment and the body, but here presented as `ML_Lex.token list` not as a string. This interface gives a lot of flexibility into the design of antiquotations. A notable example is the 'instantiate'-antiquotation. e.g:

⟨*ML*⟩

Note the nesting of antiquotations in that example. The 'environment-part' here consists of all the right hand sides of the instantiations as well as the proposition. The 'body-part' is an ML-expression that instantiates the

[1] Note that there is also a variant without the 'embedded'. suffix `ML_Antiquotation.inline`. What is the difference? According to a mail thread https://lists.cam.ac.uk/mailman/htdig/cl-isabelle-users/2021-October/msg00023.html the 'embedded' is a hint to Isabelle that antiquotation expects its argument as an 'embedded language' enclosed by a cartouche. This can nowadays be considered as the canonical way. Historically, Isabelle first distinguished between the outer-syntax (e.g. Isar or plain ML as Meta-Language) and the inner-syntax of the object logic like HOL or FOL. Meanwhile Isabelle embraces a lot of different languages that in some cases can even be nested, and the languages can be dynamically extended by declaring new commands or antiquotations.

proposition with the terms. The body part cannot be immediately evaluated during compile time as the value of parameter t of the function is not yet known. So the result is an ML-expression that references t among the right and sides of the instantiations that are evaluated at compile time.

The special-form antiquotation packs the text in a function thunk, e.g. `\<^try>\<open>I\<close>`.

26.15 Markup and Reports

Isabelle provides a rich set of markup to display information in Isabelle/-jEdit. Markup can either be directly attached to a string, or an existing source text can be decorated by sending *reports* to PIDE. An example for direct markup is the printing of terms, e.g.:

⟨*ML*⟩

You can make the markup visible by sending it to an output function, e.g.

⟨*ML*⟩

The markup is itself encoded in the string by the format `YXML`.

The markup can be expanded to the `XML.body` by `YXML.parse_body`:

⟨*ML*⟩

The plain text (without markup) can be extracted like this:

⟨*ML*⟩

It is also possible to produce pretty printed output without markup using `Pretty.pure_string_of` (see also `Pretty.string_of_ops` and `Pretty.pure_output_ops`).

To build up markup you can use `Markup`. The type for markup is a pair of element name and a list of properties. A property `Properties.T` is a key value pair which are both strings. First you build up your markup by using the various functions supplied in `Markup`, then you attach it to a string via `Markup.markup`.

⟨*ML*⟩

With `Position.report` you can add markup to a given position, e.g. here we add the warning-twigglies and an url:

⟨*ML*⟩

The standard message channels `error`, `warning` and `tracing` scan the string for position information and automatically markup the position.

⟨*ML*⟩

There are some fine points regarding the command region were the message is issued. By default markup is only shown in the actual command region.

⟨*ML*⟩

We do not get the wiggly lines in the **ML** above, instead the **ML-val**

⟨*ML*⟩

We can put markup to a previous command if we first safe the command position of the first command, and temporary use this in the second command when issuing the message.

⟨*ML*⟩

To inspect the markup attached to the source you can use the panel Sidekick and select the language "isabelle-markup" When you hover over an entity in the upper half of the panel you see the attached markup in the lower half of the panel.

Reports are also used in autocorres `Feedback` to provide warning and error markup for C programs.

An illustrative example for custom markup and reports can be seen in the document antiquotation for rail diagrams: is `$ISABELLE_HOME/src/Pure/Tools/rail.ML`. It is used extensively in the Isabelle documentation, watch out for `\<^rail>\<open>\<close>` e.g. in `$ISABELLE_HOME/src/Doc/Datatypes/Datatypes.thy`

26.16 Term Synthesize via Intro Rules

A repeated task in the various AutoCorres phases is to synthesize a typically more abstract monadic function (output) from a given monadic function (input) together with a simulation theorem that connects both versions of the function. One strategy is to formulate introduction rules for the correspondence relation for the different language constructs and then recursively apply those rules guided by the constructs in the "input" function, and as a side effect of the rule application to synthesize the "output" function. Examples are, heap abstraction, word abstraction, exception rewriting (as described above) and type strengthening. Currently there is not (yet) an uniform implementation of this strategy, but the different instances are converging. Currently the most recent incarnation is the setup for type strengthening to perform the *refines* proofs:

- A term net is used for efficient lookup of potentially matching rules. It is indexed by the input function as well as the desired result relation.
- Splitting of tuples in rules is automatically performed to match the current term.
- Priorities of rules are specified to guarantee that the most specific rules are tried first
- A cache is used both for positive and negative proof attempts. As there might be multiple applicable rules when decomposing the input

function, partial results might already be proven or have failed. Using a cache helps to prune repeated subproof attempts.

The setup for this phase can be found in `../Refines_Spec.thy`. It employs **synthesize-rules**, as defined in `../lib/ml-helpers/Synthesize.thy`. This command can be seen as generalisation of **named-theorems**, with support for the kind of features just sketched.

For a set of **synthesize-rules** you can generate patterns to support indexing of rules via subterms: **add-synthesize-pattern**.

With **print-synthesize-rules** you can inspect the set:

print-synthesize-rules *pure*

You can also supply an term to select the matching rules:

print-synthesize-rules *pure* ‹*Trueprop* (*refines-spec* (*L2-try f*) - (*return ?f'*) - (*rel-prod rel-liftE* (=)))›

Rules are added via attribute *synthesize-rule*, where you can also specify the rule sets the priority and whether to split some variables. In order to preserve the theorem name for debugging purposes you should apply that attribute in a separate **lemmas** and not directly in the original **lemma**

The main tactic to drive application of those rules is `CT.cache_deepen_tac`. This tactic has type `context_tactic` and supports implementing a cache within the `Proof.context`. The user provides a cache and a single-step tactic that is recursively tried on the emerging subgoals.

The interface of the cache is captured in record `CT.ctxt_cache`:

- `#lookup: CT.ctxt_cache -> Proof.context -> cterm -> int -> context_tactic` The lookup function gets the goal presented as a `cterm` and returns a subgoal tactic. A cache miss is implemented by `K CT.no_tac`.
- `#insert: CT.ctxt_cache -> (Timing.timing * int * int) -> thm -> Proof.context -> Proof.context` The insert functions gets timing information tupled with the total number of alternatives and the current alternative as input together with the just proven subgoal. It can use that information to decide whether to really extend the cache or to ignore the result.
- `#propagate: CT.ctxt_cache -> Proof.context -> Proof.context -> Proof.context` To propagate the cache throughout backtracking the propagate function is supplied. It gets the current context and the old context as argument and can propagate the cache from the current context to the old one (to which it backtracks).

The cache used for the type strengthening phase is implemented in `Monad_Convert.sim_nondet`. It is based on `Synthesize_Rules.gen_cond_cache`. It uses a term net to index the cached results and only adds rules for compound statements like *L2-seq*, not for atomic ones. The idea here is that proving simulation for an atomic statement is just a single rule application and is thus not worth to be cached.

Failed proof attempts are represented by $FALSE \Longrightarrow PROP\ ?P$ instantiated with the failed subgoal. This is how they are presented to the cache and can be stored. When a cache lookup is performed and the cache results in such an instance of $FALSE \Longrightarrow PROP\ ?P$ which matches the current subgoal the proof attempt will be immediately aborted at that depth and potential alternative proof steps 'higher up' (smaller depth) may get explored.

Next we illustrate the approach with a small example.

lemma *refines-liftE-nondet*: *refines* (*liftE f*) *f s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemma *refines-L2-unknown*:
shows *refines* (*L2-unknown ns*) (*L2-VARS* (*select UNIV*) *ns*) *s s* (*rel-prod rel-liftE* (=))
⟨*proof*⟩

lemmas *refines-spec-monad-rules* =
refines-L2-seq-nondet [*simplified THIN-def*]
refines-L2-gets-nondet
refines-L2-unknown

lemmas *refines-option-monad-rules* =
refines-L2-seq-option [*simplified THIN-def*]
refines-L2-gets-option

schematic-goal ‹*refines* (*L2-seq* (*L2-gets* (λ-. *n*::*nat*) []) (λ*n*. *L2-gets* (λ-. *n* + *m*) [])) (*?f'*::(*?'c* , *'a*) *res-monad*) *?s ?s ?R*›

⟨*proof*⟩

First we define the single-step tactics, which just try some rules.

⟨*ML*⟩

Now we define a simple cache which stores all emerging subgoals in a list and tries to resolve with those subgoals on a lookup.

⟨*ML*⟩

We only try nondet-rules, so we end up in the nondet monad.

schematic-goal
‹*refines* (*L2-seq* (*L2-gets* (λ-. *n*::*nat*) []) (λ*n*. *L2-gets* (λ-. *n* + *m*) []))

(*?f'*::(*?'c*, *'a*) *res-monad*) *?s ?s ?R*›
⟨*proof*⟩

schematic-goal
‹*refines* (*L2-seq* (*L2-gets* (λ-. *n*::*nat*) []) (λ*n*. *L2-gets* (λ-. *n* + *m*) [])) (*?f'*::(*?'c*, *'a*) *res-monad*) *?s ?s ?R*›
thm *refines-option-monad-rules*
⟨*proof*⟩

We only try the option-rules, so we end up in the option monad.

schematic-goal
‹*refines* (*L2-seq* (*L2-gets* (λ-. *n*::*nat*) []) (λ*n*. *L2-gets* (λ-. *n* + *m*) [])) (*?f'*::(*?'c*, *'a*) *res-monad*) *?s ?s ?R*›
⟨*proof*⟩

No we try both the option-rules and the nondet-rules. As the option rules come first we end up in the option monad

schematic-goal
‹*refines* (*L2-seq* (*L2-gets* (λ-. *n*::*nat*) []) (λ*n*. *L2-gets* (λ-. *n* + *m*) [])) (*?f'*::(*?'c*, *'a*) *res-monad*) *?s ?s ?R*›
⟨*proof*⟩

By including a *L2-unknown* in the example we have to end up in the nondet monad, as it cannot be handled in the option monad. Note the difference in the outcome of the fist and second proof attempt. In the first one we only provide the nondet-rules in the second one we also provide the option-rules. This results in a different "translation" of the first *L2-gets*.

schematic-goal
‹*refines* (*L2-seq* (*L2-gets* (λ-. *n*::*nat*) []) (λ*n*. *L2-unknown* [])) (*?f'*::(*?'c*, *'a*) *res-monad*) *?s ?s ?R*›
⟨*proof*⟩

schematic-goal
‹*refines* (*L2-seq* (*L2-gets* (λ-. *n*::*nat*) []) (λ*n*. *L2-unknown* [])) (*?f'*::(*?'c*, *'a*) *res-monad*) *?s ?s ?R*›
⟨*proof*⟩

end

26.17 Pointers to Local Variables

theory *pointers-to-locals* **imports** *AutoCorres*
begin

install-C-file *pointers-to-locals.c*
init−autocorres [*addressable-fields* = *pair.first buffer.buf 32 word*[*3*]]*pointers-to-locals.c*
autocorres [*no-heap-abs* = *inc-untyp*]*pointers-to-locals.c*

This story began with the desire to support "pointers to local variables". The idea to support "pointers to local variables" might be quite fearful as it covers lots of use cases. When trying to literally support pointers to local variables (uniform with heap pointers) one has to answer questions about the memory layout of heap and stack, how to ensure that those regions are disjoint, what happens if we run out of stack space, how do we make sure that there is no dangling pointers to a stack-frame that is already popped from the stack etc.

So let us make a step back and ask ourselves for which C-idiomatic use cases are pointers to local variables actually used. C does only support call-by-value. So pointers are sometimes used to model call-by-reference semantics, for input as well as output parameters. Especially, the use case of an additional output parameter (besides the regular return value) is often implemented as a pointer parameter. An alternative might be to return a tuple encoded as a structure instead. But as there are no ad hoc tuples in C, this requires the boilerplate to define an auxiliary structure type. So typically a pointer parameter is used instead.

Here an example of a integer addition with overflow check. The return value is used for the status, the actual result is returned as a pointer parameter.

```
int add_check(int* result, int a, int b)
{
    *result = a + b;
    if(a > 0 && b > 0 && *result < 0)
        return -1;
    if(a < 0 && b < 0 && *result > 0)
        return -1;
    return 0;
}
```

A call to the function could be the following with a local variable for the result.

```
int res, status;
status = add_check(&res, 2, 3)
```

Here `res` is an output parameter. If C would support tuples we might have code like:

```
int add_check(int a, int b)
{
    int result = a + b;
    if(a > 0 && b > 0 && *result < 0)
        return (-1, resut);
    if(a < 0 && b < 0 && *result > 0)
        return (-1, result);
```

```
    return (0, result);
}


  int res, status;

  (status, res) = add_check(2, 3)
```

Another use case is to have in input / output parameter implemented by a pointer parameter.

```
void inc(int *a)
{
  (*a)++;
}

int x = 42;
inc(&x);
```

This could be also modelled as explicit input and output:

```
int inc(int a)
{
  return (a++);
}

int x = 42;
x = inc(x);
```

So one general idea is to model those kind of pointer-parameters as explicit input / output parameters. From a pattern like:

```
int f(int *p1, ..., int * p_n, int a1, ..., int a_m) {
  ... (*p1) ...
  ... (*p_i) ...
  return res;
}

r = f(q1,..., q_n, b1, ..., b_m)
```

We make

```
(int * int ... * int) f(int p1, ..., int p_n, int a1, ..., int
a_m) {
    ... p1 ...
```

```
  ... p_i ...
  return (res, p1, ... p_n);
}

(r, q1, ..., q_n) = f(q1,..., q_n, b1, ..., b_m)
```

In which cases is such a model faithful? Different behaviours might occur, when pointer-parameters are aliased. Moreover, the translation scheme only works out if the body of the function only uses the dereferenced pointer variables, i.e. `*p_i` to obtain the value or perform an update, and does not actually use the literal address value stored in the variable. E.g. it does not make things like `p1 = p2` or stores the pointer value in some global data structure. All examples we have seen so far are benign with this respect, as there was only one pointer-variable involved, and we only used it to dereference it.

What about this swap function:

```
void swap1(int* p, int* q) {
  int tmp = *p;
  *p = *q;
  *q = tmp;
}

int a = 1; int b =2;
swap1(&a, &b);
/* a = 2, b = 1 */


(int * int) swap2(int p, int q) {
  int tmp = p;
  p = q;
  q = tmp;
  return (p, q);
}

int a = 1; int b = 2;
(a, b) = swap2(a, b);
/* a = 2, b = 1 */
```

These two swap functions behave the same, even if the parameters would alias. But in general aliasing is problematic.

```
void inc_both1(int* p, int* q) {
  *p = *p + 1;
  *q = *q + 1;
```

```
}

int x = 2;
void inc_both1(&x, &x);

/* x = 4 */


(int * int) inc_both2(int p, int q) {
  p = p + 1;
  q = q + 1;
  return (p, q);
}

int x = 2;
(x, x) = inc_both2(&x, &x);

/* x = 3 */
```

Whats the relevant difference between the increment and the swap example? In the swap example, we never read from a pointer-parameter after any pointer-parameter was assigned to. So an update within the function might not alter the values we read via any pointer-parameter.

26.17.1 Design choices

C does not distinguish pointers to local variables from heap pointers they are all the same. E.g. the increment functions above could be applied to a heap location as well as a stack location. Moreover, in the context of systems programming it is common that low-level code explicitly deals with heap-layout and pointer values and "tricks" like storing some flags as bits in the pointer values themselves are often used. That is why we aim for an uniform model for stack and heap pointers and don't want to impose assumptions on the layout.

Whenever a function creates a pointer to a local variable, we model it as part of the heap. Core ideas:

- As part of ordinary heap-typing, we track free stack locations by a distinguished C type *stack-byte*. All addresses with that type are considered free stack location.
- Additionally to the typing information with *stack-byte* (that denotes free stack space) We have a fixed set of addresses $\mathcal{S}$ that describe all the stack space (free and allocated). Note that the set $\mathcal{S}$ does not

depend on the state. The dynamic aspects are modelled within the heap typing.

- As prelude in a function that needs pointers to local variables we allocate the variable in the heap, by non-deterministically retyping a region of free stack space that fits: We retype from *stack-byte* to the type we allocate.
- All accesses / updates to the variable within the function are modelled indirectly via the pointer
- As postlude of the function we again retype the stack space as free.

  Observations:

- Ordinary C functions, that do not have an AUX-UPDATE on the heap-type also do not mess around with the stack as they do not change heap-typing. So stack-space is preserved when calling a function.
- Prelude / postlude ensures that the stack typing is the same after a function call to a function that has pointers to local variables.

Stack Allocation

The central definition for stack allocation is *stack-allocs* describing the set of possible pointers and modified heap type descriptions:

$(p,\ d') \in$ *stack-allocs* $n\ \mathcal{S}\ TYPE('a)\ d$

thm *stack-allocs-def*

After such an allocation we know that:

- Pointer p is a root pointer with the expected type in the new heap type description d'.
- The allocated addresses are within set $\mathcal{S}$.
- The new heap typing d' is obtained from the original heap typing d by retyping addresses of *stack-byte* to the type of the pointer p. The address of pointer p denotes the start the retyped region. The parameter n can be used to allocate a consecutive range of pointers in one step.

  Stack allocation might fail in the sense that the resulting set of *stack-allocs* can be empty. In Simpl we then fail in the terminal fault state reporting *StackOverflow*. In the autocorres monad we currently have decided to ignore possible stack overflows. We simply assume that stack allocation succeeds and does not return an empty set.

Stack Release

The counter part to the stack allocation above is *stack-releases n p d′*.

thm *stack-releases-def*

There is no non-determinism here. It is a plain function retyping back to *stack-byte*.

Stack discipline

We encapsulate the stacking discipline of allocation / release in some functions in the various layers.

Simpl In Simpl we have the command *With-Fresh-Stack-Ptr n init c* where

- *n* denotes the number of consecutive pointers we want to allocate, typically *1*. This number was introduced to allocate local arrays and obtain pointers to the elements. However, it turns out that allocating an array pointer $('a['b])$ *ptr* instead is sufficient. Hence the parameter is actually always *1*.
- *init s* returns a set of initial values from which we non-deterministically choose one. In case the addressed local variable is a function parameter this is a singleton set containing the value of the argument. In case of an ordinary local variable this is either the the universal set of all possible values (if uninitialized) or a singleton set of the initialisation value.
- *c* expects a fresh pointer *p* and then yields a Simpl command of the body.

  The idea is simple, we first allocate a pointer *p* with *stack-allocs*, then initialise the fresh heap location according to *init* then execute the body *c p* and finally release the stack pointer again.

thm *globals.With-Fresh-Stack-Ptr-def*
term *stack-heap-state.With-Fresh-Stack-Ptr*

A central role of the definition is the *DynCom* statements, which is the means of Simpl to bind certain execution points and provide state dependent commands. As the body of a *DynCom* always depends on the complete state (and not such a thing like the pointer *p*) we use the function *allocated-ptrs* calculate the fresh pointer from the typing information.

thm *allocated-ptrs-def*
thm *stack-allocs-allocated-ptrs*

Note that *spec-monad.Spec* might fail if the resulting set is empty, which is exposed to the failure *StackOverflow* of the command.

L1 / L2 In both L1 and L2 we use *globals.with-fresh-stack-ptr n init f*, where n as well as*init* are just the same as in Simpl and f is a monadic computation depending on the fresh pointer p.

term *globals.with-fresh-stack-ptr*
thm *globals.with-fresh-stack-ptr-def*
term *stack-heap-state.with-fresh-stack-ptr*

In contrast to Simpl we assume that stack allocation succeeds by using *assume-result-and-state*.

HL / Split Heap

In the split heap model there is no singleton heap anymore. Hence we introduce a family of functions depending on the type that is allocated.

term *heap-w32.with-fresh-stack-ptr*
thm *heap-w32.with-fresh-stack-ptr-def*
term *heap-w32.guard-with-fresh-stack-ptr*
thm *heap-w32.guard-with-fresh-stack-ptr-def*
term *heap-w32.assume-with-fresh-stack-ptr*
thm *heap-w32.assume-with-fresh-stack-ptr-def*
term *typ-heap-typing.with-fresh-stack-ptr*
term *typ-heap-typing.guard-with-fresh-stack-ptr*
term *typ-heap-typing.assume-with-fresh-stack-ptr*

While stack allocation remains the same upon stack release some more work has to be done in order to make the simulation work.

In the monolitic heap the stack release of a pointer p immediately results in *plift* $(h,$ *stack-releases* $n\ p\ d')\ p =$ *None*. So the simulation demands *the-default ZERO*$('a)$ $($*plift* $(h,$ *stack-releases* $n\ p\ d')\ p) =$ *zero*. [2].

So to simulate this in the split heap we explicitly zero out the memory. Moreover, for the simulation to work we have to argue that the stack release did not mess up with any of the other split heaps. This argument can be given if know that pointer p is still a valid root pointer before we do the release. As we know that p is a valid root pointer immediately after the allocation we have to establish that this is still true at stack release. Most C function do not alter the heap typing at all. So it can easily be shown that $f \cdot s$ *?*⦃ $\lambda r.$ *unchanged-typing-on* $\mathcal{S}\ s$ ⦄ holds and thus that p is still a valid root pointer after executing f. In case we cannot automatically proof the preservation of typing information, we enforce that p is still a valid root pointer by inserting a guard. So for code that changes the heap typing in more involved ways the argument is left for the user to prove.

thm *lifted-globals.unchanged-typing-on-def*
term *heap-typing-state.unchanged-typing-on*

Some syntax

[2] c.f. `open_struct.thy` for details on heap lifting simulation

term *PTR-VALID*(*32 word*)
term *IS-VALID*(*32 word*) *s*
term *IS-VALID*(*32 word*) *s p*

context *ts-definition-inc*
begin
thm *ts-def*
end

Pointer to an uninitialized local variable.

context *call-inc-local-uninitialized-impl*
begin
thm *call-inc-local-uninitialized-body-def*
end

context *ts-definition-call-inc-local-uninitialized*
begin
thm *ts-def*
end

Pointer to an initialized local variable.

context *ts-definition-call-inc-local-initialized*
begin
thm *ts-def*
end

Pointer to parameter.

context *ts-definition-call-inc-parameter*
begin
thm *ts-def*
end

When we cannot prove that the heap typing is unchanged we fall back to *heap-w32.guard-with-fresh-stack-ptr* instead of *heap-w32.assume-with-fresh-stack-ptr*

context *ts-definition-call-inc-untyp*
begin
thm *ts-def*
end

Nested pointers among the phases.

context *call-inc-nested-impl*
begin
thm *call-inc-nested-body-def*
end

context *l1-definition-call-inc-nested*
begin
thm *l1-def*

end

context *l2-definition-call-inc-nested*
begin
thm *l2-def*
end

context *hl-definition-call-inc-nested*
begin
thm *hl-def*
end

context *wa-definition-call-inc-nested*
begin
thm *wa-def*
end

context *ts-definition-call-inc-nested*
begin
thm *ts-def*
end

Global variable

context *ts-corres-call-inc-global*
begin
thm *ts-def*
end

Local array variable.

context *ts-definition-call-inc-array*
begin
thm *ts-def*
end

Local structure variable.

context *ts-definition-call-inc-first*
begin
thm *ts-def*
end

Local array in structure variable.

context *ts-definition-call-inc-buffer*
begin
thm *ts-def*
end

Mutual recursion and pointer to local variables.

context *l2-definition-odd-even*
begin

thm *unchanged-typing*
end
context *ts-corres-odd-even*
begin
thm *ts-def*
end

Proof rules for *runs-to-vcg*

context *pointers-to-locals-all-corres* **begin**

thm *stack-ptr-simps*

lemma (*call-inc-local-initialized'*) · *s* {‖ λ *r t*. *t* = *s* ∧ *r* = *Result 43* ‖}
⟨*proof*⟩

lemma (*call-inc-local-uninitialized'*) · *s* {‖ λ *r t*. *t* = *s* ∧ *r* = *Result 42* ‖}
⟨*proof*⟩

lemma (*call-inc-parameter'* *n*) · *s* {‖ λ *r t*. *t* = *s* ∧ *r* = *Result* (*n* + *1*) ‖}
⟨*proof*⟩

end

26.17.2 Open Ends / TODOs

There are not yet any user level proof rules for *heap-w32.guard-with-fresh-stack-ptr* and *heap-w32.assume-with-fresh-stack-ptr* to include with *runs-to-vcg*. Note that there are a lot of theorems on the primitives *stack-allocs* and *stack-releases*.

thm *stack-allocs-cases*
thm *stack-allocs-ptr-valid-cases*
thm *stack-releases-ptr-valid-cases*
thm *stack-releases-ptr-valid-cases1*

Value parameters aka. In-Out parameters

Now that we have proper pointers to local variables there is room for further abstractions, motivated by the prominent use cases described in the beginning. Those use cases suggest that in a lot of cases one can completely get rid of stack allocation / release by introducing value parameters: Instead of just passing a pointer value into the function, one passes in the actual (dereferenced) value and tuples the return value of the function with the final (dereferenced) value of that parameter.

It turns out that the core transformation here is not so much about pointers to local variables but more on transforming functions from passing in a pointer parameter to another function value parameters. This part of

the transformation is independent of the question heap pointer vs. stack pointer. After this transformation is done for the body of a function which allocates a stack pointer, the actual address of the stack pointer becomes meaningless as it is only used in "dereferenced form". So we can remove the stack allocation / stack release and replace it by an ordinary local variable.

See `In_Out_Parameters_Ex.thy` for more information.

end

26.18 In-Out Parameters, Abstracting Pointers to Values

theory *In-Out-Parameters-Ex* **imports** *AutoCorres*
begin

consts *abs-step*:: *word32* ⇒ *word32* ⇒ *bool*

26.18.1 Overview

We introduce a new phase in AutoCorres to replace pointer parameters by *in/out parameters*: Instead of passing a pointer into a function we pass the initial value of the pointer (by dereferencing the pointer) into the function and return the value at the end of the function as additional output. E.g. a function `void inc(unsigned *n)` becomes `unsigned inc(unsigned n)`. The initial motivation for this phase was to get rid of explicit pointers to local variables and replace them by ordinary values instead. This conversion has two main aspects:

- Function signatures may change: pointer parameters become value parameters and the function may return the value of the pointer at the end as an additonal return value (tupled with the ordinary return value). Functions with side effects on the heap may become pure functions on values by this transformation.
- As a result of the first transformation, pointers to local variables may be eliminated and be replaced by bound variables. This means that *stack-heap-state.with-fresh-stack-ptr* disappears.

  The new phase is called *IO* and is placed between L2 and HL. This means local variables are already represented as bound variables in L2 and we still operate on a monolithic heap.

26.18.2 Building Blocks

What ingredients do we need for the abstraction relation in the refinement proof?

- Heaps: As we eliminate pointers, the original function manipulates the heap more often then the resulting function. So we need a notion to relate the heap states and keep track of the relevant pointers. As the stack of local variable pointers is also modelled in the heap the stack free locations are also of interest. The heaps of the original and the resulting program may differ in these locations. The heap value of a stack free location should be irrelevant for the program.

- Function signatures: The function signature changes, pointer parameters become ordinary values, the return value is tupled with output parameters.

  In the following we refer to the original state (containing the heap) as s and the resulting / abstract state as t. As I first idea we want to relate states s and t such that the heaps are the same except for some pointers we want to eliminate and the stack free locations. It would be natural to describe this as a relation. However, it turns out that dealing with relations within the refinement proof are hard to make admissible in order to support recursive functions. Having a abstraction / lifting function (instead of a relation) from s to t makes admissibility straightforward.

  As a prerequisite we encourage the reader to consider the documentation about pointers to local variables: `pointers_to_locals.thy`

context *heap-state*
begin

frame

We want to express that certain portions of the heap (typing and values) are 'irrelevant', in particular regarding (free) stack space. The notion of 'irrelevant' is a bit vague, it means that the behaviour of the resulting program does not depend on those locations and also that it does not modify those locations. Moreover, we prefer an abstraction function rather than an relation between states to avoid admissibility issues for refinement. The central property is $t = \mathit{frame}\ A\ t_0\ s$, for A being a set of addresses and t, t_0 and s being states. Think of A as the set of *allocated* addresses containing those pointers we want to *abstract* aka. eliminate.

thm *frame-def*

The standard use case we have in mind is $A \cap \mathit{stack\text{-}free}\ (\mathit{htd}\ s) = \{\}$, hence $A - \mathit{stack\text{-}free}\ (\mathit{htd}\ s) = A$, but nevertheless the intuition is:

- stack free typing from s is preserved, the framed sate has at least as many stack free addresses as the original one. So we can simulate any stack allocation.

- heap values for stack free and A are taken from reference state t_0, this captures that we do not depend on the original values in s for those addresses.
- typing for allocations in A is taken from reference state t_0, this captures that we do not depend on the original typing in s for those addresses.

  By taking the same reference state t_0 to frame two states s and s', we can express that the 'irrelevant' parts of the heap did not change in the respective framed states *frame* A t_0 s and *frame* A t_0 s'.

rel-alloc, *rel-stack*, *rel-sum-stack*

The core invariant between the states that is maintained by the refinement is *rel-alloc* $\mathcal{S}$ M A t_0 s t:

- $\mathcal{S}$ is a static set of addresses containing the stack. Stack allocation and stack release are contained within $\mathcal{S}$.
- M is the set of addresses that might be modified by the original program.
- A is the set of addresses that are eliminated (abstracted) in the resulting program. Another good intuition about the set A is that the original function might read from and depend on those addresses but the resulting function does not. Moreover, the resulting function does not change any of heap locations in A. The resulting function is agnostic to those locations. It neither reads nor modifies them. For any heap valuation of A the abstract function simulates the original one.
- t_0 is the reference state explained above.
- s is the state of the original program
- t is the state of the resulting program.

thm *rel-alloc-def* [*of* $\mathcal{S}$ M A t_0]

The properties of *rel-alloc* are:

- $t =$ *frame* A t_0 s, the resulting heap in t is the same as the original in s, except for the addresses in A and the stack free space.
- *stack-free* (*htd* s) $\subseteq \mathcal{S}$: The stack free space is contained in $\mathcal{S}$.
- *stack-free* (*htd* s) $\cap$ $A = \{\}$: We only want to eliminate properly allocated pointers, which are not contained in the stack free space.

- *stack-free* (*htd s*) ∩ *M* = {}: We only mention properly allocated pointers to be modified. Modifications within the stack free space are irrelevant, as they are abstracted by the framing anyways.

For an original function f and the abstract aka. lifted function g the refinement property has the following shape: *rel-alloc* $\mathcal{S}$ *M A* t_0 *s t* $\Longrightarrow$ *refines f g s t* (*rel-stack* $\mathcal{S}$ *M1 A s* t_0 (*rel-xval-stack L R*))

Note that the output relation of *refines* relates both the resulting states and return values to each other. The return values of the error monad are related by *rel-sum-stack*, taking a relation L for error values (*Inl*) and the R for normal termination (*Inr*).

thm *rel-sum-stack-def*
thm *rel-stack-def* [*of* $\mathcal{S}$ *M1 A s* t_0 *rel-sum-stack L R*]

Consider the output states s' and t' and the output values v and w for the original and the abstracted function. Then we have:

- *rel-xval-stack L R* (*hmem s'*) *v w*: The result values are related, (see below).
- *rel-alloc* $\mathcal{S}$ *M1 A* t_0 *s' t'*: In particular t' = *frame A* t_0 *s'*. Note that we use the same A and the same reference state t_0 as for the initial states. So this means that the abstract program g does not change the heap values and typing in A and does not depend on the values.
- *equal-upto* (*M1* ∪ *stack-free* (*htd s'*)) (*hmem s'*) (*hmem s*): The original program only modifies pointers contained in *M1* or in the stack free portions of the heap (e.g. by nested function calls).
- *equal-upto M1* (*htd s'*) (*htd s*): Changes to the heap typing of the original program are confined within set *M1*.
- *equal-on* $\mathcal{S}$ (*htd s'*) (*htd s*): The stack typing of the original program remains unchanged. Note that it might change temporarily by nested function calls but the stacking discipline restores the state. By construction we have *M1* ⊆ *M*.

Stacking pointers: *rel-singleton-stack*, *rel-push* The relation on the output value captures the idea that portions of the heap are stacked / tupled into result values. On the boundaries of a function the relation on error values L is always the trivial λ-. (=). Errors are terminal as there is no exception handling in C. Within the boundaries of a function there can be more complex relations reflecting the nesting of break / continue / return and goto. For ordinary values the values of pointers are tupled by

rel-singleton-stack, in case the original function returned no result (void), or nested by *rel-push* in case the original function returned some result.

thm *rel-singleton-stack-def*
thm *rel-push-def*

A typical relation is R = *rel-singleton-stack p* for some pointer *p*. This means that the abstract value can be obtained by looking into the heap at pointer *p*: *rel-singleton-stack p h* () *v* means that *v* = *h-val h p*. The heap of the original program is propagated down the into the relations. Similar *rel-push p* (λ-. (=)) *h x* (*v*, *x*) relates the result value *x* to the tuple (*v*, *x*) where *v* = *h-val h p*. Note that *rel-push* can be nested represent multiple output parameters. Depending on the role of the pointers there are additional assumptions about the pointers, in particular things like *ptr-span p* ⊆ *A* or *ptr-span p* ⊆ *M* and disjointness properties like *distinct-sets* [*ptr-span p*, *ptr-span q*].

IOcorres

There is an additional predicate *IOcorres* which represents the result of the refinement proof in the canonical autocorres ideom that is used within the other phases. This form is what is expected when constructing the final theorem combining all autocorres layers ⟦*L1corres ?check-termination ?Gamma ?c-L1.0 ?c*; *L2corres ?st-L2.0 ?rx-L2.0 ?ex-L2.0 ?P-L2.0 ?c-L2.0 ?c-L1.0*; *IOcorres ?P-IO ?Q-IO ?st-IO ?rx-IO ?ex-IO ?c-IO ?c-L2.0*; *L2Tcorres ?st-HL ?c-HL ?c-IO*; *corresTA ?P-WA ?rx-WA ?ex-WA ?c-WA ?c-HL*; ⋀*s*. *refines ?c-WA ?A s s* (*rel-prod rel-liftE* (=))⟧ ⟹ *ac-corres′* (λ-. *Exception default*) (*?st-HL* ∘ *?st-IO* ∘ *?st-L2.0*) *?check-termination* {*AssumeError*, *StackOverflow*} *?Gamma* (λ*s*. (*?rx-WA* ∘ (λ*v*. *?rx-IO v* (*?st-L2.0 s*)) ∘ *?rx-L2.0*) *s*) (λ*s*. (*?ex-WA* ∘ (λ*e*. *?ex-IO e* (*?st-L2.0 s*)) ∘ *?ex-L2.0*) *s*) (*?P-L2.0 and ?P-IO* ∘ *?st-L2.0 and ?P-WA* ∘ *?st-HL* ∘ *?st-IO* ∘ *?st-L2.0*) *?A ?c*

⟦*L1corres ?check-termination ?Gamma ?c-L1.0 ?c*; *L2corres ?st-L2.0 ?rx-L2.0 ?ex-L2.0 ?P-L2.0 ?c-L2.0 ?c-L1.0*; *IOcorres ?P-IO ?Q-IO ?st-IO ?rx-IO ?ex-IO ?c-IO ?c-L2.0*; *L2Tcorres ?st-HL ?c-HL ?c-IO*; *corresTA ?P-WA ?rx-WA ?ex-WA ?c-WA ?c-HL*; ⋀*s*. *refines ?c-WA ?A s s* (*rel-prod* (=) (=))⟧ ⟹ *ac-corres′ Exn* (*?st-HL* ∘ *?st-IO* ∘ *?st-L2.0*) *?check-termination* {*AssumeError*, *StackOverflow*} *?Gamma* (λ*s*. (*?rx-WA* ∘ (λ*v*. *?rx-IO v* (*?st-L2.0 s*)) ∘ *?rx-L2.0*) *s*) (λ*s*. (*?ex-WA* ∘ (λ*e*. *?ex-IO e* (*?st-L2.0 s*)) ∘ *?ex-L2.0*) *s*) (*?P-L2.0 and ?P-IO* ∘ *?st-L2.0 and ?P-WA* ∘ *?st-HL* ∘ *?st-IO* ∘ *?st-L2.0*) *?A ?c*. The *refines* version and the *IOcorres* version can be converted into each other by ⟦*L1corres ?check-termination ?Gamma ?c-L1.0 ?c*; *L2corres ?st-L2.0 ?rx-L2.0 ?ex-L2.0 ?P-L2.0 ?c-L2.0 ?c-L1.0*; *IOcorres ?P-IO ?Q-IO ?st-IO ?rx-IO ?ex-IO ?c-IO ?c-L2.0*; *L2Tcorres ?st-HL ?c-HL ?c-IO*; *corresTA ?P-WA ?rx-WA ?ex-WA ?c-WA ?c-HL*; ⋀*s*. *refines ?c-WA ?A s*

s (*rel-prod rel-liftE* (=))⟧ ⟹ *ac-corres′* (λ-. *Exception default*) (*?st-HL* ∘ *?st-IO* ∘ *?st-L2.0*) *?check-termination* {*AssumeError*, *StackOverflow*} *?Gamma* (λ*s*. (*?rx-WA* ∘ (λ*v*. *?rx-IO v* (*?st-L2.0 s*)) ∘ *?rx-L2.0*) *s*) (λ*s*. (*?ex-WA* ∘ (λ*e*. *?ex-IO e* (*?st-L2.0 s*)) ∘ *?ex-L2.0*) *s*) (*?P-L2.0 and ?P-IO* ∘ *?st-L2.0 and ?P-WA* ∘ *?st-HL* ∘ *?st-IO* ∘ *?st-L2.0*) *?A ?c*

⟦*L1corres ?check-termination ?Gamma ?c-L1.0 ?c*; *L2corres ?st-L2.0 ?rx-L2.0 ?ex-L2.0 ?P-L2.0 ?c-L2.0 ?c-L1.0*; *IOcorres ?P-IO ?Q-IO ?st-IO ?rx-IO ?ex-IO ?c-IO ?c-L2.0*; *L2Tcorres ?st-HL ?c-HL ?c-IO*; *corresTA ?P-WA ?rx-WA ?ex-WA ?c-WA ?c-HL*; ⋀*s*. *refines ?c-WA ?A s s* (*rel-prod* (=) (=))⟧ ⟹ *ac-corres′ Exn* (*?st-HL* ∘ *?st-IO* ∘ *?st-L2.0*) *?check-termination* {*AssumeError*, *StackOverflow*} *?Gamma* (λ*s*. (*?rx-WA* ∘ (λ*v*. *?rx-IO v* (*?st-L2.0 s*)) ∘ *?rx-L2.0*) *s*) (λ*s*. (*?ex-WA* ∘ (λ*e*. *?ex-IO e* (*?st-L2.0 s*)) ∘ *?ex-L2.0*) *s*) (*?P-L2.0 and ?P-IO* ∘ *?st-L2.0 and ?P-WA* ∘ *?st-HL* ∘ *?st-IO* ∘ *?st-L2.0*) *?A ?c*.

thm *IOcorres-def*
thm *IOcorres-refines-conv*
thm *ac-corres-chain-sims*
end

install-C-file *in-out-parameters.c*

26.18.3 Options

To control the in/out- parameter abstraction there are two options of autocorres:

- *skip-io-abs* a boolean which is *true* by default. This skips the complete phase for all functions.
- *in-out-parameters* which takes a list of in-out specifications for functions. As soon as there is at least one specification present the phase is enabled (taking precedence over *skip-io-abs*.

  So to properly enable the IO phase you have two main choices to specify what should be done either in **init−autocorres** or subsequent calls to **autocorres**. You can only disable option *skip-io-abs* in **init−autocorres** and then provide the individual options for *in-out-parameters* in subsequent **autocorres** invocations, or you can already define all *in-out-parameters* already in **init−autocorres**.

  More details on the options are provided in the following examples.

26.18.4 Examples

init−autocorres [*in-out-parameters* =
  compare(*cmp*:*out*) **and**

inc(*y*:*in-out*) **and**
inc-int(*y*:*in-out*) **and**
dec(*y*:*in-out*) **and**
swap(*x*:*in-out*, *y*:*in-out*) **and**
swap-pair(*x*:*in-out*, *y*:*in-out*) **and**
call-inc-ptr(*p*: *in-out*) **and**
inc-twice(*y*:*in-out*) **and**
inc2(*y*:*in-out*, *z*:*in-out*) **and**
heap-inc2-part(*z*:*in-out*) **and**
heap-inc2-part-swap(*y*:*in-out*) **and**
safe-add(*result*: *in-out*) **and**
inc-pair(*p*:*in-out*) **and**
get-arr-idx(*arr*:*in*) **and**
get-pair-arr-idx-second(*arr*:*in*) **and**
inc-pair(*p*:*in-out*) **and**
xyz(*x*:*in-out*) **and**
resab(*res*:*in-out*) **and**
abcmp(*cmpRst*:*in-out*) **and**
out(*out*:*out*) **and**
out-two(*out1*:*out*, *out2*:*out*) **and**
out2(*out*:*in-out*) **and**
out-seq(*out*:*out*) **and**
inc-loop (*x*:*in*, *y*:*in-out*) **and**
call-inc-int-other(*m*: *in-out*) **and**
inc-int-no-exit(*y*:*in-out*) **and**
call-inc-int-other-mixed(*m*: *in-out*) **and**
inc-int2(*y*:*in-out*, *z*:*in-out*) **and**
call-inc-int2(*n*:*in-out*) **and**
call-inc-int-pair(*p*:*in-out*) **and**
call-inc-int-array(*p*:*in-out*) **and**
call-compare-ptr(*m*:*out*) **and**
mixed-global-local-inc(*q*:*in-out*) **and**
g1(*out*:*in-out*) **and**
set-void(*p*:*data*) **and**
set-void2(*p*:*data*) **and**
set-byte(*p*:*data*) **and**
read-char(*p*:*in-out*, *len*: *in-out*) **and**
goto-read-char1(*p*:*in-out*) **and**
goto-read-char5(*elem*: *in*) **and** *smex7*() **and**
cmp-no-io() **where** *disjoint* [] **and**
call-cmp-no-io1() **where** *disjoint* [] **and**
deref-ptr(*p*:*in*) **and**
deref-ptr-glob(*p*:*in*) **and**
call-deref-ptr-glob-single() **and**
call-deref-ptr-glob-pair() **and**
uinc(*p*:*in-out*) **and**
udec(*p*:*in-out*) **and**
call-unop() **for** *p*(*in-out*); **and**
call-binop() **for** *p*(*keep*∗, *keep*∗); **and**

f-empty1(*p*:*in-out*) **and**
f-empty2(*p*:*in-out*) **and**
call-f-only-return(*p*:*in-out*) ,
method-in-out-fun-ptr-specs =
object.unop(*keep*∗) **and**
object.binop(*data*, *data*),

in-out-globals =
keep-inc-global-array
inc-global-array
keep-inc-global-array2
inc-global-array2
call-keep-inc-global-array
call-inc-global-array
mixed-global-local-inc shuffle global-array-update
read-char call-read-char call-read-char-loop
goto-read-char1 goto-read-char2 goto-read-char3 goto-read-char4 goto-read-char5
goto-read-char6
call-cmp-no-io call-cmp-no-io1
deref-ptr-glob call-deref-ptr-glob-pair call-deref-ptr-glob-single
,
ignore-addressable-fields-error,
addressable-fields =
pair.first
pair.second
array.elements
int-pair.int-first
int-pair.int-second

,
ts-force exit = *do-return*
]*in-out-parameters.c*

autocorres [*ts-force option* = *fnc1*] *in-out-parameters.c*

context includes *in-out-parameters-wa-impl-corres*
begin
thm *fun-ptr-intros*
term $\mathcal{P}$*-io-ptr-to-unsigned---unsigned*
end

context includes *in-out-parameters-ts-impl-corres*
begin
thm *fun-ptr-intros*
thm *known-function*
thm *known-function-corres*
end

context *in-out-parameters-hl-corres*

begin
thm *known-function*
thm *known-function-corres*
end

context *in-out-parameters-wa-corres*
begin
thm *known-function*
thm *known-function-corres*
end
context *in-out-parameters-ts-corres*
begin
thm *known-function*
thm *known-function-corres*
end
context *in-out-parameters-l2-corres*
begin
thm *known-function*
thm *known-function-corres*
end

context *in-out-parameters-io-corres*
begin
thm *known-function*
thm *known-function-corres*
end

In option *in-out-parameters* you provide a parameter specification for the functions you want to abstract. The parameter specs following the parameter name are

- *in*: pointer is only used as input to the function. The referenced value does not change during the function call.
- *out*: the pointer is only used to output a result from the function. The referenced value at the beginning is irrelevant, the abstracted function should return the referenced value at the end of the original function.
- *in-out* the pointer is used to input a value and t o return a result.

  When no specification is given for a function, the list is considered empty. This means that the signature of the function shall not be changed by the abstraction. Pointer parameters in the original function stay pointer parameters in the resulting function. Note that this has a quite different effect from just skipping the phase, as autocorres is still attempting to do a proper refinement proof. More details are provided by the examples.

locale *keep-globals* = *in-out-parameters-global-addresses* +

assumes *keep-global-array*: $\{$*ptr-val global-array ..+128*$\} \subseteq \mathcal{G}$
begin
lemma *global-array-contained1*:
assumes *i-bound*: $i < 12$
shows *ptr-span* (*global-array* $+_p$ (*int i*)) $\subseteq \mathcal{G}$
⟨*proof*⟩

lemma *gobal-array-contained2*[*polish*]:
assumes *i-bound*: *numeral i* $<$ (*12*::*nat*)
shows *ptr-span* (*global-array* $+_p$ (*numeral i*)) $\subseteq \mathcal{G}$
⟨*proof*⟩

end

autocorres [

ts-force nondet = *shuffle*] *in-out-parameters.c*

context *in-out-parameters-all-corres*
begin

thm *io-corres*
thm *ts-def*
term *ptr-valid*

inc′

The function `inc` increments the value which is passed as a pointer `y`.

```
void inc (unsigned* y) {
  *y = *y + 1;
}
```

When we abstract this function to have an in-out parameter instead, we obtain a function like:

```
void inc (unsigned y) {
  y = y + 1;
  return y;
}
```

Note that this is now a pure function, that does no longer depend on the heap. This is also the final output of autocorres in *inc′ ?y* ≡ *?y + 1*:

thm *inc′-def*

Let us zoom into the refinement step from L2 to IO.

thm *l2-inc′-def*
thm *io-inc′-def*

Note that the type of parameter *y* changes from a pointer type to a value

type. Here is the generated refinement theorem.

thm *io-inc′-corres*
lemma *c-guard y* $\Longrightarrow$
distinct-sets [*ptr-span y*] $\Longrightarrow$
ptr-span y $\subseteq$ *A* $\Longrightarrow$
ptr-span y $\subseteq$ *M* $\Longrightarrow$
globals.rel-alloc $\mathcal{S}$ *M A* t_0 *s t* $\Longrightarrow$
refines
(*l2-inc′ y*)
(*io-inc′* (*h-val* (*hrs-mem* (*t-hrs-′ s*)) *y*)) *s t*
(*globals.rel-stack* $\mathcal{S}$ (*ptr-span y*) *A s* t_0 (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) (*rel-singleton-stack y*)))
⟨*proof*⟩

Parameter *y* was specified as *in-out*. So function *io-inc′* becomes the actual value of the pointer passed in: *h-val* (*hrs-mem* (*t-hrs-′ s*)) *y*. To discharge the *c-guard y* guards within the body of *l2-inc′* we need the assumption that the guard holds at the beginning. As we attempt to abstract pointer *y* we have its pointer span included in set *A*. Moreover, the content of pointer *y* is modified by *l2-inc′* hence it appears in *M* as well as explicit in the final relation *globals.rel-stack*. Pointer parameters are assumed to be distinct. As we have only one parameter the assumption trivially *distinct-sets* [*ptr-span y*] holds. Note that set *A* is only constrained by *ptr-span y* $\subseteq$ *A*. So we could instantiate it with the universal set *UNIV*. Recalling the meaning of *A* this means that the resulting function *io-inc′* is a pure function, in the sense that it does not depend on the heap anymore. The relation for the termination with `exit` is *rel-exit* (λ- - -. *False*), which expresses that this path is unreachable for the current function. The function will never terminate with `exit`.

call-inc-parameter′

Next we look at a function that calls our previously defined integer function on a pointer to its parameter.

```
unsigned call_inc_parameter(unsigned n) {
  inc(&n);
  return(n);
}
```

After autocorres the function is just a mere wrapper to the increment function

thm *call-inc-parameter′-def*

Here are the relevant definitions and the IO refinement theorem.

thm *l2-call-inc-parameter′-def*
thm *io-call-inc-parameter′-def*
thm *io-call-inc-parameter′-corres* [*no-vars*]

thm *io-call-inc-parameter'-corres*
lemma *distinct-sets* ([]::*addr set list*) $\Longrightarrow$
globals.rel-alloc $\mathcal{S}$ *M A* t_0 *s t* $\Longrightarrow$
refines
(*l2-call-inc-parameter'* *n*)
(*io-call-inc-parameter'* *n*) *s t*
(*globals.rel-stack* $\mathcal{S}$ {} *A s* t_0 (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) (λ-. (=))))
⟨*proof*⟩

Note that here we do not have any assumption on the set A and the modified variables in the final state are empty. This is because we eliminate a pointer to a temporary stack variable that is out of scope after the function and in the beginning of the function it is part of the *stack-free* space.

const*keep-inc'*

Now let us consider a copy of the increment function were we do not convert to in out parameters but leave it as it is. Unsurprisingly, the resulting function still operates on pointers.

thm *keep-inc'-def*

Let us look into the IO corres phase.

thm *l2-keep-inc'-def*
thm *io-keep-inc'-def*
thm *io-keep-inc'-corres*

lemma *distinct-sets* [*ptr-span y*] $\Longrightarrow$
ptr-span y $\cap$ *A* = {} $\Longrightarrow$
ptr-span y $\cap$ *stack-free* (*hrs-htd* (*t-hrs-'* *s*)) = {} $\Longrightarrow$ — FIXME: redundant assumption?, follows from the next ones
globals.rel-alloc $\mathcal{S}$ *M A* t_0 *s t* $\Longrightarrow$
ptr-span y $\subseteq$ *M* $\Longrightarrow$
refines
(*l2-keep-inc'* *y*)
(*io-keep-inc'* *y*) *s t*
(*globals.rel-stack* $\mathcal{S}$ (*ptr-span y*) *A s* t_0 (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) (λ-. (=))))
⟨*proof*⟩

As pointer y is an allocated pointer that we do not abstract it should not belong to A and must not collide with *stack-free* space. As we might modify the pointer it is contained in M as well as the final modifies set.

Note that the bodies of *io-keep-inc'* and *l2-keep-inc'* are equivalent but the refinement statement is not a trivial statement and in particular is not the same as when skipping the IO corres phase. For example from the refinement statement we can derive that any heap location different from *ptr-span y* is unchanged.

Next let's look what happens when we call this function on a local parameter.

keep-call-inc-parameter′

thm *keep-call-inc-parameter′-def*
thm *io-keep-call-inc-parameter′-def*
thm *l2-keep-call-inc-parameter′-def*
thm *io-keep-call-inc-parameter′-corres* [*no-vars*]

lemma *distinct-sets* ([]::*addr set list*) $\Longrightarrow$
globals.rel-alloc $\mathcal{S}$ *M A* t_0 *s t* $\Longrightarrow$
refines
(*l2-keep-call-inc-parameter′ n*)
(*io-keep-call-inc-parameter′ n*) *s t*
(*globals.rel-stack* $\mathcal{S}$ {} *A s* t_0 (*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) (λ-. (=))))
⟨*proof*⟩

The local variable pointer is still present in the final function. Still the refinement statement indicates that we have a 'pure' functions, as we do not have constraints on A and also do not modify anything. The notion of pure we use here, is that the function does not depend on proper heap pointers. The *stack-free* space that we temporarily use is excluded by our definitions of *globals.frame*, *globals.rel-alloc* and *globals.rel-stack*.

safe-add′

This function is an example of an arithmetic function that checks for overflows. The status of the check is the return value and the result of the operation is passed via pointer. We make an in-out parameter for the result. So the final function returns a tuple of the result value and the status (*rel-push* in the refinement statement).

thm *safe-add′-def*
thm *l2-safe-add′-def*
thm *io-safe-add′-def*
thm *io-safe-add′-corres*

Recursion

thm *l2-fac′.simps*
thm *io-fac′.simps*
thm *io-fac′-corres*

thm *l2-even′.simps l2-odd′.simps*
thm *io-even′.simps io-odd′.simps*
thm *io-even′-corres io-odd′-corres*

swap′

We abstract a function that swaps the values referenced by two input pointers by a function that takes the values and returns the swapped values (as a pair). Note that in the refinement statement we assume disjointness of the input pointers. The swap function would still be correct if the pointers are equal. To generalise the refinement proofs to 'disjoint or equal' inputs is left as future work.

thm *swap′-def*
thm *l2-swap′-def*
thm *io-swap′-def*
thm *io-swap′-corres*

Out parameters

The function *compare* has an mere 'out' parameter. Until phase L2 the function has three parameters, after phase IO the function only has two parameters.

thm *l2-compare′-def*
thm *io-compare′-def*
thm *compare′-def*

thm *l2-call-compare′-def*
thm *io-call-compare′-def*
thm *call-compare′-def*

Pointers to compound types (structs / arrays)

Here are some examples of function that work on (parts) of a compound type.

thm *get-arr-idx′-def*
thm *l2-get-arr-idx′-def*
thm *io-get-arr-idx′-def*

thm *inc-pair′-def*
thm *l2-inc-pair′-def*
thm *io-inc-pair′-def*

Misc

thm *wa-def*
thm *ts-def*
thm *io-corres*

26.18.5 Handling `exit` in function calls

When we call a function that was refined by in-out-parameters there are various cases we have to consider, e.g. do we pass in a heap pointer or a

local pointer, do we eliminate the local pointer or keep it? In general when we pass in a pointer that is not supposed to be eliminated as an in-out parameter we first pass in the dereferenced value and after the function returns we take the value from the embellished result and assign it to the pointer. So the pointer assignment that would take place in the middle of the called function in the original program, is moved outside of the call in the refined program. After this the heaps of the original program and the refined program are in sync again at the pointer. To establish the heap refinement it turns out that we also have to bring the heaps in sync again even if the function terminates with `exit`. We catch the `exit`, update the heap and then re-throw the *exit*. So the called function also returns embellished exit values which hold the value of the relevant pointers. This extra work is a little annoying as the `exit` terminates the complete program and usually we do not catch an `exit`. So this step might introduce *L2-catch* to wrap up the procedure call. Note that this is currently the only remaining use case for *L2-catch*, the other occurrences of *L2-catch* were already transformed to *L2-try* as post processing in phase L2 when we move from 'flat' exception handling to 'nested' exception handling.

term *inc-int′*
thm *inc-int′-def*
term *call-inc-int-ptr′*
thm *call-inc-int-ptr′-def*

26.18.6 Global Heap Pointers

A core part of the IO phase is to keep track of pointers and in particular to do some bookkeeping of the parts of the heap that might get modified. This analysis and bookkeeping is focused around the pointers that are visible in the signature of a function. This fails short as soon as pointers to global heap are used within the body of a function, which are not mentioned in the function signature. To handle some common use cases with global heap we have a mechanism to over-approximate the impact of a function on the heap by referring to a static set $\mathcal{G}$ of addresses that should not be abstracted by the IO phase. We can annotate a function to make use of this mechanism by mentioning them in **init−autocorres** with the option `in_out_globals`. For the refinement statement we assume that

- $\mathcal{G} \cap A = \{\}$: pointers in $\mathcal{G}$ are not supposed to be abstracted, and
- $\mathcal{G} \subseteq M$: the function might modify any global variable.

  Within the body of such a function (marked with option `in_out_globals`), whenever a pointer is used which does not occur in the signature as pointer parameter and is specified otherwise we assume that this pointer a global pointer and assert this formally by adding a guard

ptr-span $x \subseteq \mathcal{G}$. If a function has multiple pointer parameters we also assert disjointness of pointers.

thm *io-keep-inc-global-array'-corres*
thm *io-keep-inc-global-array'-def*
thm *l2-keep-inc-global-array'-def*

thm *io-inc-global-array'-corres*
thm *io-inc-global-array'-def*
thm *l2-inc-global-array'-def*

thm *io-inc-global-array2'-corres*
thm *io-inc-global-array2'-def*
thm *l2-inc-global-array2'-def*
thm *ac-corres*

Note that a function can also have both in-out parameters and `in_out_globals`.

thm *io-read-char'-corres*
thm *io-read-char'-def*
thm *l2-read-char'-def*

thm *io-call-read-char'-corres*
thm *io-call-read-char'-def*
thm *l2-call-read-char'-def*

thm *io-call-read-char-loop'-corres*
thm *io-call-read-char-loop'-def*
thm *l2-call-read-char-loop'-def*

26.18.7 Disclaimer / Caution

As we have seen in the examples the refinement theorems can contain assumptions on the input pointers, in particular disjointness assumptions between different pointer parameters and also disjointness of pointer parameters to complete memory areas like *stack-free*. When a function is called these assumptions have to be discharged. When these pointer parameters are eliminated (replaced by in-out value parameters) the assumptions disappear. So when you work on the final output of autocorres you know nothing about pointers or disjointness.

In case a local pointer is eliminated these assumptions are be discharged locally as an intermediate step in the proof. So this use case is fine. However, if the function is called on a heap pointer these assumptions are propagated to the caller. So the assumptions move up the call stack and might end up as implicit hidden assumptions on your toplevel functions (API).

Rule of thumb: In case your toplevel API function has more then one pointer parameter don't specify any in-out parameters on that function to avoid implicit assumptions.

26.18.8 Implementation Aspects

The proof of the refinement theorem requires to keep track of various pointers and changes the signature of functions, including extending plain return values to tuples. We decided to make a rather explicit forward proof for this, to keep tight control of the various aspects. We make heavy use of ML antiquotations to match and build cterms. An obstacle here is that the state record *globals.* which holds the heap, is not yet defined but is only present as a locale. So our code is implemented within the locale *stack-heap-state*. We have introduced the concept of *interpretation data* (c.f. `../lib/ml-helpers/interpretation_data.ML`) to get hold of the interpretation we are interested in. The data is initialised in `../AutoCorres.thy` as declaration within *stack-heap-state*. In particular this allows us to match against and build heap lookup and heap update expressions.

The interface is via `In_Out_Parameters.operations`. This takes the morphism of the interpretation as an argument to derive the instance we are interested in. Note that we have crafted the main functions to benefit from the eager evaluation of ML. In particular `In_Out_Parameters.operations` takes the morphism *phi* as its only parameter and thus the instances of the main functions are only evaluated once.

end

26.18.9 Pointer-parameters as data

In a case where a pointer is not used to access the heap (e.g., *set-void*), the "modifies" part of the corres theorem should be empty. Otherwise functions like *set-byte*, that pass "arbitrarily" computed pointers fail to get IO-lifted. We enforce this by declaring pointer parameters as "data" in the *in-out-parameters* option of autocorres.

Future work: Automate this analysis to infer which pointer-parameters are treated as pure data.

context *io-corres-set-void* **begin**

lemma *distinct-sets* ([]::*addr set list*)
— should be able to get rid of this assumption, too (for multiple parameters) $\Longrightarrow$
globals.rel-alloc $\mathcal{S}$ *M A* t_0 *s t* $\Longrightarrow$
refines (*l2-set-void′ p*) (*io-set-void′ p*) *s t*
(*globals.rel-stack* $\mathcal{S}$ {} *A s* t_0
(*rel-xval-stack* (*rel-exit* (λ- - -. *False*)) (λ-. (=))))
⟨*proof*⟩
end
context *io-corres-set-void2* **begin**
thm *io-set-void2′-corres*
end
context *io-corres-set-byte* **begin**
thm *io-set-byte′-corres*

end

26.18.10 Function pointers

The basics how function pointers are handled is decribed in `fnptr.thy`.

To support function pointers during the IO phase we need the "in-out specification" for the function pointer to build the corres proposition for the function pointer and to conduct the corres proof. The specification follows those for other functions with the peculiarity that the arguments of a function pointer are specified positionally (as we do not have names for the arguments).

The general format for a function pointer p is: `p(kind, kind, ...) [might_exit] [in_out_globals])`

- where kind is on of *data*, *in*, *out*, *in-out*, *keep* or *keep∗*. The default for a non-pointer type or a function pointer type is *data*. The default for a pointer parameter is *keep∗*. The distinction between *keep∗* and *keep* allows us to fine tune the disjointness assumptions about the pointer parameters. Pointers specified with *in*, *out*, *in-out* and *keep* have to be disjoint. Pointers specified with *data* or *keep∗* do not have to be disjoint. Note that the kind of a parameter can influence the type of the resulting function and thus in which program environment the abstracted function pointer is defined. Thus it can be the case that we need multiple environments depending on the target type of the abstraction.
- The default for *might-exit* is false
- The default for *in-out-globals* is false

  There are 3 use-cases of function pointers that are supported which are handled slightly differently:

1. Function pointer as explicit parameter of another function: The specification of the pointer parameters is given as part of *in-out-parameters* specification of the outer function: `outer_function(...) \<open>for\<close> p1(...) and p2(...) ... ;` where the specs for the pointer parameter are as described above.
2. Function pointer via global variable. You do not need an explicit specification here. It is inferred from the functions that might be referenced via the global variable. Note that all functions that might be target of the same function pointer must have the same in-out specification.
3. C-style object method calls. Here we have the section *method-in-out-fun-ptr-specs* in the autocorres parameters. You specify the function pointer by

designating the structure field, `struct.field(...) ...` and giving a specification as described above. Note that currently all such method pointers with the same function signature must share the same in-out specification.

26.18.11 Future work / Open Ends

- support functions that might modify the heap typing (e.g. via *exec-concrete*) currently our *rel-alloc* / *rel-stack* setup enforces that heap typing does not change at all this is to restrictive. Before we had that heap typing does not change in $\mathcal{S}$ which was too weak to modify the A frame to handle invocations on heap pointers. I guess a good approximation would be to say that heap typing is unchanged on all relevant addresses: $\mathcal{S} \cup A \cup M$ Including M might be too restrictive, e.g. consider an alloc function that zeros some memory and adapts the heap-typing.

- Refined treatment of *in-out-globals*: allow the user to add a guard that is inserted in front of the abstract program. E.G. locale for procedure: *ptr-span* (*global-array* + *MAX-IDX*) $\subseteq \mathcal{G}$ + a Guard depending on a parameter *idx*, e.g. *idx* < *MAX-IDX*. Then we can automatically discharge local guards like *ptr-span* (*global-array* + *idx*) $\subseteq \mathcal{G}$. Not sure if this is useful though

- Cleanup unused *Generic-Data* that was replaced by interpretation data

- Remove struct-rewrite step from Heap Lifting. As we normalise pointers to root pointers in L2-Opt the pointers should already be in normal form and struct-rewrite does nothing.

- Add sanity checks to *in-out* specs, especially if parameter names actually occur in signature

- Remove redundant assumptions for keep parameters: *ptr-span y* $\cap$ *stack-free* (*hrs-htd* (*t-hrs-'* *s*)) = {} $\Longrightarrow$ *globals.rel-alloc* $\mathcal{S}$ *M A* t_0 *s t* $\Longrightarrow$ *ptr-span y* $\subseteq$ *M* $\Longrightarrow$ ...

  - The first one can be derived from the following ones

- Peephole: remove *ptr-disjoint* of singletons

- Peephole simplification, especially Guard True. Not really necessary, will disappear in hl anyways.

- Eliminate pointers to global variables: When a pointer to a global variable is only used as a *in-out parameter* it would be nice to abstract it to a record field in *lifted-globals* (or to something similar, i.e. lookup / update function in the heap with disjointnes and commutation of updates for other globals)
- Support more relaxed disjoint assumptions, especially *disjoint-or-eq*, e.g. swap also works if the input pointers are equal.
  - All 'keep' pointers are currently assumed disjoint and also are in the modifies set. We could be more relaxed here.
- HL phase has some explicit references to phase L2 instead of abstract *prev-phase* (probably related to function pointers)
- Would be nice if synthesize rules would also participate in thm, and add / del stuff like *named-rules*

end

26.19 Function Pointers

theory *fnptr*
imports *AutoCorres*
begin

install-C-file *fnptr.c*

⟨*ML*⟩

declare [[*ML-print-depth=1000*]]
init−autocorres [*phase=L1*,
ts-force nondet = *voidcaller*,
ts-force-known-functions = *option*] *fnptr.c*

autocorres [*phase=L1*,*single-threaded*,
ts-force nondet = *voidcaller*,
ts-force-known-functions = *option*] *fnptr.c*

autocorres [*phase=L2*,*single-threaded*,
ts-force nondet = *voidcaller*,
ts-force-known-functions = *option*] *fnptr.c*

autocorres [*phase=HL*,
ts-force nondet = *voidcaller*,

ts-force-known-functions = option] *fnptr.c*

autocorres [*phase=TS*,
ts-force nondet = voidcaller,
ts-force-known-functions = option] *fnptr.c*

autocorres [
ts-force nondet = voidcaller,
ts-force-known-functions = option] *fnptr.c*

Dealing with function pointers in AutoCorres has the following main challenges:

- Parameter passing
- Correspondence proofs
- Mutual recursion

term *from-bytes*

Parameter passing

In C a function pointer type does not have to specify the names of the function parameters. Hence when translating a call to a function pointer from C into SIMPL we need a uniform way to pass the function parameters. Recall that in SIMPL local variables are represented as part of the state, there is no lambda binding at that stage. For ordinary function calls we know the names of the parameters and can refer to them, for function pointers we do not know the names. Our approach is to generally switch to uniform names for the parameters, encoding the position and the type like: *in1-int*, *in2-unsigned*.

The mapping to the original names in the C file is implemented as syntactic sugar in the locale. As this mapping depends on the actual function we have to make sure to work in the correct context. The main interface to this mapping is defined in `../c-parser/HPInter.ML` e.g.:

- `HPInter.all_locvars`
- `HPInter.enter_scope`

  Note that the current implementation is a little bit fragile as it also depends on the syntax translations for lookup and update in the underlying **statespace**.

Correspondence proofs

When we come along a function pointer call in a correspondence proof we need a correspondence proof for the function pointer. Were to we get it from? In full generality we do not know much about were the function pointer might be pointing to. One drastic way out could be to add a guard at the call point, which guarantees that there is such a proof. This makes the proof trivial, but the code gets polluted with correspondence guards which also show up at the user level. So whenever one wants to proof a property about the resulting function one has to deal with this guard.

To keep the user level proofs clean, we decided to take another approach, by restricting the programs we can handle to some common use cases:

- Constant function pointer as parameter to a function. Here constant means that the parameter is not modified within the function body.
- Function pointer via constant global variable. Think of the global variable as a kind of dispatcher or class table, e.g. Array of structs representing "object descriptions" with function pointers as fields.
- C-style object method calls: Instead of a pure function pointer, a pointer to a structure which containes function pointers is passed as a parameter.

  In the first case we can add the correspondence assumption to the locale of the function. In the second case we can even resolve the correspondence proof as we can statically infer which functions might be called via the global variable.

Mutual recursion

The general approach to resolve a function pointer call is via a lookup in a program environment which maps the pointer to the definition of the function. In SIMPL the function is defined by a piece of syntax, and the program environment is an explicit context Γ which appears in the semantic rules for SIMPL. Moreover, for each function that might be called via a function pointer we introduce a pointer to that function. These pointers, and the assumption that all of them are distinct are put to the locale of global variables.

In each phase of Autocorres we introduce a similar program environment $\mathcal{P}$ that maps the pointer to a monadic HOL function definition. As from phase L2 on the parameters of functions become lambda abstractions we introduce a distinct program environment for each function type. These program environments are introduced as locale parameters. Then we successively add the implementation equations of the form $\mathcal{P}$ *some-function-pointer* = *some-function* as the functions are defined.

The correspondence theorems (or assumptions) for function pointer calls then relates the functions resolved via their respective environment, e.g. relate *the* (Γ *p*) with $\mathcal{P}$ *p*.

We also support mutual recursion via global function pointers (case 2 above), e.g. a function might call itself indirectly via a global function pointer variable. The global program analysis takes this into account to determine the strongly connected components aka. (recursive) cliques.

This adds another twist to the approach with program environments just described. The assumption $\mathcal{P}$ *some-function-pointer* = *some-recursive-function-via-*$\mathcal{P}$ can only be added after the function *some-recursive-function-via-*$\mathcal{P}$ is defined, but the definition already depends on the program environment. How do we cut the loop?

The core idea is to split the definition into two phases. First we define the **fixed-point** by explicitly extending the program environment at each call. Instead of calling $\mathcal{P}$ *p* directly we call *map-of-default* $\mathcal{P}$ [(*some-function-pointer*, *some-recursive-function-via-*$\mathcal{P}$)] *p* where *some-recursive-function-via-*$\mathcal{P}$ participates in the fixed point construction. Once the function is defined, we create a new locale, extending the program environment with the assumption $\mathcal{P}$ *some-function-pointer* = *some-recursive-function-via-*$\mathcal{P}$ and simplifying

map-of-default $\mathcal{P}$ [(*some-function-pointer*, *some-recursive-function-via-*$\mathcal{P}$)] *p* = $\mathcal{P}$ *p*

in the function body. After this extra step we again uniformly represent recursive and non-recursive function pointer calls with $\mathcal{P}$ *p*.

26.19.1 Global locales

The global locales fix the program environments (mapping function pointers to definitions) for a phase. These locales are created in the initialisation phase of **autocorres** before the individual functions are processed.

The fundamental locale is storing the addresses of global variables, including function pointers and some basic properties about them.

context *fnptr-global-addresses*
begin

For each function pointer a constant is declared e.g. *fnptr.odd-disp*, *fnptr.add*. Note that these constants have to be qualified by the program name. In case there would be a conflict with Isabelle internal names, e.g. a function ending with -- the name is suffixed with `Hoare.proc_deco`, cf. `NameGeneration.fun_ptr_name`.

⟨*ML*⟩

Moreover the global variables and their defining equations, i.e. initialisation expressions are collected in *odd-even-dispatcher* ≡ *fupdate* (*Suc 0*) (λ-. *fnptr.even-disp*) (*fupdate 0* (λ-. *fnptr.odd-disp*) (*ARRAY* -. *NULL*))

gi ≡ *0*

dispatcher-u ≡ *fupdate* (*Suc 0*) (*unop-u-C-update* (λ-. *fnptr.inc-u*)) (*fupdate* (*Suc 0*) (*binop-u-C-update* (λ-. *fnptr.add-gu*)) (*fupdate 0* (*unop-u-C-update* (λ-. *fnptr.inc-u*)) (*fupdate 0* (*binop-u-C-update* (λ-. *fnptr.add-u*)) (*ARRAY* -. *object-u-C NULL NULL*))))

dispatcher ≡ *fupdate 4* (*object-C.unop-C-update* (λ-. *NULL*)) (*fupdate 4* (*binop-C-update* (λ-. *NULL*)) (*fupdate 3* (*object-C.unop-C-update* (λ-. *fnptr.inc*)) (*fupdate 3* (*binop-C-update* (λ-. *fnptr.minus*)) (*fupdate 2* (*object-C.unop-C-update* (λ-. *fnptr.dec*)) (*fupdate 2* (*binop-C-update* (λ-. *fnptr.mul*)) (*fupdate* (*Suc 0*) (*object-C.unop-C-update* (λ-. *fnptr.dec*)) (*fupdate* (*Suc 0*) (*binop-C-update* (λ-. *fnptr.minus*)) (*fupdate 0* (*object-C.unop-C-update* (λ-. *fnptr.inc*)) (*fupdate 0* (*binop-C-update* (λ-. *fnptr.add*)) (*ARRAY* -. *object-C NULL NULL*)))))))))

add-u-p ≡ *fnptr.add-u*

add-p ≡ *fnptr.add*.

thm *global-const-defs*
thm *global-const-array-selectors* — more efficient access to array components
thm *global-const-non-array-selectors*
thm *global-const-selectors*

To express distinctness of function pointers we make use of the infrastructure in `$ISABELLE_HOME/src/HOL/Statespace/DistinctTreeProver.thy`, which is also the foundation for **statespace**. A tree data structure is used to efficiently derive distinctness and subset properties.

thm *fun-ptr-distinct*
thm *fun-ptr-subtree*

For every function pointer the locale assumes that they are not NULL, i.e. *c-fnptr-guard*. Based on the distinctness of the function pointers we derive some simplification rules for *map-of-default*.

thm *fun-ptr-simps*
end

Note that the program environment $\mathcal{P}$ for l1 programs is only declared as constant (without a definition). This environment is 'populated' in derived locales as the definitions of function become available.

A program environment for each distinct function pointer type is introduced for each further phase. e.g. $\mathcal{P}$*-l2-unsigned---unsigned*, $\mathcal{P}$*-l2-int---int*

context *fnptr-global-addresses*
begin

For example if the function pointer would index into the dispatcher array *dispatcher-u* and select a *binop-u-C*, it would result into subgoals for each of the possible pointers, namely and *fnptr.add-u fnptr.add-gu*.

term *dispatcher-u*

In this bundle we define some introduction rules to support the correspondence proofs. In phase L1 and L2 the correspondence proof is quite

explicitly done in ML and the rules for function pointer calls are explicitly instantiated in ML. Here in layer HL the proof is done by applying intro rules and synthesising the abstract version of the program. The following rules relate the abstract and concrete program environments that are used for function pointer calls.

context includes *fnptr-hl-impl-corres*
begin
thm *fun-ptr-intros*
end

Note for example, that in the current goal state the concrete program would be of the shape $\mathcal{P}$*-l2----int p* whereas the abstract program would be a plain schematic variable. This variable is then instantiated with $\mathcal{P}$*-hl----int p*. Once the rule is applied the identity marker *DYN-CALL* triggers the side-condition splitter / solver: `AutoCorresUtil.dyn_call_split_simp_sidecondition_tac`, which splits the generic correspondence of some generic function pointer to its possible values.

For example if the function pointer would index into the dispatcher array *dispatcher-u* and select a *binop-u-C*, it would result into subgoals for each of the possible pointers, namely *fnptr.add-u* and *fnptr.add-gu*.

context includes *fnptr-wa-impl-corres*
begin

Like in phase HL we also have some custom intro rules to support the instantiation of function pointer calls.

thm *fun-ptr-intros*
thm *global-const-defs*
thm *fun-ptr-simps*
end

The TS phase adds another dimension to the program environments, the monad type. E.g. $\mathcal{P}$*-pure-unsigned---unsigned*, $\mathcal{P}$*-gets-unsigned---unsigned*, $\mathcal{P}$*-option-unsigned---unsigned*, $\mathcal{P}$*-nondet-unsigned---unsigned*, $\mathcal{P}$*-exit-unsigned---unsigned*.

context includes *fnptr-ts-impl-corres*
begin

Again we have custom intro rules to support instantiation of the abstract program environment.

thm *fun-ptr-intros*
thm *global-const-defs*
thm *fun-ptr-simps*
end
end

26.19.2 Function / Recursive-clique specific locales

L1

context *l1-definition-add* — holds the definition of the function
begin
thm *l1-def* — all relevant l1 definitions
thm *ac-def* — definitions of all layers

Note that there are two variants of the definition of *l1-add′*. First the original definition and then an optimised version, already removing some exception handling. Here are the names of the theorems:

thm *l1-add′-def*
thm *l1-opt-add′-def*
end

context *l1-impl-add*
begin

As *l1-add′* might also be called indirectly via a function pointer, we populate the program environment with the definition

thm *l1-add′-impl* — Mapping of the function-pointer to the definition in the corresponding environment.

The equation is also added to *gets-the* (*ogets* (λ-. *?f*)) = *return ?f*
gets-the (*ogets ?f*) = *gets ?f*
c-fnptr-guard fnptr.f
c-fnptr-guard fnptr.add
c-fnptr-guard fnptr.dec
c-fnptr-guard fnptr.inc
c-fnptr-guard fnptr.mul
c-fnptr-guard fnptr.add-u
c-fnptr-guard fnptr.inc-u
c-fnptr-guard fnptr.minus
c-fnptr-guard fnptr.add-gu
c-fnptr-guard fnptr.odd-disp
c-fnptr-guard fnptr.callable1
c-fnptr-guard fnptr.even-disp
c-fnptr-guard fnptr.intcallable2
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.odd-disp* = *?f1.0*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.even-disp* = *?f2.0*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.intcallable2* = *?d fnptr.intcallable2*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.callable1* = *?d fnptr.callable1*

map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.add-gu* = *?d fnptr.add-gu*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.minus* = *?d fnptr.minus*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.add-u* = *?d fnptr.add-u*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.inc* = *?d fnptr.inc*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.mul* = *?d fnptr.mul*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.add* = *?d fnptr.add*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.f* = *?d fnptr.f*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.dec* = *?d fnptr.dec*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.inc-u* = *?d fnptr.inc-u*
$\mathcal{P}$ *fnptr.add* = *l1-add'*.

thm *fun-ptr-simps*
thm *l1-def*
end

context *l1-definition-call-binop*
begin

l1-call-binop' makes a function pointer call via the *dispatcher* array selecting field *binop-C*. So it might call *l1-add'*, *l1-minus'* or *l1-mul'*. The definitions of these functions are also imported into the locale (cf. *l1-add'* ≡ *L1-catch* (*L1-seq* (*L1-init* (λ*upd*. *add.ret'*:=$_{\mathcal{L}}$ *upd*)) (*L1-seq* (*L1-modify* *add.k*:=$_{\mathcal{L}}$ (λ-. *gi*)) (*L1-seq* (*L1-guard* (λ*s*. − *2147483648* ≤ *sint* (*s* ·$_{\mathcal{L}}$ *add.n*) + *sint* (*s* ·$_{\mathcal{L}}$ *add.m*) ∧ *sint* (*s* ·$_{\mathcal{L}}$ *add.n*) + *sint* (*s* ·$_{\mathcal{L}}$ *add.m*) ≤ *2147483647*)) (*L1-seq* (*L1-modify* (λ*s*. *s*⟨*add.ret'* :=$_{\mathcal{L}}$ λ-. (*s* ·$_{\mathcal{L}}$ *add.n*) + (*s* ·$_{\mathcal{L}}$ *add.m*)⟩)) (*L1-seq* (*L1-modify* (*global-exn-var'-'-update* (λ-. *Return*))) *L1-throw*))))) (*L1-condition* (λ*s*. *is-local* (*global-exn-var'-'* *s*)) *L1-skip* *L1-throw*)

l1-add' ≡ *L1-seq* (*L1-init* (λ*upd*. *add.ret'*:=$_{\mathcal{L}}$ *upd*)) (*L1-seq* (*L1-modify* *add.k*:=$_{\mathcal{L}}$ (λ-. *gi*)) (*L1-seq* (*L1-guard* (λ*s*. − *2147483648* ≤ *sint* (*s* ·$_{\mathcal{L}}$ *add.n*) + *sint* (*s* ·$_{\mathcal{L}}$ *add.m*) ∧ *sint* (*s* ·$_{\mathcal{L}}$ *add.n*) + *sint* (*s* ·$_{\mathcal{L}}$ *add.m*) ≤ *2147483647*)) (*L1-seq* (*L1-modify* (λ*s*. *s*⟨*add.ret'* :=$_{\mathcal{L}}$ λ-. (*s* ·$_{\mathcal{L}}$ *add.n*) + (*s* ·$_{\mathcal{L}}$ *add.m*)⟩)) (*L1-modify* (*global-exn-var'-'-update* (λ-. *Return*))))))

l1-mul' ≡ *L1-catch* (*L1-seq* (*L1-init* (λ*upd*. *mul.ret'*:=$_{\mathcal{L}}$ *upd*)) (*L1-seq* (*L1-guard* (λ*s*. − *2147483648* ≤ *sint* (*s* ·$_{\mathcal{L}}$ *mul.n*) ∗ *sint* (*s* ·$_{\mathcal{L}}$ *mul.m*) ∧ *sint* (*s* ·$_{\mathcal{L}}$ *mul.n*) ∗ *sint* (*s* ·$_{\mathcal{L}}$ *mul.m*) ≤ *2147483647*)) (*L1-seq* (*L1-modify* (λ*s*. *s*⟨*mul.ret'* :=$_{\mathcal{L}}$ λ-. (*s* ·$_{\mathcal{L}}$ *mul.n*) ∗ (*s* ·$_{\mathcal{L}}$ *mul.m*)⟩)) (*L1-seq* (*L1-modify* (*global-exn-var'-'-update* (λ-. *Return*))) *L1-throw*)))) (*L1-condition* (λ*s*. *is-local*

(*global-exn-var'-'* *s*)) *L1-skip L1-throw*)

l1-mul' ≡ *L1-seq* (*L1-init* (λ*upd*. *mul.ret'*:=$_{\mathcal{L}}$ *upd*)) (*L1-seq* (*L1-guard* (λ*s*. − *2147483648* ≤ *sint* (*s* ·$_{\mathcal{L}}$ *mul.n*) ∗ *sint* (*s* ·$_{\mathcal{L}}$ *mul.m*) ∧ *sint* (*s* ·$_{\mathcal{L}}$ *mul.n*) ∗ *sint* (*s* ·$_{\mathcal{L}}$ *mul.m*) ≤ *2147483647*)) (*L1-seq* (*L1-modify* (λ*s*. *s*⟨*mul.ret'* :=$_{\mathcal{L}}$ λ-. (*s* ·$_{\mathcal{L}}$ *mul.n*) ∗ (*s* ·$_{\mathcal{L}}$ *mul.m*)⟩)) (*L1-modify* (*global-exn-var'-'-update* (λ-. *Return*)))))

l1-minus' ≡ *L1-catch* (*L1-seq* (*L1-init* (λ*upd*. *ret'*:=$_{\mathcal{L}}$ *upd*)) (*L1-seq* (*L1-guard* (λ*s*. − *2147483648* ≤ *sint* (*s* ·$_{\mathcal{L}}$ *n*) − *sint* (*s* ·$_{\mathcal{L}}$ *m*) ∧ *sint* (*s* ·$_{\mathcal{L}}$ *n*) − *sint* (*s* ·$_{\mathcal{L}}$ *m*) ≤ *2147483647*)) (*L1-seq* (*L1-modify* (λ*s*. *s*⟨*ret'* :=$_{\mathcal{L}}$ λ-. (*s* ·$_{\mathcal{L}}$ *n*) − (*s* ·$_{\mathcal{L}}$ *m*)⟩)) (*L1-seq* (*L1-modify* (*global-exn-var'-'-update* (λ-. *Return*))) *L1-throw*)))) (*L1-condition* (λ*s*. *is-local* (*global-exn-var'-'* *s*)) *L1-skip* *L1-throw*)

l1-minus' ≡ *L1-seq* (*L1-init* (λ*upd*. *ret'*:=$_{\mathcal{L}}$ *upd*)) (*L1-seq* (*L1-guard* (λ*s*. − *2147483648* ≤ *sint* (*s* ·$_{\mathcal{L}}$ *n*) − *sint* (*s* ·$_{\mathcal{L}}$ *m*) ∧ *sint* (*s* ·$_{\mathcal{L}}$ *n*) − *sint* (*s* ·$_{\mathcal{L}}$ *m*) ≤ *2147483647*)) (*L1-seq* (*L1-modify* (λ*s*. *s*⟨*ret'* :=$_{\mathcal{L}}$ λ-. (*s* ·$_{\mathcal{L}}$ *n*) − (*s* ·$_{\mathcal{L}}$ *m*)⟩)) (*L1-modify* (*global-exn-var'-'-update* (λ-. *Return*)))))

l1-call-binop' ≡ *L1-catch* (*L1-seq* (*L1-init* (λ*upd*. *call-binop.ret'*:=$_{\mathcal{L}}$ *upd*)) (*L1-seq* (*L1-modify* *call-binop.r*:=$_{\mathcal{L}}$ (λ-. *0*)) (*L1-seq* (*L1-guard* (λ*s*. *UCAST*(*8* → *32 signed*) (*s* ·$_{\mathcal{L}}$ *call-binop.i*) <*s* *5*)) (*L1-seq* (*L1-guard* (λ*s*. *0* ≤*s* *UCAST*(*8* → *32 signed*) (*s* ·$_{\mathcal{L}}$ *call-binop.i*))) (*L1-seq* (*L1-condition* (λ*s*. *PTR-COERCE*(*unit* → *unit*) (*binop-C* (*dispatcher.*[*unat* (*s* ·$_{\mathcal{L}}$ *call-binop.i*)])) ≠ *NULL*) (*L1-guarded* (λ*s*. *0* ≤*s* *UCAST*(*8* → *32 signed*) (*s* ·$_{\mathcal{L}}$ *call-binop.i*) ∧ *UCAST*(*8* → *32 signed*) (*s* ·$_{\mathcal{L}}$ *call-binop.i*) <*s* *5* ∧ *c-fnptr-guard* (*binop-C* (*dispatcher.*[*unat* (*s* ·$_{\mathcal{L}}$ *call-binop.i*)]))) (*do* {
p ← *gets* (λ*s*. *binop-C* (*dispatcher.*[*unat* (*s* ·$_{\mathcal{L}}$ *call-binop.i*)]));
L1-call (λ*s*. *locals-update* (*cupdate* (*Suc 0*) (λ-. *s* ·$_{\mathcal{L}}$ *call-binop.m*)) (*locals-update* (*cupdate 0* (λ-. *s* ·$_{\mathcal{L}}$ *call-binop.n*)) *s*)) (*P* *p*) (λ*s t*. *s*(|*globals* := *globals t*|)) (λ*s t*. *s*(|*global-exn-var'-'* := *Nonlocal* (*the-Nonlocal* (*global-exn-var'-'* *t*))|)) (λ*uu*-. *call-binop.r*:=$_{\mathcal{L}}$ (λ-. *clookup 2* (*locals uu*-)))
})) *L1-skip*) (*L1-seq* (*L1-modify* (λ*s*. *s*⟨*call-binop.ret'* :=$_{\mathcal{L}}$ λ-. *s* ·$_{\mathcal{L}}$ *call-binop.r*⟩)) (*L1-seq* (*L1-modify* (*global-exn-var'-'-update* (λ-. *Return*))) *L1-throw*))))))) (*L1-condition* (λ*s*. *is-local* (*global-exn-var'-'* *s*)) *L1-skip* *L1-throw*)

l1-call-binop' ≡ *L1-seq* (*L1-init* (λ*upd*. *call-binop.ret'*:=$_{\mathcal{L}}$ *upd*)) (*L1-seq* (*L1-modify* *call-binop.r*:=$_{\mathcal{L}}$ (λ-. *0*)) (*L1-seq* (*L1-guard* (λ*s*. *UCAST*(*8* → *32 signed*) (*s* ·$_{\mathcal{L}}$ *call-binop.i*) <*s* *5*)) (*L1-seq* (*L1-guard* (λ*s*. *0* ≤*s* *UCAST*(*8* → *32 signed*) (*s* ·$_{\mathcal{L}}$ *call-binop.i*))) (*L1-catch* (*L1-seq* (*L1-condition* (λ*s*. *PTR-COERCE*(*unit* → *unit*) (*binop-C* (*dispatcher.*[*unat* (*s* ·$_{\mathcal{L}}$ *call-binop.i*)])) ≠ *NULL*) (*L1-guarded* (λ*s*. *0* ≤*s* *UCAST*(*8* → *32 signed*) (*s* ·$_{\mathcal{L}}$ *call-binop.i*) ∧ *UCAST*(*8* → *32 signed*) (*s* ·$_{\mathcal{L}}$ *call-binop.i*) <*s* *5* ∧ *c-fnptr-guard* (*binop-C* (*dispatcher.*[*unat* (*s* ·$_{\mathcal{L}}$ *call-binop.i*)]))) (*do* {
p ← *gets* (λ*s*. *binop-C* (*dispatcher.*[*unat* (*s* ·$_{\mathcal{L}}$ *call-binop.i*)]));
L1-call (λ*s*. *locals-update* (*cupdate* (*Suc 0*) (λ-. *s* ·$_{\mathcal{L}}$ *call-binop.m*)) (*locals-update* (*cupdate 0* (λ-. *s* ·$_{\mathcal{L}}$ *call-binop.n*)) *s*)) (*P* *p*) (λ*s t*. *s*(|*globals* :=

globals t⦈) (λ*s t*. *s*(⦇*global-exn-var*′-′ := *Nonlocal* (*the-Nonlocal* (*global-exn-var*′-′ *t*))⦈) (λ*uu*-. *call-binop.r*:=$_{\mathcal{L}}$ (λ-. *clookup 2* (*locals uu*-)))
})) *L1-skip*) (*L1-seq* (*L1-modify* (λ*s*. *s*⟨*call-binop.ret*′ :=$_{\mathcal{L}}$ λ-. *s* ·$_{\mathcal{L}}$ *call-binop.r*⟩)) (*L1-seq* (*L1-modify* (*global-exn-var*′-′*-update* (λ-. *Return*))) *L1-throw*))) (*L1-condition* (λ*s*. *is-local* (*global-exn-var*′-′ *s*)) *L1-skip* *L1-throw*))))))), as well as the corresponding entries in the program environment (cf. *gets-the* (*ogets* (λ-. *?f*)) = *return ?f*

gets-the (*ogets ?f*) = *gets ?f*
c-fnptr-guard fnptr.f
c-fnptr-guard fnptr.add
c-fnptr-guard fnptr.dec
c-fnptr-guard fnptr.inc
c-fnptr-guard fnptr.mul
c-fnptr-guard fnptr.add-u
c-fnptr-guard fnptr.inc-u
c-fnptr-guard fnptr.minus
c-fnptr-guard fnptr.add-gu
c-fnptr-guard fnptr.odd-disp
c-fnptr-guard fnptr.callable1
c-fnptr-guard fnptr.even-disp
c-fnptr-guard fnptr.intcallable2
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.odd-disp* = *?f1.0*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.even-disp* = *?f2.0*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.intcallable2* = *?d fnptr.intcallable2*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.callable1* = *?d fnptr.callable1*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.add-gu* = *?d fnptr.add-gu*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.minus* = *?d fnptr.minus*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.add-u* = *?d fnptr.add-u*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.inc* = *?d fnptr.inc*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.mul* = *?d fnptr.mul*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.add* = *?d fnptr.add*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.f* = *?d fnptr.f*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.dec*

$=$ *?d fnptr.dec*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.inc-u*
$=$ *?d fnptr.inc-u*
$\mathcal{P}$ *fnptr.add* $=$ *l1-add′*
$\mathcal{P}$ *fnptr.mul* $=$ *l1-mul′*
$\mathcal{P}$ *fnptr.minus* $=$ *l1-minus′*). Also note that the function pointer call is wrapped in a *L1-guarded*. In particular the guard ensures that the index into the array is in range. It is essential to have this information available in the correspondence proof to split the function pointer call to its potential targets.

thm *l1-call-binop′-def*
thm *global-const-defs*
thm *l1-def*
thm *fun-ptr-simps*
end

context *l1-corres-call-binop*
begin

The correspondence proofs are performed within the corres locales. The results are added to *L1corres True* Γ *l1-add′* (*Call fnptr.add*)
L1corres True Γ *l1-mul′* (*Call fnptr.mul*)
L1corres True Γ *l1-minus′* (*Call fnptr.minus*)
L1corres True Γ *l1-call-binop′* (*Call fnptr.call-binop*). Note that the correspondence proofs of the potential callees are present. **autocorres** resolves call dependencies and first performs the correspondence proofs of the potential callees.

thm *l1-call-binop′-corres*
thm *l1-corres*
thm *l1-def*
end

context *l1-definition-parameter-call*
begin

This function performs a function pointer call on a function parameter. Recall that we only support the case were the function pointer parameter value stays the same within the function. The phase L1 is special, as we postpone the correspondence proof of the function pointer call to the L2 layer. The reason is that in the L2 layer we replace the parameters with lambda abstraction and thus the fact that the value does not change becomes trivial. In L1 we would have to propagate this information trough every state update, from the invocation of the function to the function pointer call. This proof is anyway performed in the L2 phase. Postponing the correspondence proof is achieved by just assuming the correspondence in *L1-guarded*.

thm *l1-parameter-call′-def*

thm *l1-def*
end

context *l1-corres-parameter-call*
begin
thm *l1-parameter-call′-corres*
thm *l1-corres*
end

L2

context *l2-definition-add*
begin
thm *fun-ptr-simps*
end

context *l2-impl-add*
begin
thm *fun-ptr-simps*
end

context *l2-definition-call-binop*
begin
thm *fun-ptr-simps*
thm *l2-call-binop′-def*
thm *l2-def*
end

context *l2-corres-call-binop*
begin
thm *l2-call-binop′-corres*
thm *l2-corres*
thm *l2-def*
end

context *l2-definition-parameter-call*
begin

Note that in contrast to L1 we do not have any correspondence assumption in *L2-guarded*. The function pointer parameter is just inserted in the function pointer call and thus is immediately the same value.

thm *l2-parameter-call′-def*
thm *l2-def*
end

context *l2-corres-parameter-call*
begin

In contrast to the function pointer call via a global variable, where we statically resolve which functions are potentially called, we add an explicit

assumption on the parameter to the correspondence theorem. As in phase L2 we also have to proof the postponed L1 correspondence we actually have both assumptions here.

thm *l2-parameter-call'-corres*
thm *l2-corres*
thm *l2-def*
end

When defining (mutual) recursion indirectly via function pointers, there is a subtlety in the definition of the functions. In order to have an admissible *L2corres* property the program environment for the current clique is temporarily explicitly exended via *map-of-default* in the function bodies. After definition of the function and extending the hypothetical program environment with the new definitions this construction is hidden again.

Compare the variation of in *l2-corres-odd-disp-even-disp* vs. *l2-impl-odd-disp-even-disp* and the extended program environment in *gets-the* (*ogets* (λ-. *?f*)) = *return ?f*

gets-the (*ogets ?f*) = *gets ?f*.

Also recall the general flow of how to arrive at a new level with definition and correspondence proof.

- First the induction step of the correspondence proof is performed, assuming that the recursive calls are in the correspondence relation.
- If that proof is successful the function body or bodies are used to do the actual definition(s) of the function(s).
- After the definition is done the actual correspondence proof is performed using the induction step as major ingredient, replacing the body / bodies with the new definition(s).
- The program environments are extended with the assumptions that the pointers are mapped to the new definitions and the right hand sides of the definitions are simplified.

context *l2-corres-odd-disp-even-disp*
begin
thm *fun-ptr-simps*
thm *l2-corres*
thm *l2-def*
thm *l2-odd-disp'-def* — Foundational definition of **fixed-point**. FIXME: should we conceale this?
thm *l2-odd-disp'.simps*
end

context *l2-impl-odd-disp-even-disp*
begin

thm *fun-ptr-simps*
thm *l2-corres*
thm *l2-def*
thm *l2-odd-disp′.simps*
thm *l2-impl-odd-disp′-def* — canonical variant. FIXME: should we rename this (simps?)?
end

HL

context *hl-definition-call-binop*
begin
thm *fun-ptr-simps*
thm *hl-call-binop′-def*
thm *hl-def*
end

context *hl-corres-call-binop*
begin
thm *hl-call-binop′-corres*
thm *hl-corres*
thm *hl-def*
end

context *hl-definition-parameter-call*
begin
thm *hl-parameter-call′-def*
thm *hl-def*
end

context *hl-corres-parameter-call*
begin
thm *hl-parameter-call′-corres*
thm *hl-corres*
thm *hl-def*
end

WA

context *wa-definition-call-binop*
begin
thm *fun-ptr-simps*
thm *wa-call-binop′-def*
thm *wa-def*
end

context *wa-corres-call-binop*
begin
thm *wa-call-binop′-corres*
thm *wa-corres*

thm *wa-def*
end

context *wa-definition-parameter-call*
begin
thm *wa-parameter-call'-def*
thm *wa-def*
end

context *wa-corres-parameter-call*
begin
thm *wa-parameter-call'-corres*
thm *wa-corres*
thm *wa-def*
end

TS

context *ts-impl-add*
begin

add' is defined within the option monad. Note that (lifted version) also end up in the more expressive program environments (cf. *gets-the* (*ogets* (λ-. *?f*)) = *return ?f*

gets-the (*ogets ?f*) = *gets ?f*
c-fnptr-guard fnptr.f
c-fnptr-guard fnptr.add
c-fnptr-guard fnptr.dec
c-fnptr-guard fnptr.inc
c-fnptr-guard fnptr.mul
c-fnptr-guard fnptr.add-u
c-fnptr-guard fnptr.inc-u
c-fnptr-guard fnptr.minus
c-fnptr-guard fnptr.add-gu
c-fnptr-guard fnptr.odd-disp
c-fnptr-guard fnptr.callable1
c-fnptr-guard fnptr.even-disp
c-fnptr-guard fnptr.intcallable2
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.odd-disp* = *?f1.0*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.even-disp* = *?f2.0*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.intcallable2* = *?d fnptr.intcallable2*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.callable1* = *?d fnptr.callable1*
map-of-default ?d [(*fnptr.odd-disp*, *?f1.0*), (*fnptr.even-disp*, *?f2.0*)] *fnptr.add-gu*

$= ?d\ fnptr.add\text{-}gu$
$\quad map\text{-}of\text{-}default\ ?d\ [(fnptr.odd\text{-}disp,\ ?f1.0),\ (fnptr.even\text{-}disp,\ ?f2.0)]\ fnptr.minus$
$= ?d\ fnptr.minus$
$\quad map\text{-}of\text{-}default\ ?d\ [(fnptr.odd\text{-}disp,\ ?f1.0),\ (fnptr.even\text{-}disp,\ ?f2.0)]\ fnptr.add\text{-}u$
$= ?d\ fnptr.add\text{-}u$
$\quad map\text{-}of\text{-}default\ ?d\ [(fnptr.odd\text{-}disp,\ ?f1.0),\ (fnptr.even\text{-}disp,\ ?f2.0)]\ fnptr.inc$
$= ?d\ fnptr.inc$
$\quad map\text{-}of\text{-}default\ ?d\ [(fnptr.odd\text{-}disp,\ ?f1.0),\ (fnptr.even\text{-}disp,\ ?f2.0)]\ fnptr.mul$
$= ?d\ fnptr.mul$
$\quad map\text{-}of\text{-}default\ ?d\ [(fnptr.odd\text{-}disp,\ ?f1.0),\ (fnptr.even\text{-}disp,\ ?f2.0)]\ fnptr.add$
$= ?d\ fnptr.add$
$\quad map\text{-}of\text{-}default\ ?d\ [(fnptr.odd\text{-}disp,\ ?f1.0),\ (fnptr.even\text{-}disp,\ ?f2.0)]\ fnptr.f$
$= ?d\ fnptr.f$
$\quad map\text{-}of\text{-}default\ ?d\ [(fnptr.odd\text{-}disp,\ ?f1.0),\ (fnptr.even\text{-}disp,\ ?f2.0)]\ fnptr.dec$
$= ?d\ fnptr.dec$
$\quad map\text{-}of\text{-}default\ ?d\ [(fnptr.odd\text{-}disp,\ ?f1.0),\ (fnptr.even\text{-}disp,\ ?f2.0)]\ fnptr.inc\text{-}u$
$= ?d\ fnptr.inc\text{-}u$
$\quad \mathcal{P}\text{-}option\text{-}int'int\text{---}int\ fnptr.add = add'$
$\quad \mathcal{P}\text{-}nondet\text{-}int'int\text{---}int\ fnptr.add = (\lambda x1\ x2.\ gets\text{-}the\ (add'\ x1\ x2))$
$\quad \mathcal{P}\text{-}exit\text{-}int'int\text{---}int\ fnptr.add = (\lambda x1\ x2.\ liftE\ (gets\text{-}the\ (add'\ x1\ x2))))$.

So function pointer calls to this function can be performed in any of those monads.

thm *add′-def*
thm *fun-ptr-simps*
thm *ts-def*
end

context *ts-definition-call-binop*
begin
thm *fun-ptr-simps*
thm *call-binop′-def*
thm *ts-def*
end

context *ts-corres-call-binop*
begin
thm *call-binop′-corres*
thm *ts-corres*
thm *ts-def*
end

context *ts-definition-fac*
begin
thm *fun-ptr-simps*
thm *fac′.simps*
thm *ts-def*

end

context *ts-corres-fac*
begin
thm *fac′-corres*
thm *ts-corres*
thm *ts-def*
end

Recall that in case of a function pointer call via a parameter we assume correspondence of the parameter. So in which monad should we end up? Which monad should we assume for the parameter? Currently we assume the same monad as for the function. We consecutively try the correspondence from the most restrictive to the most expressive monad and we end up in the monad with the first successful proof. If you want a different (more expressive) monad you can configure this via the autocorres option *ts-force*.

context *ts-definition-parameter-call*
begin
thm *parameter-call′-def*
thm *wa-def*
end

context *ts-corres-parameter-call*
begin
thm *parameter-call′-corres*
thm *wa-corres*
thm *wa-def*
end

context *ts-impl-call-binop*
begin
thm *ts-def*
end

Final AutoCorres

Once all phases are passed and the correspondence proofs between the consecutive layers are ready they are combined to the final correspondence layer, from SIMPL upto TS.

context *corres-parameter-call*
begin
thm *parameter-call′-ac-corres*
end

context *corres-call-binop*
begin
thm *call-binop′-ac-corres*
end

context *corres-odd-disp-even-disp*
begin
thm *odd-disp′-ac-corres*
thm *even-disp′-ac-corres*
end

When all functions of a C file are translated the following summary locales are introduced.

context *fnptr-all-impl* — All implementation locales of phase TS
begin
thm *fun-ptr-simps*
thm *even-disp′.simps*
thm *impl-even-disp′-def*
thm *ac-def*
end

context *fnptr-all-corres* — All corres locales of all phases
begin
thm *l1-corres*
thm *l2-corres*
thm *hl-corres*
thm *wa-corres*
thm *ts-corres*
thm *ac-corres*

thm *l1-def*
thm *l2-def*
thm *l2-even-disp′-def*
thm *l2-even-disp′.simps*
thm *hl-def*
thm *wa-def*
thm *ts-def*
thm *ac-def*

thm *global-const-defs*
thm *fun-ptr-simps*
thm *fun-ptr-intros*
thm *fun-ptr-distinct*
thm *fun-ptr-subtree*
end

Method calls of C-style objects

The goal is to also support function pointer invocations via C-style object methods: Instead of a plain function pointer a function gets a pointer to a structure (object) that contains function pointers as structure fields. This adds an important twist to the refinement proofs of autocorres. When we invoke a method via `p->method(...)` we actually calculate the function

pointer from the heap object referenced by pointer `p`. As with the cases described before at that point we have to argue that the referenced function fulfills the 'corres' predicate. However, now we have the indirection via the heap. So somehow we have to keep track of the function pointers stored in the heap. As other pointer objects might also be stored and manipulated in the heap this typically requires user-level reasoning about the heap layout and disjointness of certain pointers. As autocorres is mainly used as a preprocessing tool for arbitrary C programs we want to avoid making too restrictive assumptions here. Instead we postpone the argument to the user. The core idea is that we want to argue that the actually referenced function is a 'known function' for which we know or can assume that the 'corres' predicate holds. We introduce a predicate *known-function* into our locale hierarchy, which takes a function pointer p and tells us if this is a known function for which we can assume 'corres'. On every invocation of a object method we introduce a guard *known-function* (*method p*) into the code. Autocorres can then utilise this guard together with the assumption that the correspondence holds for all known functions. It is then left as an exercise for the user to discharge the guards. This boils down to an invariant argument that every time you assign something to *method p* it has to be a *known-function*. To support this, everytime a function is defined which might be such a function pointer, we utilise the implementation locales to store the information that this is a *known-function*.

Here are the locales about the *known-function* which state correspondence. These are imported as parents in all individual corres locales.

context *fnptr-l1-corres*
begin
thm *known-function-corres*
end

context *fnptr-l2-corres*
begin
thm *known-function-corres*
end

context *fnptr-hl-corres*
begin
thm *known-function-corres*
end

The locales for phases WA and TS are special as they might change the signature of the resulting function depending on the called functions. We make assumptions about called methods to derive the definition of the calling function. E.g. for phase WA we have to know whether a called method will do signed or unsigned word abstractions. For phase TS we have to know in which monad the called methods are defined in order

to select the right monad for the caller. We can use the autocorres options `ts_force_known_functions`, `unsigned_word_abs_known_functions` and `no_signed_word_abs_known_functions` analogous to the corresponding options for individual functions.

In our example program we have chosen the option monad as a result monad. Note that autocorres does some sanity checks to ensure that e.g. the chosen monad is expressible enough, but ultimately it is the responsibility of the user to ensure that the monad and the word abstraction options match all potential instances of the methods.

context *fnptr-wa-corres*
begin
thm *known-function-corres*
end

context *fnptr-ts-corres*
begin
thm *known-function-corres*
end

context *ts-definition-call-object-binop-return*
begin
thm *ts-def*
end

context *ts-definition-call-object-binop-assign*
begin
thm *ts-def*
end

context *ts-definition-call-object-binop-emb*
begin
thm *ts-def*
end

Here you can see all known functions that autocorres has detected. Note that for this analysis autocorres assumes that it has total knowledge of the complete program. It collects all functions for which addresses are calculated in the program and considers them as *known-function*.

context *fnptr-all-impl*
begin
thm *known-function*
thm *fun-ptr-simps*
thm *fun-ptr-distinct*
end

context *fnptr-all-corres*
begin

thm *global-const-defs*
thm *fun-ptr-simps*
thm *fun-ptr-intros*
thm *fun-ptr-distinct*
thm *fun-ptr-subtree*
thm *known-function*
thm *known-function-corres*
end

end

26.20 Unions

theory *union-ac*
imports
AutoCorres

begin

26.20.1 Union support in C-Parser and Autocorres

Fully fledged C unions can have a lot of semantic subtleties in particular with respect to undefined behavior when it comes to different padding in the variants, effective types and accessing the union via different variants, e.g. writing a pointer through one variant and subsequently reading or writing the same union via another variant. See for example https://robbertkrebbers.nl/research/thesis.pdf in subsections 2.5.5 ff. for a discussion of some fine points.

Our current solution is very pragmatic by only supporting a subset of unions that does not exhibit the more "weired" behaviours:

- Single variant unions
- Multi-variant unions where each variant is packed (no implicit padding) and has the same size.

  For these limited unions the semantics is deterministic and basically boils down to casting / coercing different C-types. Note that every *c-type* is equipped with conversions *from-bytes* and *to-bytes* which allows us to take a value, convert it to a byte list and then make another value (potentially with a different type) from that byte list again. So the basic machinery is already there.

  The C parser analyses which variants of a union are actually actively used in the code you want to verify. Only those variants (internally referred to as active variants) are taken into account. If there is only a single variant we are fine and a single record type is constructed in

HOL as the C-type representing that type. It is the same construction as for a structure type of that variant.

In case there are multiple active variants the C parser checks that all those variants have the same size and are *packed* which means that there is no padding. The original C code might have to be rewritten with explicit padding fields to fulfill this requirement.

We create a record type for each variant of the union, like a 'struct' for each variant. One variant is considered as the *canonical* variant. Currently this always is the first variant of the union. The idea of the canonical variant is that every access or update to a union is always done via this canonical variant and then 'casted' to the variant you need. This casting is implemented via *PTR-COERCE*($'a \rightarrow 'b$) for pointers and *COERCE*($'a \rightarrow 'b$) and *coerce-map* for values.

Example:

```
typedef struct {
  unsigned fst;
  unsigned snd;
 } unsigned_pair;

typedef struct {
  signed fst;
  signed snd;
 } signed_pair;



typedef union {
  unsigned_pair u;
  signed_pair s;
} open_union;
```

The union *open-union* has two variants. The first one u is considered to be the canonical one. So when you want to update a variant s in the heap via a pointer to *open-union* we first read the canonical variant u from the heap then coerce the value to a variant s, then the update on the value is performed then the resulting value is coerced back to variant u and finally written back to the heap.

A good mental model for this implementation is that accesses to unions are normalised towards the canonical variants of the root pointers. This also carries over to the split heap model you obtain when you perform the HL phase of autocorres. There is only one relevant heap for the union, which is the heap for the canonical variant. This means that you can also define *addressable-fields* for the canonical variant. Currently you cannot define *addressable-fields* for the other variants.

include-C-file *union.h* **for** *union.c*
install-C-file *union.c*
autocorres [*addressable-fields* = *open-union.u*]*union.c*

context *union-all-corres*
begin
thm *ts-def*
thm *l2-def*

Names

The union *open-union* has two variants *u* and *s*. The first one, *u*, is considered to be the canonical one. The corresponding c-type becomes "THE" type of union and is named *open-union-C*. The other variant *s* gets the qualified name *open-union-C's-C*.

term *open-union-C*
term *u-C*
thm *u-C-def*
term *u-C-update*
thm *u-C-update-def*

term *open-union-C's-C*
term *s-C*
thm *s-C-def*
term *s-C-update*
thm *s-C-update-def*

term *heap-open-union-C*
thm *heap-open-union-C-def*

Anonymous Structures and Union

For anonymous structures and types we create names by considering the surrounding typedef and structure / union. For example see the typedef of *my-union*. The canonical variant for the union gets *my-union-C*. The variant *f* is an anonymous structure. The type of the anonymous structure is type *my-union-C'f-C'struct*, whereas the type of the variant *f* is *my-union-C'f-C*. Analogously there is also a suffix *union* in case of an anoymous union. Note the naming convention: For nested names note that sufffix *union* or *struct*

is for the nested union or structure itself, in contrast to the name without suffix which denotes the variant of the enclosing union.

typ *my-union-C*
term *my-union-C*

typ *my-union-C'f-C'struct*
term *f1-C*
thm *f1-C-def*

typ *my-union-C'f-C*
term *f-C*
thm *f-C-def*

typ *my-union-C'g-C'union*
term *x1-C*
thm *x1-C-def*

typ *my-union-C'g-C*
term *g-C*
thm *g-C-def*

Lemmas to coerce values / heap accesses

thm *coerce-id*
thm *coerce-cancel-packed*
thm *coerce-map-id*
thm *coerce-coerce-map-cancel-packed*

thm *h-val-coerce-ptr-coerce-packed*
thm *h-val-field-ptr-coerce-from-bytes-packed*
thm *heap-update-field-ptr-coerce-from-bytes-packed*
thm *heap-state.L2-modify-heap-update-ptr-coerce-packed-conv*
thm *heap-state.L2-modify-heap-update-ptr-coerce-packed-field-root-conv*
thm *L2-modify-heap-update-field-root-conv*

end

end

26.21 Pointers into Structures in Split Heap

theory *open-struct*
imports *AutoCorres*
begin

26.21.1 Overview

Heap lifting in AutoCorres transforms the monolithic byte-level heap *heap-mem* with explicit term level heap type description *heap-typ-desc* and typing constraints, like *c-guard*, *h-t-valid* and $d \Vdash_t p$ to a more abstract split heap model with implicit HOL-typing. The major advantage of the split heap model is that, updates to one particular heap do not affect all other distinct heaps, removing the obligation to explicitly take care about aliasing. Aliasing between different heaps is ruled out by the model already.

The question that arises is on which granularity do we split the byte-level heap? The original AutoCorres splitted at the granularity of types. Each distinct c-type is mapped into a separate heap. In that model you can still have intra-type aliasing (i.e. pointers of the same type can alias), but no inter-type aliasing (i.e. pointers of different type are distinct, and cannot alias). As an example a pointer to a list structure refers to a different (split) heap as a pointer to a tree structure. So when a function modifies lists, no pointer to a tree is affected.

Of course not every C-program can be represented in this split-heap model. In those cases the abstraction fails. In particular think of a low-level memory allocator, that allocates some raw-bytes and delivers a void-pointer, that is then retyped to point to the actual structure. To be able to combine those byte-level functions with high-level typed functions (in the split heap) one can switch between both layers of abstraction by *exec-concrete* and *exec-abstract*. The good thing is that this gives us a very expressive tool to switch between byte-level and typed view. We can do the specification and proofs of different parts of the program on the adequate level and still combine the results. But switching between the layers can be tedious and thus should be limited to the low-level functions that really inherently need byte-level reasoning.

Unfortunately the type granularity of the split heaps also rules out programs dealing with pointers into a structure. For example a structure containing a *int* field *count* cannot pass a *int* pointer to that field *count* to a another function, that might update it. The complete structure has a separate (split) heap that is distinct from the heap containing *int*s. This would mean that we have to resort to the byte-level heap to reason about those functions.

We extended the split heap model to support this common use-case. The core idea is that in addition to split heaps on the granularity of types you can also specify split heaps on the granularity of structure fields. Conceptually you 'open' the structure into its components (aka. fields) and each field gets a separate heap. Additionally you can specify which fields should be *addressable* and thus shared. In the example you can specify that *count* is addressable, so it is represented within the same heap as all other *int*. This means that an *int* pointer can now point to a plain *int* or to a *count* field

within a structure. You explicitly allow this limited inter-type aliasing on *int* pointers. All other fields that are not explicitly marked as *addressable* still have their own heap and thus cannot alias with another pointer of the same field type. Note that mainly for performance reasons we actually avoid to have a separate heap for each single field but try to minimise the number of heaps by clustering the fields that are treated the same. Some more details on this construction are below.

26.21.2 Example Program and some Intuition

We illustrate the approach with the following example specifying some structures, with addressable and non-addressable fields. Note that for an addressable field that is of an array type, each element of the array is considered to be addressable.

install-C-file *open-struct.c*

declare [[*show-types=false*, *show-sorts=false*,
verbose = 0, *verbose-timing = 0*, *ML-print-depth = 1000*, *goals-limit = 20*]]

init−autocorres [*addressable-fields* =
inner.fld1
inner.fld2
outer.inner
outer-array.inner-array
array.elements
other.fy
two-dimensional.matrix
data-struct1.d1.y1
data-struct1.d1.y2
data-struct1.d1.y3
data-struct1.d1.y4
data-struct1.d2
data-struct2.d
data-array.array] *open-struct.c*

autocorres *open-struct.c*

Which split heaps are generated? Each split heap is a component of the record *lifted-globals*.

print-record *lifted-globals*

Structure *closed* does not have any addressable field. So there is a separate heap for that component *closed-C*.

Structure *inner* has three *unsigned* fields *fld1-C*, *fld2-C* and *fld3-C*, The first two fields are addressable whereas the third one is not. This means that the first two fields are put into the same common heap for unsigned

integers *heap-w32* whereas the third field is stored in a dedicated heap for all remaining fields *dedicated-heap-inner-C*.

We use the following nomenclature to distinguish the various kinds of split heaps and types:

- closed type or structure: There is no addressable field within the type / structure, e.g. *closed-C*.
- open type or structure: There is at least one addressable field within the type / structure, e.g. *inner-C*.
- atomic heap: Heap for a closed type, or a common heap that might be nested in an open structure. e.g. *heap-closed-C*.
- dedicated heap: Internal heap for all non-addressable fields of an open type, e.g. *dedicated-heap-inner-C*. These dedicated heaps are used for the construction of a derived heap.
- derived heap: Heap for an open type / structure. The name 'derived' indicates that it is not a fundamental heap but a composition of several common heaps and a dedicated heap, e.g. *outer-C.inner-C*
- fundamental heap: those heaps that are directly present in the new global state *lifted-globals*. These are atomic heaps, and dedicated heaps.
- virtual heap: heaps that represent the user level view. They are directly visible in the program after heap-lifting. This excludes dedicated heaps, which non-the-less may be used in the model as representation of that type.

  Dedicated heaps play a special role, as they are subsumed within derived heaps. In a sense they are not part of the 'API' for split heaps: After heap lifting the resulting program only directly mentions atomic, common or derived heaps. Nevertheless dedicated heaps are important for the construction of derived heaps and might show up after unfolding the definitions for derived heaps.

  Also dedicated heaps may be used by the user as the actual heap for a specific type. Only the dedicated heaps and the atomic heaps commute, the derived heaps do not necessary commute with each other as they might have common components that alias each other.

26.21.3 Background

The starting point for the split heap abstraction is still the unified memory model (UMM) by Harvey Tuch (https://trustworthy.systems/publications/papers/Tuch%3Aphd.pdf). The UMM provides the general concept of *c-type* to convert between a byte-list representations of C values to abstract HOL-types. It also provides an explicit notion of typing and what a well-typed byte level heap is (with respect to a heap type description). With these notions in place one can describe the set of valid pointers for a C program. The model is quite elaborate and respects the potential nesting of structures. A well-typed pointer to a structure in the heap gives rise to a bunch of other valid sub-pointers, pointing to sub-fields, which again could point to a nested structure. In the UMM, array types are modelled as a special case of a structure, where each index corresponds to an artificial field, where the field name is a unary encoding of the index. Hence, for most of the explanation in this file it is sufficient to elaborate on structures. The main limitation of the UMM is that it does not handle unions.

For the original split heap version of AutoCorres https://trustworthy.systems/publications/nicta_full_text/8758.pdf, David Greenaway put a simplified layer on top of the UMM focusing only on the root-pointers of a structure, ignoring any nested pointers to a field. This provides a split heap abstraction on the granularity of types.

The new split heap model takes an intermediate view. For closed structures it coincides with the original split heap model. Only root-pointers to a closed structure are valid. In addition for open structures also the nested pointers to addressable fields are welcome and valid. This flexibility allows the user to carefully open up the structures only as far as needed. Note the trade-off between expressibility and "proof-burden" of open structures. Aliasing is by construction ruled out for distinct split heaps, but has to be dealt with explicitly on the user level for common heaps.

To allow this flexibility some existing notions where slightly refined and some notions where introduced.

Footprints and valid pointers

The central typing judgement of UMM is *h-t-valid*, with infix syntax $d,g \Vdash_t p$, meaning pointer p is valid with respect to heap type description d and a guard g. The default guard is *c-guard*, ensuring that the pointer is aligned, is not NULL and that the referenced value fits into the address space, meaning that the addresses of the individual bytes do not overflow the address space. This instantiation with *c-guard* is abbreviated with $d \Vdash_t p$.

thm *c-guard-def*
thm *TypHeap.h-t-valid-def*
thm *h-t-valid-valid-footprint*

The judgement $d,g \models_t p$ also ensures that the footprint is valid: *valid-footprint* d (*ptr-val* p) (*typ-uinfo-t* $TYPE('a)$). Note that p is a typed pointer, carrying the (phantom) type of the value it points to p. This type annotation is no longer present in *valid-footprint* which only talks about the address of the pointer *ptr-val* and relates it to a type tag via *typ-uinfo-t*.

thm *valid-footprint-def*

Without going into details of the definition of *valid-footprint* we get guarantees for all nested pointers of the structure which also have *valid-footprint*. Moreover, what we do not know is whether the pointer p is itself a nested pointer of some enclosing structure.

To get the additional knowledge that term p is a root pointer, meaning it is not enclosed in a bigger structure, we have the notion *valid-root-footprint*. This notion was refined from the original version of AutoCorres *valid-simple-footprint* to get the important property that every *valid-root-footprint* is indeed also a *valid-footprint*.

thm *valid-root-footprint-valid-footprint*

Also for all types that fit into the address space we can go from *valid-root-footprint* to *valid-simple-footprint*. The restriction on the address space size follows from the implicit property of the legacy *valid-simple-footprint*: The first byte and all the rest have a different tag, so there must not be a complete wrap around in the address space.

thm *valid-simple-footprint-size-td*
thm *valid-root-footprint-valid-simple-footprint*

The corresponding judgment to $d \models_t p$ for root pointers is *root-ptr-valid* d p. It implies *c-guard* p as well as *valid-root-footprint* d (*ptr-val* p) (*typ-uinfo-t* $TYPE('a)$).

thm *root-ptr-valid-def*

From byte heap to typed heaps

In UMM the core wrapper functions to provide a typed view on the raw byte level heap are:

- *h-val* to lookup typed values by typed pointers and
- *heap-update* to update the heap on the location designated by a typed pointer with a typed value.

  Provided a byte level heap and a heap type description there is a function to lift the heap into a typed heap:

- *lift-t* takes a pointer guard and a pair of byte level heap and heap type description and provides a partial function from typed pointers

to typed values. The most prominent instance is abbreviation *clift* = *clift*.

The resulting function is partial as it only lifts those pointers that actually point to valid addresses according to *h-t-valid*. This fact and the connection to *h-val* is reflected in *lift-t ?g (?h, ?d)* = $(\lambda p.$ *if ?d,?g* $\models_t$ *p then Some (h-val ?h p) else None*).

thm *lift-t-if*

To lift root pointers there is

- *simple-lift*. It yields only defined values for root pointers, according to *root-ptr-valid*.

thm *simple-lift-def*

For the derived heaps we introduce the family of functions *plift*, which are defined in *wf-ptr-valid* and instantiated for each collection of open types. Like *clift* the definition is based on *lift-t*, but instead of the plain *c-guard* a custom validity-guard for the type is supplied *PTR-VALID*($'a$). The relevant definitions are collected as named theorems:

thm *plift-defs*

Note that we used to introduced a separate distinct instance of *plift* and *PTR-VALID*($'a$) for each type. Now we have encapsulated this construction in *open-types*. From a specification of which fields of a type should be addressable we first derive an variant without phantom typing of all the concepts especially *open-types.ptr-valid-u* and then define the phantom typed version based on that. This separation works around the limitation of the Isabelle type system that are also reflected in locales. It does not allow explicit quantification on type variables. So we can only have fixed type "variables" in the locale assumptions. However, the lemmas derived within a locale can still be polymorphic. So we express the needed assumptions in the locale *open-types* in the "untyped" world and derive the "typed" polymorphic versions as lemmas within that locale.

thm *open-struct.ptr-valid-u.simps*
thm *open-types.ptr-valid-u.simps*
thm *open-struct.ptr-valid-def*
thm *open-types.ptr-valid-def*

Let us look into the definitions for the open structure *inner-C* which itself is part of the open structure *outer-C*.

lemma *plift h = lift-t (ptr-valid (hrs-htd h)) h*
⟨*proof*⟩

thm *inner-C.ptr-valid.unfold*
lemma
fixes *p*::*inner-C ptr*
shows *ptr-valid d p* =
(*root-ptr-valid d p* ∨
(∃ *q*::*outer-C ptr*. *ptr-valid d q* ∧ *ptr-val p* = &(*q*→[''*inner-C*'']) ∨
(∃ *i*<5. ∃ *q*::(*inner-C*[5]) *ptr*. *ptr-valid d q* ∧ *ptr-val p* = &(*q*→[*replicate i CHR* ''*1*'']))))
⟨*proof*⟩

lemma
fixes *p*::*outer-C ptr*
shows *ptr-valid d p* = *root-ptr-valid d p*
⟨*proof*⟩

A pointer to an *inner-C* is valid according to *PTR-VALID*('*a*) *d p*, if it is either

- a root pointer to an *inner-C*, or
- there is a valid pointer to an enclosing pointer *q* to*outer-C*, where *p* is obtained by dereferencing field ''*inner-C*'', (Note that valid *outer-C* are already root pointers). or
- there is a valid pointer to an array *q*, where *p* is obtained by indexing into the array. Indexing into arrays is modelled by unary encoded field names *replicate i CHR* ''*1*''. (Note that valid array pointers themselve might have various possible extensions to a root pointer of an enclosing structure)

  These pointer-valid predicates are derived from the specification of addressable fields in the corresponding **autocorres** command. They enumerate the possibilities to arrive at a valid pointer starting from a valid root-pointer. The specification is converted to the parameter $\mathcal{T}$ of *open-types*. In our example program this is:

thm $\mathcal{T}$*-def*

The $\mathcal{T}$ is an association list mapping a type tag *typ-uinfo-t* of a *mem-type* to the list of addressable fields within that structure. From this specification we derive a lot of useful notions within the locale. Notably:

thm *ptr-valid* — These go up the chain to the root pointer.

From typed heaps to split heaps

Note that the functions in the previous subsubsection still are defined on the level of the monolithic UMM memory model. They are central building blocks to finally arrive at the split heap model. Essentially there is a

split heap component in the lifted globals corresponding to *plift* for all the fundamental types.

However, instead of partial functions as provided by *wf-ptr-valid.plift* the split heap separates definedness from the actual value, by providing a pair of a total function from pointer to value and a validity predicate which indicates definedness. This design choice was introduced in the original AutoCorres.

As a prerequisite to lift the Functions into the split heap model, the new global state record *lifted-globals* is defined. For each fundamental heap this record contains a field for the split heap e.g. *heap-w32*.

These are the lookup functions provided by the record package, there are of course also the corresponding update functions, e.g.: *heap-w32-update* Note that the update function behaves like a "map", which maps an update function on the selected heap to an update function on *lifted-globals*.

The typing information (heap type description) is preserved from the monolithic heap in component *heap-typing*.

print-record *lifted-globals*

For the derived heaps we provide a similar abstraction level by defining a derived heap lookup and a derived heap update function. These functions are a composition of the underlying functions for the fields of the structure.

thm *derived-heap-defs*

For example for *outer-C* we have.

- Lookup: *heap-outer-C*
- Pointwise update: *heap-outer-C-map*
- Validity is generically based on the heap typing for all heaps: $\lambda s.$ *IS-VALID*$('a)$ s

  Note the difference in the signature of the derived update *heap-outer-C-map* and the fundamental *heap-w32-update*. Besides the difference in naming between 'map' and 'update', the derived update is defined pointwise instead of heap-wide. The reason is that the pointwise update is what we actually need for autocorres and it is natural to break down a pointwise update defined via a function on f to the update functions on the fields of *outer-C*.

thm *heap-outer-C-def*
thm *heap-outer-C-map-def*

The connection between *globals* (containing the byte level heap) and *lifted-globals* is defined via the function *lift-global-heap*. The fundamental heaps are lifted via the corresponding *wf-ptr-valid.plift* functions. Validity

of pointers is maintained by literally preserving the heap typing from the monolithic heap.

thm *lift-global-heap-def*
thm *lift-t-def*
thm *plift-defs*

26.21.4 User Level

Due to the abstraction layer of derived heap types, the structure of the programs after heap lifting is analogous for closed and opened structures.

context *ts-definition-set-c1*
begin
thm *ts-def*

lemma *set-c1' p c* ≡ *do* {*guard* (λ*s*. *IS-VALID*(*closed-C*) *s p*);
modify (*heap-closed-C-update* (λ*h*. *h*(*p* := *c1-C-update* (λ-. *c*) (*h p*))))
}
⟨*proof*⟩

end

context *ts-definition-outer-fld1-upd*
begin
thm *ts-def*
end

context *ts-definition-set-element*
begin
thm *ts-def*
end

context *ts-definition-set-matrix-element*
begin
thm *ts-def*
end

Of course, when it comes to specifications and proving properties about the programs, the user has to take care about aliasing of addressable fields. AutoCorres generates a bunch of theorems and sets up the simpsets with some obvious rules for derived heaps, which hopefully helps in doing so. However, as a last resort one might still have to unfold the definitions and work with the parts. Here is some examples of some theorems or collections of theorems.

thm *lifted-globals-ext-simps*
thm *ptr-valid*

thm *heap-inner-C.write-comp*
thm *heap-inner-C.write-id*
thm *heap-inner-C.write-other-commute*
thm *heap-inner-C.write-same*
thm *update-commute*
thm *read-write-same*
thm *read-write-other*
thm *write-cong*
thm *update-compose*
thm *valid-implies-c-guard*
thm *read-commutes*
thm *write-commutes*

thm *fg-cons-simps*
thm *typ-info-simps*
thm *td-names-simps*
thm *typ-name-simps*
thm *upd-lift-simps*
thm *upd-other-simps*
thm *size-align-simps*
thm *fl-Some-simps*
thm *fl-ti-simps*
thm *typ-tag-defs*
thm *size-simps*
thm *typ-name-itself*

addressable-fields,*merge-addressable-fields*,*read-dedicated-heap*,*write-dedicated-heap*

We distinguish three aspects when thinking and reasoning about split heaps: the monolithic heap stored in *globals* the split heap components in *lifted-globals* and the function *lift-global-heap* which transforms between both types. Autocorres does a refinement proof that a program operating on *lifted-globals* refines a program on *globals* where the heaps are related by function *lift-global-heap*.

For open structures, the fields which are addressable have two representations in *lifted-globals*:

1. the relevant value in the common heap, i.e. *heap-w32* for the *inner.fld1* pointers

2. an unused value in the field of the dedicate heap, i.e. the *fld1-C* in the *dedicated-heap-inner-C*. Having the dedicated heap allows us to keep all non-addressable fields of a structure in a single split heap component of *lifted-globals*. The values of the addressable fields are irrelevant for the virtual heap. When reading from a virtual heap we start with the value in the dedicated heap and overwrite the addressable fields by considering the common heaps.

thm *heap-inner-C-def*

When writing to a virtual heap we have a series of updates on *lifted-globals*. We start by updating the non-addressable fields in the dedicated heap and then updating the common heaps with the new values for the addressable fields:

thm *heap-inner-C-map-def*

To express "update the non-addressable fields of the dedicated heap" in an uniform way we make use of the concept of *scenes* from the world of *lenses*. In contrast to lenses, scenes are formalised by properties of a single function $'a \Rightarrow 'a \Rightarrow 'a$ which is referred to as *merge* or 'overrider' operation. The intuition of *merge x y* is that we take certain portions of a compound value x and merge (or override) them into another compound value y. Scenes are formalised in locale *is-scene*. The notion *merge* emphasises the the symmetric nature of the merge operation. Like a lense, a scene 'focuses' on a portion of a compound value $'a$. The merge operation takes that portion from the first argument and the complement of that portion from the second argument. Think for example of a pair and the 'portion' being the first component. Then *merge x y* = (*fst x*, *snd y*). The notion *overrider* puts more emphasis on the merge as an update function, in particular this view becomes obvious if we think of a partial application *merge x*: this is an update function which overrides the 'portion' within an value y to that portion of x. You can also think of fixing the 'portion' to those values in x. Analogously λx. *merge x y* fixes the complement of the 'portion' to those values in y. Scenes are closely related to lenses in the sense that it is straightforward to derive a scene from a lense *lense ?r ?w* $\Longrightarrow$ *is-scene* (λa. *?w* (λ-. *?r a*)).

thm *is-scene-of-lense*

For HOL the decisive advantage of scenes compared to lenses is that they only have one type parameter, which fixes the type of the compound value. But still you can formally 'talk' about different 'portions' of that value in an uniform way. For example you can have a list or a set of scenes to describe their composition. This property can be used to uniformly describe "all addressable fields" of a value or "all non-addressable fields" of a value.

The relevant merging function for us is *merge-addressable-fields* which is an abbreviation $\lambda\mathcal{T}$. *merge-ti-list* (*map snd* (*open-types.addressable-fields* $\mathcal{T}$ *TYPE*($'a$))).

term ‹*open-types.merge-addressable-fields*›

The term*merge-addressable-fields old new* has to effects:

- The addressable fields are taken from *old*
- The non-addressable fields are taken from *new*.

To express the same semantics that we have just described for *lifted-globals* on the UMM heap in *globals* we make use of the functions *read-dedicated-heap* and *write-dedicated-heap*. These functions internally again make use of *merge-addressable-fields*. We zero out all parts of the compound value we are not actually interested in. This gives us canonical values which we can easily relate to the *lifted-globals*. For example for *read-dedicated-heap* we zero out all addressable fields and also all values which are not even *PTR-VALID*('*a*) (according to *plift*).

lemma (**in** *open-types*)
read-dedicated-heap h p =
merge-addressable-fields ZERO('*a*::*mem-type*) (*the-default ZERO*('*a*) (*plift h p*))
⟨*proof*⟩

thm *open-types.read-dedicated-heap-def*
thm *open-types.write-dedicated-heap-def*

The lifting function *lift-global-heap* directly uses *read-dedicated-heap* to construct a dedicated heap from the UMM heap:

thm *lift-global-heap-def*

To have a canonical representation we overwrite the fields in the dedicated heap with zeros, using the generic constants *addressable-fields* and *merge-addressable-fields* defined in *open-types*.

read-dedicated-heap,*write-dedicated-heap* combine these with *h-val*, to get the canonical heap access.

Why all the variants? *h-val* / *read-dedicated-heap*

thm *lifted-globals-ext-simps*
thm *lift-global-heap-def*
thm *open-types.read-dedicated-heap-def*

When taking a close look in the definition of *lift-global-heap* one sees that the abstraction for a split heap component from the monolithic heap is:

read-dedicated-heap (*t-hrs-*′ *g*)

So this might seem a bit surprising at first sight. What is all the indirection useful for:

1. the *read-dedicated-heap* adds an option layer to the plain *h-val* and *plift*
2. with *t-hrs-*′ we get the heap representation (including bytes and types)

   When we know that a pointer is valid there is a tight connection between *read-dedicated-heap* and *h-val*

thm *open-struct.ptr-valid.ptr-valid-plift-Some-hval*
thm *open-struct.read-dedicated-heap-def*

This connection directly carries over to lifting.

thm *read-commutes*

The difference between *read-dedicated-heap* and *h-val* becomes apparent when looking at partial heaps and invalid pointers. While *read-dedicated-heap* always yields $ZERO('a)$ the plain *h-val* will still construct some value out of the bytes in the heap. So in a sense the indirection to *read-dedicated-heap* makes heaps equal if the agree on the valid pointers only. So we encode "heap equality only on valid pointers" in an ordinary equality on the complete lifted heap. This choice ensure compositionality for the polymorphic $ZERO('a)$, in the sense that the $ZERO('a)$ of a structure means that all subcomponents are $ZERO('a)$.

Moreover *read-dedicated-heap* only reads the fields which are not addressable, and overwrites the addressable fields with $ZERO('a)$. This gives us a canonical view on the partial heap.

thm *zero-simps*
thm *open-struct.ptr-valid.plift-None*

I miss the typing in the split heap!

A peculiar property of the original split heap model of autocorres was that you loose parts of the type information that were directly available in the byte-level heap. There was no heap-type description in *lifted-globals* but only the more abstract *is-valid-<type>* fields for each type. For closed types this isn't so much of an issue as essentially all relevant type information is captured in the fact that all pointers to that type have a distinct split heap, which is separate from all other split heaps. But for shared types this is no longer true. Thus we maintain the typing information also in the split heap in the compponent *heap-typing*. The relation to the typing information of the original byte-level heap is encapsulated in *lift-global-heap*. For low-level operations that are embedded via *exec-concrete* you can directly connect the typing of the monolithic and the split heap. In particular you can derive *c-guard p* from that information. Or you can derive that if you have two valid pointers of the same type, where you only know that the address is different, you can still conclude that the complete pointer span of the pointers do not overlap.

But keep in mind that typing is only a "discipline". As in the split heap you can manipulate the values on the heap or split heap independent of the typing information. When well-typedness or $PTR\text{-}VALID('a)$ is available you might be able to derive properties like distinctness of pointer spans. The distinctness of pointers spans is the actual reason that certain heap updates commute, e.g. *disjnt* (*ptr-span ?p*) (*ptr-span ?q*) $\Longrightarrow$

heap-inner-C-map ?q ?g (heap-inner-C-map ?p ?f ?s) = heap-inner-C-map ?p ?f (heap-inner-C-map ?q ?g ?s)

thm *heap-inner-C.write-other-commute*

26.21.5 Simulation Proof

The simulation of a concrete program C operating on the byte-level heap by an abstract program A operating on the split heaps is captured in predicate*L2Tcorres lift-global-heap A C*. The proof for an instance of C follows the syntactic structure of C by applying introduction rules and synthesizing the abstract program A.

Intuitively the core properties on which the simulation proof builds are:

- Abstract programs only operate on abstract heap values not on byte-lists. Byte-level concepts like padding fields are irrelevant for the abstract program. The abstract heaps and programs don't distinguish between two byte-level heaps if they agree on all values of non-padding fields of properly allocated and typed portions of the heap.
- Lookup and update via a pointer into a structure can be simulated by an lookup and update via the root pointer of the structure.
- Each non-addressable field of a structure is mapped into exactly one split heap. For a 'closed' structure without any addressable fields this is the heap for the type.
- For an 'open' structure an addressable field is mapped into a shared heap or multiple shared heaps in case the field is again an open structure.
- The pointer spans of two valid root pointers of different types do not overlap.
- The pointer spans of two valid root pointers to the same type might overlap only if the pointer value is the same.

  The actual proof of a simulation is divided into three main steps.

- First some general theorems relating byte-level and split heaps are derived. Most prominently *typ-heap-simulation* and related predicates.
- These theorems are used to instantiate the syntax driven introduction rules collected in named theorems ⟦*abs-expr ?st ?X ?f1.0 ?f1'*; *abs-expr ?st ?Y ?f2.0 ?f2'*⟧ ⟹ *abs-expr ?st* (λ*s*. *?X s* ∧ *?Y s*) (λ*s*. (*?f1.0 s*, *?f2.0 s*)) (λ*s*. (*?f1' s*, *?f2' s*))

  abs-expr ?st (λ-. *True*) (λ*s*. *?a*) (λ*s*. *?a*)

  abs-expr ?st ?P ?a' ?a ⟹ *abs-guard ?st* (λ*s*. *?P s* ∧ *?a' s*) *?a*

abs-guard ?st (λ-. *?P*) (λ-. *?P*)

⟦*abs-guard ?st ?G ?G′*; *abs-guard ?st ?H ?H′*⟧ ⟹ *abs-guard ?st* (λ*s*. *?G s* ∧ *?H s*) (λ*s*. *?G′ s* ∧ *?H′ s*)

⟦*struct-rewrite-modifies ?P ?b ?c*; *abs-guard ?st ?P′ ?P*; *abs-modifies ?st ?Q ?a ?b*⟧ ⟹ *L2Tcorres ?st* (*L2-seq* (*L2-guard* (λ*s*. *?P′ s* ∧ *?Q s*)) (λ-. *L2-modify ?a*)) (*L2-modify ?c*)

⟦*struct-rewrite-expr ?P ?b ?c*; *abs-guard ?st ?P′ ?P*; *abs-expr ?st ?Q ?a ?b*⟧ ⟹ *L2Tcorres ?st* (*L2-seq* (*L2-guard* (λ*s*. *?P′ s* ∧ *?Q s*)) (λ-. *L2-gets ?a ?n*)) (*L2-gets ?c ?n*)

L2Tcorres ?st (*L2-gets* (λ-. *?a*) *?n*) (*L2-gets* (λ-. *?a*) *?n*)

⟦*struct-rewrite-guard ?b ?c*; *abs-guard ?st ?a ?b*⟧ ⟹ *L2Tcorres ?st* (*L2-guard ?a*) (*L2-guard ?c*)

⟦*THIN* (⋀*x*. *L2Tcorres ?st* (*?B′ x*) (*?B x*)); *THIN* (⋀*r*. *struct-rewrite-expr* (*?G r*) (*?C′ r*) (*?C r*)); *THIN* (⋀*r*. *abs-guard ?st* (*?G′ r*) (*?G r*)); *THIN* (⋀*r*. *abs-expr ?st* (*?H r*) (*?C″ r*) (*?C′ r*))⟧ ⟹ *L2Tcorres ?st* (*L2-seq* (*L2-guard* (λ*s*. *?G′ ?i s* ∧ *?H ?i s*)) (λ-. *L2-while ?C″* (λ*i*. *L2-seq* (*?B′ i*) (λ*r*. *L2-seq* (*L2-guard* (λ*s*. *?G′ r s* ∧ *?H r s*)) (λ-. *L2-gets* (λ-. *r*) *?n*))) *?i ?n*)) (*L2-while ?C ?B ?i ?n*)

abs-spec ?st ?P ?A ?C ⟹ *L2Tcorres ?st* (*L2-seq* (*L2-guard ?P*) (λ-. *L2-spec ?A*)) (*L2-spec ?C*)

abs-assume ?st ?P ?A ?C ⟹ *L2Tcorres ?st* (*L2-seq* (*L2-guard ?P*) (λ-. *L2-assume ?A*)) (*L2-assume ?C*)

abs-spec ?st (λ-. *True*) {(*a*, *b*). *?C*} {(*a*, *b*). *?C*}

⟦*THIN* (*L2Tcorres ?st ?L ?L′*); *THIN* (*L2Tcorres ?st ?R ?R′*); *THIN* (*struct-rewrite-expr ?P ?C′ ?C*); *THIN* (*abs-guard ?st ?P′ ?P*); *THIN* (*abs-expr ?st ?Q ?C″ ?C′*)⟧ ⟹ *L2Tcorres ?st* (*L2-seq* (*L2-guard* (λ*s*. *?P′ s* ∧ *?Q s*)) (λ-. *L2-condition ?C″ ?L ?R*)) (*L2-condition ?C ?L′ ?R′*)

⟦*THIN* (*L2Tcorres ?st ?L′ ?L*); *THIN* (⋀*r*. *L2Tcorres ?st* (*?R′ r*) (*?R r*))⟧ ⟹ *L2Tcorres ?st* (*L2-seq ?L′ ?R′*) (*L2-seq ?L ?R*)

⟦*struct-rewrite-guard ?b ?c*; *abs-guard ?st ?a ?b*; ⋀*s s′*. ⟦*?c s*; *s′* = *?st s*⟧ ⟹ *L2Tcorres ?st ?f ?g*⟧ ⟹ *L2Tcorres ?st* (*L2-guarded ?a ?f*) (*L2-guarded ?c ?g*)

⟦*THIN* (*L2Tcorres ?st ?L ?L′*); *THIN* (⋀*r*. *L2Tcorres ?st* (*?R r*) (*?R′ r*))⟧ ⟹ *L2Tcorres ?st* (*L2-catch ?L ?R*) (*L2-catch ?L′ ?R′*)

L2Tcorres ?st ?L ?L′ ⟹ *L2Tcorres ?st* (*L2-try ?L*) (*L2-try ?L′*)

L2Tcorres ?st (*L2-unknown ?ns*) (*L2-unknown ?ns*)

L2Tcorres ?st (*L2-throw ?x ?n*) (*L2-throw ?x ?n*)

$(\bigwedge x\ y.\ L2Tcorres\ ?st\ (?P\ x\ y)\ (?P'\ x\ y)) \Longrightarrow L2Tcorres\ ?st\ (case\ ?a\ of\ (x,\ y) \Rightarrow ?P\ x\ y)\ (case\ ?a\ of\ (x,\ y) \Rightarrow ?P'\ x\ y)$

$[\![THIN\ (L2Tcorres\ ?st\ ?L\ ?L');\ THIN\ (L2Tcorres\ ?st\ ?R\ ?R')]\!] \Longrightarrow L2Tcorres\ ?st\ (L2\text{-}seq\ ?L\ (\lambda\text{-}.\ ?R))\ (L2\text{-}seq\ ?L'\ (\lambda\text{-}.\ ?R'))$

$(\bigwedge a\ b.\ abs\text{-}expr\ ?st\ (?P\ a\ b)\ (?A\ a\ b)\ (?C\ a\ b)) \Longrightarrow abs\text{-}expr\ ?st\ (case\ ?r\ of\ (a,\ b) \Rightarrow ?P\ a\ b)\ (case\ ?r\ of\ (a,\ b) \Rightarrow ?A\ a\ b)\ (case\ ?r\ of\ (a,\ b) \Rightarrow ?C\ a\ b)$

$(\bigwedge a\ b.\ abs\text{-}guard\ ?st\ (?A\ a\ b)\ (?C\ a\ b)) \Longrightarrow abs\text{-}guard\ ?st\ (case\ ?r\ of\ (a,\ b) \Rightarrow ?A\ a\ b)\ (case\ ?r\ of\ (a,\ b) \Rightarrow ?C\ a\ b)$

$L2Tcorres\ ?st\ L2\text{-}fail\ L2\text{-}fail$

$abs\text{-}expr\ id\ (\lambda\text{-}.\ True)\ ?A\ ?A$

$abs\text{-}expr\ ?st\ ?P\ ?A\ ?C \Longrightarrow abs\text{-}expr\ ?st\ ?P\ (\lambda s\ r.\ ?A\ s)\ (\lambda s\ r.\ ?C\ s)$

$abs\text{-}modifies\ id\ (\lambda\text{-}.\ True)\ ?A\ ?A$

$L2Tcorres\ id\ ?A\ ?C \Longrightarrow L2Tcorres\ ?st\ (exec\text{-}concrete\ ?st\ (L2\text{-}call\ ?A\ ?emb\ ?ns))\ (L2\text{-}call\ ?C\ ?emb\ ?ns)$

$L2Tcorres\ id\ ?A\ ?C \Longrightarrow L2Tcorres\ ?st\ (exec\text{-}concrete\ ?st\ (L2\text{-}call\text{-}L1\ ?arg\text{-}xf\ ?gs\ ?ret\text{-}xf\ ?A))\ (L2\text{-}call\text{-}L1\ ?arg\text{-}xf\ ?gs\ ?ret\text{-}xf\ ?C)$

$L2Tcorres\ ?st\ ?A\ ?C \Longrightarrow L2Tcorres\ id\ (exec\text{-}abstract\ ?st\ (L2\text{-}call\ ?A\ ?emb\ ?ns))\ (L2\text{-}call\ ?C\ ?emb\ ?ns)$

$L2Tcorres\ ?st\ ?A\ ?C \Longrightarrow L2Tcorres\ ?st\ (L2\text{-}call\ ?A\ ?emb\ ?ns)\ (L2\text{-}call\ ?C\ ?emb\ ?ns)$

$[\![typ\text{-}heap\text{-}simulation\ ?st\ ?getter\ ?setter\ ?vgetter\ ?t\text{-}hrs\ ?t\text{-}hrs\text{-}update;\ abs\text{-}expr\ ?st\ ?P\ ?x'\ ?x]\!] \Longrightarrow abs\text{-}guard\ ?st\ (\lambda s.\ ?P\ s \wedge ?vgetter\ s\ (?x'\ s))\ (\lambda s.\ c\text{-}guard\ (?x\ s))$

$[\![typ\text{-}heap\text{-}simulation\ ?st\ ?r\ ?w\ ?v\ ?t\text{-}hrs\ ?t\text{-}hrs\text{-}update;\ abs\text{-}expr\ ?st\ ?P\ ?x'\ ?x]\!] \Longrightarrow abs\text{-}expr\ ?st\ (\lambda s.\ ?P\ s \wedge ?v\ s\ (?x'\ s))\ (\lambda s.\ ?r\ s\ (?x'\ s))\ (\lambda s.\ h\text{-}val\ (hrs\text{-}mem\ (?t\text{-}hrs\ s))\ (?x\ s))$

$[\![typ\text{-}heap\text{-}simulation\ ?st\ ?r\ ?w\ ?v\ ?t\text{-}hrs\ ?t\text{-}hrs\text{-}update;\ abs\text{-}expr\ ?st\ ?Pb\ ?b'\ ?b;\ \bigwedge v.\ abs\text{-}expr\ ?st\ ?Pc\ (?c'\ v)\ (?c\ v)]\!] \Longrightarrow abs\text{-}modifies\ ?st\ (\lambda s.\ ?Pb\ s \wedge ?Pc\ s \wedge ?v\ s\ (?b'\ s))\ (\lambda s.\ ?w\ (?b'\ s)\ (\lambda\text{-}.\ ?c'\ (?r\ s\ (?b'\ s))\ s)\ s)\ (\lambda s.\ ?t\text{-}hrs\text{-}update\ (hrs\text{-}mem\text{-}update\ (heap\text{-}update\ (?b\ s)\ (?c\ (heap\text{-}lift\text{--}h\text{-}val\ (hrs\text{-}mem\ (?t\text{-}hrs\ s))\ (?b\ s))\ s)))\ s)$

$struct\text{-}rewrite\text{-}expr\ ?P\ ?a'\ ?a \Longrightarrow struct\text{-}rewrite\text{-}guard\ (\lambda s.\ ?P\ s \wedge ?a'\ s)\ ?a$

$struct\text{-}rewrite\text{-}guard\ (\lambda\text{-}.\ ?P)\ (\lambda\text{-}.\ ?P)$

$[\![struct\text{-}rewrite\text{-}guard\ ?b'\ ?b;\ struct\text{-}rewrite\text{-}guard\ ?a'\ ?a]\!] \Longrightarrow struct\text{-}rewrite\text{-}guard\ (\lambda s.\ ?a'\ s \wedge ?b'\ s)\ (\lambda s.\ ?a\ s \wedge ?b\ s)$

$(\bigwedge a\ b.\ \textit{struct-rewrite-guard}\ (?A\ a\ b)\ (?C\ a\ b)) \Longrightarrow \textit{struct-rewrite-guard}\ (\textit{case}\ ?r\ \textit{of}\ (a,\ b) \Rightarrow ?A\ a\ b)\ (\textit{case}\ ?r\ \textit{of}\ (a,\ b) \Rightarrow ?C\ a\ b)$

$[\![\textit{valid-struct-field}\ ?\textit{field-name}\ ?\textit{field-getter}\ ?\textit{field-setter}\ ?\textit{t-hrs}\ ?\textit{t-hrs-update};\ \textit{struct-rewrite-expr}\ ?P\ ?p'\ ?p;\ \textit{struct-rewrite-guard}\ ?Q\ (\lambda s.\ \textit{c-guard}\ (?p'\ s))]\!] \Longrightarrow \textit{struct-rewrite-guard}\ (\lambda s.\ ?P\ s \wedge ?Q\ s)\ (\lambda s.\ \textit{c-guard}\ (PTR(?'f)\ \&(?p\ s{\rightarrow}?\textit{field-name})))$

$[\![\textit{valid-struct-field}\ ?\textit{field-name}\ ?\textit{field-getter}\ ?\textit{field-setter}\ ?\textit{t-hrs}\ ?\textit{t-hrs-update};\ \textit{struct-rewrite-expr}\ ?P\ ?p'\ ?p;\ \textit{struct-rewrite-guard}\ ?Q\ (\lambda s.\ \textit{c-guard}\ (?p'\ s))]\!] \Longrightarrow \textit{struct-rewrite-guard}\ (\lambda s.\ ?P\ s \wedge ?Q\ s \wedge 0 \leq ?k \wedge \textit{nat}\ ?k < CARD(?'n))\ (\lambda s.\ \textit{c-guard}\ (\textit{PTR-COERCE}(?'f[?'n] \rightarrow ?'f)\ (PTR(?'f[?'n])\ \&(?p\ s{\rightarrow}?\textit{field-name}))\ +_p\ ?k))$

$\textit{struct-rewrite-expr}\ (\lambda\text{-}.\ \textit{True})\ (\lambda\text{-}.\ ?a)\ (\lambda\text{-}.\ ?a)$

$\textit{struct-rewrite-expr}\ ?P\ ?A\ ?C \Longrightarrow \textit{struct-rewrite-expr}\ ?P\ (\lambda s\ \text{-}.\ ?A\ s)\ (\lambda s\ \text{-}.\ ?C\ s)$

$(\bigwedge a\ b.\ \textit{struct-rewrite-expr}\ (?P\ a\ b)\ (?A\ a\ b)\ (?C\ a\ b)) \Longrightarrow \textit{struct-rewrite-expr}\ (\textit{case}\ ?r\ \textit{of}\ (a,\ b) \Rightarrow ?P\ a\ b)\ (\textit{case}\ ?r\ \textit{of}\ (a,\ b) \Rightarrow ?A\ a\ b)\ (\textit{case}\ ?r\ \textit{of}\ (a,\ b) \Rightarrow ?C\ a\ b)$

$\textit{struct-rewrite-expr}\ (\lambda\text{-}.\ \textit{True})\ (\lambda s.\ \textit{h-val}\ (?h\ s)\ (?p\ s))\ (\lambda s.\ \textit{h-val}\ (?h\ s)\ (?p\ s))$

$\textit{struct-rewrite-expr}\ ?P\ ?a\ ?c \Longrightarrow \textit{struct-rewrite-expr}\ ?P\ (\lambda s.\ \textit{h-val}\ (?h\ s)\ (?a\ s))\ (\lambda s.\ \textit{h-val}\ (?h\ s)\ (?c\ s))$

$[\![\textit{valid-struct-field}\ ?\textit{field-name}\ ?\textit{field-getter}\ ?\textit{field-setter}\ ?\textit{t-hrs}\ ?\textit{t-hrs-update};\ \textit{struct-rewrite-expr}\ ?P\ ?p'\ ?p;\ \textit{struct-rewrite-expr}\ ?Q\ ?a\ (\lambda s.\ \textit{h-val}\ (\textit{hrs-mem}\ (?\textit{t-hrs}\ s))\ (?p'\ s))]\!] \Longrightarrow \textit{struct-rewrite-expr}\ (\lambda s.\ ?P\ s \wedge ?Q\ s)\ (\lambda s.\ ?\textit{field-getter}\ (?a\ s))\ (\lambda s.\ \textit{h-val}\ (\textit{hrs-mem}\ (?\textit{t-hrs}\ s))\ (PTR(?'f)\ \&(?p\ s{\rightarrow}?\textit{field-name})))$

$[\![\textit{valid-struct-field}\ ?\textit{field-name}\ ?\textit{field-getter}\ ?\textit{field-setter}\ ?\textit{t-hrs}\ ?\textit{t-hrs-update};\ \textit{abs-expr}\ ?\textit{st}\ ?P\ ?a\ ?c]\!] \Longrightarrow \textit{abs-expr}\ ?\textit{st}\ ?P\ (\lambda s.\ ?\textit{field-getter}\ (?a\ s))\ (\lambda s.\ ?\textit{field-getter}\ (?c\ s))$

$[\![\textit{valid-struct-field}\ ?\textit{field-name}\ ?\textit{field-getter}\ ?\textit{field-setter}\ ?\textit{t-hrs}\ ?\textit{t-hrs-update};\ \textit{struct-rewrite-expr}\ ?P\ ?p'\ ?p;\ \textit{struct-rewrite-expr}\ ?Q\ ?a\ (\lambda s.\ \textit{h-val}\ (\textit{hrs-mem}\ (?\textit{t-hrs}\ s))\ (?p'\ s))]\!] \Longrightarrow \textit{struct-rewrite-expr}\ (\lambda s.\ ?P\ s \wedge ?Q\ s \wedge 0 \leq ?k \wedge \textit{nat}\ ?k < CARD(?'n))\ (\lambda s.\ ?\textit{field-getter}\ (?a\ s).[\textit{nat}\ ?k])\ (\lambda s.\ \textit{h-val}\ (\textit{hrs-mem}\ (?\textit{t-hrs}\ s))\ (\textit{PTR-COERCE}(?'f[?'n] \rightarrow ?'f)\ (PTR(?'f[?'n])\ \&(?p\ s{\rightarrow}?\textit{field-name}))\ +_p\ ?k))$

$[\![\textit{valid-struct-field}\ ?\textit{field-name}\ ?\textit{field-getter}\ ?\textit{field-setter}\ ?\textit{t-hrs}\ ?\textit{t-hrs-update};\ \textit{struct-rewrite-expr}\ ?P\ ?p'\ ?p;\ \textit{struct-rewrite-expr}\ ?Q\ ?a\ (\lambda s.\ \textit{h-val}\ (\textit{hrs-mem}\ (?\textit{t-hrs}\ s))\ (?p'\ s))]\!] \Longrightarrow \textit{struct-rewrite-expr}\ (\lambda s.\ ?P\ s \wedge ?Q\ s \wedge 0 \leq ?k \wedge \textit{nat}\ ?k < CARD(?'n))\ (\lambda s.\ ?\textit{field-getter}\ (?a\ s).[\textit{nat}\ ?k])\ (\lambda s.\ \textit{h-val}\ (\textit{hrs-mem}\ (?\textit{t-hrs}\ s))\ (PTR(?'f)\ \&(?p\ s{\rightarrow}?\textit{field-name})\ +_p\ ?k))$

struct-rewrite-modifies (λ-. *True*) *?A ?A*

⟦*typ-heap-simulation ?st ?getter ?setter ?vgetter ?t-hrs ?t-hrs-update*; *struct-rewrite-expr ?P ?p′ ?p*; *struct-rewrite-expr ?Q ?v′ ?v*⟧ ⟹ *struct-rewrite-modifies* (λ*s*. *?P s* ∧ *?Q s*) (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?p′ s*) (*?v′ s*))) *s*) (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?p s*) (*?v s*))) *s*)

heap-lift--wrap-h-val (*heap-lift--h-val ?s ?p*) (*h-val ?s ?p*)

heap-lift--wrap-h-val (*h-val ?s* (*PTR*(*?′a*) &(*?p*→*?f*))) (*h-val ?s* (*PTR*(*?′a*) &(*?p*→*?f*)))

heap-lift--wrap-h-val (*h-val ?s* (*PTR-COERCE*(*?′b* → *?′a*) *?p* +$_p$ *?k*)) (*h-val ?s* (*PTR-COERCE*(*?′b* → *?′a*) *?p* +$_p$ *?k*))

⟦*valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update*; *struct-rewrite-expr ?P ?p′ ?p*; *struct-rewrite-expr ?Q ?f′ ?f*; ⋀*s*. *heap-lift--wrap-h-val* (*?h-val-p′ s*) (*h-val* (*hrs-mem* (*?t-hrs s*)) (*?p′ s*)); *struct-rewrite-modifies ?R* (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?p″ s*) (*?u s* (*?field-setter* (*?f′ s*))))) *s*) (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?p′ s*) (*?field-setter* (*?f′ s*) (*?h-val-p′ s*)))) *s*); *struct-rewrite-guard ?S* (λ*s*. *c-guard* (*?p′ s*))⟧ ⟹ *struct-rewrite-modifies* (λ*s*. *?P s* ∧ *?Q s* ∧ *?R s* ∧ *?S s*) (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?p″ s*) (*?u s* (*?field-setter* (*?f′ s*))))) *s*) (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*PTR*(*?′f*) &(*?p s*→*?field-name*)) (*?f s* (*h-val* (*hrs-mem* (*?t-hrs s*)) (*PTR*(*?′f*) &(*?p s*→*?field-name*)))))) *s*)

⟦*valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update*; *struct-rewrite-expr ?P ?p′ ?p*; *struct-rewrite-expr ?Q ?f′ ?f*; ⋀*s*. *heap-lift--wrap-h-val* (*?h-val-p′ s*) (*h-val* (*hrs-mem* (*?t-hrs s*)) (*?p′ s*)); *struct-rewrite-modifies ?R* (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?p″ s*) (*?u s* (*?field-setter* (λ*a*. *Arrays.update a* (*nat ?k*) (*?f′ s* (*a*.[*nat ?k*])))))) *s*) (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?p′ s*) (*?field-setter* (λ*a*. *Arrays.update a* (*nat ?k*) (*?f′ s* (*a*.[*nat ?k*])))) (*?h-val-p′ s*)))) *s*); *struct-rewrite-guard ?S* (λ*s*. *c-guard* (*?p′ s*))⟧ ⟹ *struct-rewrite-modifies* (λ*s*. *?P s* ∧ *?Q s* ∧ *?R s* ∧ *?S s* ∧ *0* ≤ *?k* ∧ *nat ?k* < *CARD*(*?′n*)) (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?p″ s*) (*?u s* (*?field-setter* (λ*a*. *Arrays.update a* (*nat ?k*) (*?f′ s* (*a*.[*nat ?k*])))))) *s*) (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*PTR-COERCE*(*?′f*[*?′n*] → *?′f*) (*PTR*(*?′f*[*?′n*]) &(*?p s*→*?field-name*)) +$_p$ *?k*) (*?f s* (*h-val* (*hrs-mem* (*?t-hrs s*)) (*PTR-COERCE*(*?′f*[*?′n*] → *?′f*) (*PTR*(*?′f*[*?′n*]) &(*?p s*→*?field-name*)) +$_p$ *?k*))))) *s*)

valid-globals-field ?st ?old-getter ?old-setter ?new-getter ?new-setter ⟹ *abs-expr ?st* (λ-. *True*) *?new-getter ?old-getter*

⟦*valid-globals-field ?st ?old-getter ?old-setter ?new-getter ?new-setter*; ⋀*old*. *abs-expr ?st* (*?P old*) (*?v old*) (*?v′ old*)⟧ ⟹ *abs-modifies ?st*

$(\lambda s.\ \forall\ old.\ ?P\ old\ s)\ (\lambda s.\ ?new\text{-}setter\ (\lambda old.\ ?v\ old\ s)\ s)\ (\lambda s.\ ?old\text{-}setter\ (\lambda old.\ ?v'\ old\ s)\ s)$

valid-globals-field ?st ?old-getter ?old-setter ?new-getter ?new-setter $\Longrightarrow$ *struct-rewrite-expr* $(\lambda\text{-}.\ True)$ *?old-getter ?old-getter*

$[\![$*valid-globals-field ?st ?old-getter ?old-setter ?new-getter ?new-setter*; $\bigwedge$*old. struct-rewrite-expr (?P old) (?v' old) (?v old)*$]\!] \Longrightarrow$ *struct-rewrite-modifies* $(\lambda s.\ \forall\ old.\ ?P\ old\ s)\ (\lambda s.\ ?old\text{-}setter\ (\lambda old.\ ?v'\ old\ s)\ s)\ (\lambda s.\ ?old\text{-}setter\ (\lambda old.\ ?v\ old\ s)\ s)$

abs-spec ?st $(\lambda\text{-}.\ True)$ $\{(a,\ b).\ b$ *may-not-modify-globals* $a\}$ $\{(a,\ b).\ b$ *may-not-modify-globals* $a\}$

$[\![$*valid-globals-field ?st ?old-getter ?old-setter ?new-getter ?new-setter*; *abs-spec ?st* $(\lambda\text{-}.\ True)$ $\{(a,\ b).\ ?C\ a\ b\}$ $\{(a,\ b).\ ?C'\ a\ b\}]\!] \Longrightarrow$ *abs-spec ?st* $(\lambda\text{-}.\ True)$ $\{(a,\ b).\ mex\ (\lambda x.\ ?C\ (?new\text{-}setter\ (\lambda\text{-}.\ x)\ a)\ b)\}$ $\{(a,\ b).\ mex\ (\lambda x.\ ?C'\ (?old\text{-}setter\ (\lambda\text{-}.\ x)\ a)\ b)\}$

$[\![$*typ-heap-simulation ?st ?r ?w ?v ?t-hrs ?t-hrs-update*; *abs-expr ?st ?P ?x'* $(\lambda s.\ PTR\text{-}COERCE(?'a[?'b] \to ?'a)\ (?x\ s))]\!] \Longrightarrow$ *abs-guard ?st* $(\lambda s.\ ?P\ s \wedge (\forall\ a{\in}set\ (array\text{-}addrs\ (?x'\ s)\ CARD(?'b)).\ ?v\ s\ a))\ (\lambda s.\ c\text{-}guard\ (?x\ s))$

$[\![$*typ-heap-simulation ?st ?r ?w ?v ?t-hrs ?t-hrs-update*; *abs-expr ?st ?Pb ?b' ?b*; *abs-expr ?st ?Pn ?n' ?n*; *abs-expr ?st ?Pv ?y' ?y*$]\!] \Longrightarrow$ *abs-modifies ?st* $(\lambda s.\ ?Pb\ s \wedge ?Pn\ s \wedge ?Pv\ s \wedge ?n'\ s < CARD(?'b) \wedge (\forall\ ptr{\in}set\ (array\text{-}addrs\ (PTR\text{-}COERCE(?'a[?'b] \to ?'a)\ (?b'\ s))\ CARD(?'b)).\ ?v\ s\ ptr))\ (\lambda s.\ ?w\ (PTR\text{-}COERCE(?'a[?'b] \to ?'a)\ (?b'\ s)\ +_p\ int\ (?n'\ s))\ (\lambda v.\ ?y'\ s)\ s)\ (\lambda s.\ ?t\text{-}hrs\text{-}update\ (hrs\text{-}mem\text{-}update\ (heap\text{-}update\ (?b\ s)\ (Arrays.update\ (h\text{-}val\ (hrs\text{-}mem\ (?t\text{-}hrs\ s))\ (?b\ s))\ (?n\ s)\ (?y\ s))))\ s)$

$[\![$*typ-heap-simulation ?st ?r ?w ?v ?t-hrs ?t-hrs-update*; *abs-expr ?st ?Pb ?b' ?b*; *abs-expr ?st ?Pn ?n' ?n*$]\!] \Longrightarrow$ *abs-expr ?st* $(\lambda s.\ ?Pb\ s \wedge ?Pn\ s \wedge ?n'\ s < CARD(?'b) \wedge ?v\ s\ (PTR\text{-}COERCE(?'a[?'b] \to ?'a)\ (?b'\ s)\ +_p\ int\ (?n'\ s)))\ (\lambda s.\ ?r\ s\ (PTR\text{-}COERCE(?'a[?'b] \to ?'a)\ (?b'\ s)\ +_p\ int\ (?n'\ s)))\ (\lambda s.\ h\text{-}val\ (hrs\text{-}mem\ (?t\text{-}hrs\ s))\ (?b\ s).[?n\ s])$

abs-expr lift-global-heap ?P ?a ?c $\Longrightarrow$ *L2Tcorres lift-global-heap (L2-seq (L2-guard* $(\lambda t.\ IS\text{-}VALID(32\ word)\ t\ ?p \wedge ?P\ t))\ (\lambda\text{-}.\ L2\text{-}modify\ (\lambda s.\ heap\text{-}w32\text{-}update\ (\lambda h.\ h(?p := ?a\ s))\ s)))\ (globals.IO\text{-}modify\text{-}heap\text{-}paddingE\ ?p\ ?c)$

abs-expr lift-global-heap ?P ?a ?c $\Longrightarrow$ *L2Tcorres lift-global-heap (L2-seq (L2-guard* $(\lambda t.\ IS\text{-}VALID(32\ word)\ t\ (PTR\text{-}COERCE(32\ signed\ word \to 32\ word)\ ?p) \wedge ?P\ t))\ (\lambda\text{-}.\ L2\text{-}modify\ (\lambda s.\ heap\text{-}w32\text{-}update\ (\lambda h.\ h(PTR\text{-}COERCE(32\ signed\ word \to 32\ word)\ ?p := UCAST(32\ signed \to 32)\ (?a\ s)))\ s)))\ (globals.IO\text{-}modify\text{-}heap\text{-}paddingE\ ?p\ ?c)$

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(8 word) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-w8-update (λh. h(?p := ?a s)) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(8 word) t (PTR-COERCE(8 signed word → 8 word) ?p) ∧ ?P t)) (λ-. L2-modify (λs. heap-w8-update (λh. h(PTR-COERCE(8 signed word → 8 word) ?p := UCAST(8 signed → 8) (?a s))) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(32 word ptr) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-w32'ptr-update (λh. h(?p := ?a s)) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(unit ptr) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-unit'ptr-update (λh. h(?p := ?a s)) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(data-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-data-C-update (λh. h(?p := ?a s)) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(closed-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-closed-C-update (λh. h(?p := ?a s)) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(unpacked-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-unpacked-C-update (λh. h(?p := ?a s)) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(arr-struct-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-arr-struct-C-update (λh. h(?p := ?a s)) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. valid-array-base.valid-array (λh. IS-VALID(unpacked-C) h) t ?p ∧ ?P t)) (λ-. L2-modify (λs. unpacked-C.heap-array-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. valid-array-base.valid-array (valid-array-base.valid-array (λh. IS-VALID(unpacked-C) h)) t ?p ∧ ?P t)) (λ-. L2-modify (λs. un-

packed-C.outer.heap-array-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(two-dimensional-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-two-dimensional-C-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(data-struct1-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-data-struct1-C-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(data-struct2-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-data-struct2-C-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(inner-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-inner-C-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. valid-array-base.valid-array (λh. IS-VALID(inner-C) h) t ?p ∧ ?P t)) (λ-. L2-modify (λs. inner-C.heap-array-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(outer-array-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-outer-array-C-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. valid-array-base.valid-array (λh. IS-VALID(data-C) h) t ?p ∧ ?P t)) (λ-. L2-modify (λs. data-C.heap-array-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(data-array-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-data-array-C-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(outer-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-outer-C-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(other-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs.

heap-other-C-map ?p (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE ?p ?c*)

abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t. valid-array-base.valid-array* (λ*h. IS-VALID*(*unpacked-C*) *h*) *t ?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s. unpacked-C.heap-array-map ?p* (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE ?p ?c*)

abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t. IS-VALID*(*array-C*) *t ?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s. heap-array-C-map ?p* (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE ?p ?c*)

⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* ∧ *?Q s*)) (λ-. *heap-w32.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦⋀*s p. ?f$_c$ p* ∙ *s* ?{ λ*r. globals.typing.unchanged-typing-on* 𝒮 *s* }; *struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* ∧ *?Q s*)) (λ-. *heap-w32.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* ∧ *?Q s*)) (λ-. *heap-w8.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦⋀*s p. ?f$_c$ p* ∙ *s* ?{ λ*r. globals.typing.unchanged-typing-on* 𝒮 *s* }; *struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* ∧ *?Q s*)) (λ-. *heap-w8.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* ∧ *?Q s*)) (λ-. *heap-w32'ptr.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

$[\![\bigwedge s\ p.\ ?f_c\ p \cdot s\ ?\{\!|\ \lambda r.\ globals.typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ |\!\}; struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}w32'ptr.assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$[\![struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}unit'ptr.guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$[\![\bigwedge s\ p.\ ?f_c\ p \cdot s\ ?\{\!|\ \lambda r.\ globals.typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ |\!\}; struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}unit'ptr.assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$[\![struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}data\text{-}C.guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$[\![\bigwedge s\ p.\ ?f_c\ p \cdot s\ ?\{\!|\ \lambda r.\ globals.typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ |\!\}; struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}data\text{-}C.assume\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$[\![struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap\ (?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq\ (L2\text{-}guard\ (\lambda s.\ ?P'\ s \wedge ?Q\ s))\ (\lambda\text{-}.\ heap\text{-}closed\text{-}C.guard\text{-}with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_a\ (L2\text{-}VARS\ ?f_a\ ?nm)))\ (globals.with\text{-}fresh\text{-}stack\text{-}ptr\ ?n\ ?init_c\ (L2\text{-}VARS\ ?f_c\ ?nm))$

$[\![\bigwedge s\ p.\ ?f_c\ p \cdot s\ ?\{\!|\ \lambda r.\ globals.typing.unchanged\text{-}typing\text{-}on\ \mathcal{S}\ s\ |\!\}; struct\text{-}rewrite\text{-}expr\ ?P\ ?init_c'\ ?init_c; abs\text{-}guard\ lift\text{-}global\text{-}heap\ ?P'\ ?P; abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?Q\ ?init_a\ ?init_c'; THIN\ (\bigwedge p.\ L2Tcorres\ lift\text{-}global\text{-}heap$

$(?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow$ *L2Tcorres lift-global-heap (L2-seq (L2-guard* $(\lambda s.\ ?P'\ s \wedge ?Q\ s)$*) (λ-. heap-closed-C.assume-with-fresh-stack-ptr ?n* $?init_a$ *(L2-VARS* $?f_a$ *?nm))) (globals.with-fresh-stack-ptr ?n* $?init_c$ *(L2-VARS* $?f_c$ *?nm))*

$[\![$*struct-rewrite-expr ?P* $?init_c'$ $?init_c$*; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q* $?init_a$ $?init_c'$*; THIN (*$\bigwedge$*p. L2Tcorres lift-global-heap* $(?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow$ *L2Tcorres lift-global-heap (L2-seq (L2-guard* $(\lambda s.\ ?P'\ s \wedge ?Q\ s)$*) (λ-. heap-unpacked-C.guard-with-fresh-stack-ptr ?n* $?init_a$ *(L2-VARS* $?f_a$ *?nm))) (globals.with-fresh-stack-ptr ?n* $?init_c$ *(L2-VARS* $?f_c$ *?nm))*

$[\![\bigwedge s\ p.\ ?f_c\ p \cdot s\ ?\{\!|\ \lambda r.$ *globals.typing.unchanged-typing-on* $\mathcal{S}\ s\ |\!\}$*; struct-rewrite-expr ?P* $?init_c'$ $?init_c$*; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q* $?init_a$ $?init_c'$*; THIN (*$\bigwedge$*p. L2Tcorres lift-global-heap* $(?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow$ *L2Tcorres lift-global-heap (L2-seq (L2-guard* $(\lambda s.$ $?P'\ s \wedge ?Q\ s)$*) (λ-. heap-unpacked-C.assume-with-fresh-stack-ptr ?n* $?init_a$ *(L2-VARS* $?f_a$ *?nm))) (globals.with-fresh-stack-ptr ?n* $?init_c$ *(L2-VARS* $?f_c$ *?nm))*

$[\![$*struct-rewrite-expr ?P* $?init_c'$ $?init_c$*; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q* $?init_a$ $?init_c'$*; THIN (*$\bigwedge$*p. L2Tcorres lift-global-heap* $(?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow$ *L2Tcorres lift-global-heap (L2-seq (L2-guard* $(\lambda s.\ ?P'\ s \wedge ?Q\ s)$*) (λ-. heap-arr-struct-C.guard-with-fresh-stack-ptr ?n* $?init_a$ *(L2-VARS* $?f_a$ *?nm))) (globals.with-fresh-stack-ptr ?n* $?init_c$ *(L2-VARS* $?f_c$ *?nm))*

$[\![\bigwedge s\ p.\ ?f_c\ p \cdot s\ ?\{\!|\ \lambda r.$ *globals.typing.unchanged-typing-on* $\mathcal{S}\ s\ |\!\}$*; struct-rewrite-expr ?P* $?init_c'$ $?init_c$*; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q* $?init_a$ $?init_c'$*; THIN (*$\bigwedge$*p. L2Tcorres lift-global-heap* $(?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow$ *L2Tcorres lift-global-heap (L2-seq (L2-guard* $(\lambda s.$ $?P'\ s \wedge ?Q\ s)$*) (λ-. heap-arr-struct-C.assume-with-fresh-stack-ptr ?n* $?init_a$ *(L2-VARS* $?f_a$ *?nm))) (globals.with-fresh-stack-ptr ?n* $?init_c$ *(L2-VARS* $?f_c$ *?nm))*

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard $(\lambda t.$ *IS-VALID(unpacked-C*$[3][2])$ *t ?p* $\wedge$ *?P t)) (λ-. L2-modify* $(\lambda s.$ *unpacked-C.outer.heap-array-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)*

$[\![$*struct-rewrite-expr ?P* $?init_c'$ $?init_c$*; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q* $?init_a$ $?init_c'$*; THIN (*$\bigwedge$*p. L2Tcorres lift-global-heap* $(?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow$ *L2Tcorres lift-global-heap (L2-seq (L2-guard* $(\lambda s.$ $?P'\ s \wedge ?Q\ s)$*) (λ-. heap-unpacked-C'array-3'array-2.guard-with-fresh-stack-ptr ?n* $?init_a$ *(L2-VARS* $?f_a$ *?nm))) (globals.with-fresh-stack-ptr ?n* $?init_c$ *(L2-VARS* $?f_c$ *?nm))*

$[\![\bigwedge s\ p.\ ?f_c\ p \cdot s\ ?\{\!|\ \lambda r.$ *globals.typing.unchanged-typing-on* $\mathcal{S}\ s\ |\!\}$*; struct-rewrite-expr ?P* $?init_c'$ $?init_c$*; abs-guard lift-global-heap ?P' ?P;*

abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN ($\bigwedge$p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-unpacked-C'array-3'array-2.assume-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))

⟦struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN ($\bigwedge$p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-two-dimensional-C.guard-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))

⟦$\bigwedge$s p. ?f$_c$ p ∙ s ?{ λr. globals.typing.unchanged-typing-on $\mathcal{S}$ s }; struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN ($\bigwedge$p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-two-dimensional-C.assume-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))

⟦struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN ($\bigwedge$p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-data-struct1-C.guard-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))

⟦$\bigwedge$s p. ?f$_c$ p ∙ s ?{ λr. globals.typing.unchanged-typing-on $\mathcal{S}$ s }; struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN ($\bigwedge$p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-data-struct1-C.assume-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))

⟦struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN ($\bigwedge$p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-data-struct2-C.guard-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))

⟦$\bigwedge$s p. ?f$_c$ p ∙ s ?{ λr. globals.typing.unchanged-typing-on $\mathcal{S}$ s }; struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN ($\bigwedge$p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-data-struct2-C.assume-with-fresh-stack-ptr

?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))

⟦struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN (⋀p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-inner-C.guard-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))

⟦⋀s p. ?f$_c$ p ∙ s ?{ λr. globals.typing.unchanged-typing-on 𝒮 s }; struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN (⋀p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-inner-C.assume-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(inner-C[5]) t ?p ∧ ?P t)) (λ-. L2-modify (λs. inner-C.heap-array-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

⟦struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN (⋀p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-inner-C'array-5.guard-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))

⟦⋀s p. ?f$_c$ p ∙ s ?{ λr. globals.typing.unchanged-typing-on 𝒮 s }; struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN (⋀p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-inner-C'array-5.assume-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))

⟦struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN (⋀p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-outer-array-C.guard-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))

⟦⋀s p. ?f$_c$ p ∙ s ?{ λr. globals.typing.unchanged-typing-on 𝒮 s }; struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN (⋀p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs.

?P′ s ∧ ?Q s)) (λ-. *heap-outer-array-C.assume-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm)*)) (*globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm)*))

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (*L2-seq* (*L2-guard* (λ*t. IS-VALID*(*data-C*[*10*]) *t ?p ∧ ?P t*)) (λ-. *L2-modify* (λ*s. data-C.heap-array-map ?p* (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE ?p ?c*)

⟦*struct-rewrite-expr ?P ?init$_c$′ ?init$_c$*; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P′ s ∧ ?Q s*)) (λ-. *heap-data-C′array-10.guard-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm)*)) (*globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm)*))

⟦⋀*s p. ?f$_c$ p* ∙ *s ?*{ λ*r. globals.typing.unchanged-typing-on* 𝒮 *s* }; *struct-rewrite-expr ?P ?init$_c$′ ?init$_c$*; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P′ s ∧ ?Q s*)) (λ-. *heap-data-C′array-10.assume-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm)*)) (*globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm)*))

⟦*struct-rewrite-expr ?P ?init$_c$′ ?init$_c$*; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P′ s ∧ ?Q s*)) (λ-. *heap-data-array-C.guard-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm)*)) (*globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm)*))

⟦⋀*s p. ?f$_c$ p* ∙ *s ?*{ λ*r. globals.typing.unchanged-typing-on* 𝒮 *s* }; *struct-rewrite-expr ?P ?init$_c$′ ?init$_c$*; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P′ s ∧ ?Q s*)) (λ-. *heap-data-array-C.assume-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm)*)) (*globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm)*))

⟦*struct-rewrite-expr ?P ?init$_c$′ ?init$_c$*; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P′ s ∧ ?Q s*)) (λ-. *heap-outer-C.guard-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm)*)) (*globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm)*))

⟦⋀*s p. ?f$_c$ p* ∙ *s ?*{ λ*r. globals.typing.unchanged-typing-on* 𝒮 *s* }; *struct-rewrite-expr ?P ?init$_c$′ ?init$_c$*; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′*; *THIN* (⋀*p. L2Tcorres lift-global-heap*

$(?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow$ *L2Tcorres lift-global-heap (L2-seq (L2-guard* $(\lambda s.\ ?P'\ s \wedge ?Q\ s)$*) (λ-. heap-outer-C.assume-with-fresh-stack-ptr ?n* $?init_a$ *(L2-VARS* $?f_a$ *?nm))) (globals.with-fresh-stack-ptr ?n* $?init_c$ *(L2-VARS* $?f_c$ *?nm))*

$[\![$*struct-rewrite-expr ?P* $?init_c'$ $?init_c$*; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q* $?init_a$ $?init_c'$*; THIN (*$\bigwedge$*p. L2Tcorres lift-global-heap* $(?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow$ *L2Tcorres lift-global-heap (L2-seq (L2-guard* $(\lambda s.\ ?P'\ s \wedge ?Q\ s)$*) (λ-. heap-other-C.guard-with-fresh-stack-ptr ?n* $?init_a$ *(L2-VARS* $?f_a$ *?nm))) (globals.with-fresh-stack-ptr ?n* $?init_c$ *(L2-VARS* $?f_c$ *?nm))*

$[\![\bigwedge s\ p.\ ?f_c\ p \cdot s\ ?\{\!|\ \lambda r.$ *globals.typing.unchanged-typing-on* $\mathcal{S}\ s\ |\!\}$*; struct-rewrite-expr ?P* $?init_c'$ $?init_c$*; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q* $?init_a$ $?init_c'$*; THIN (*$\bigwedge$*p. L2Tcorres lift-global-heap* $(?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow$ *L2Tcorres lift-global-heap (L2-seq (L2-guard* $(\lambda s.\ ?P'\ s \wedge ?Q\ s)$*) (λ-. heap-other-C.assume-with-fresh-stack-ptr ?n* $?init_a$ *(L2-VARS* $?f_a$ *?nm))) (globals.with-fresh-stack-ptr ?n* $?init_c$ *(L2-VARS* $?f_c$ *?nm))*

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard $(\lambda t.$ *IS-VALID(unpacked-C[2]) t ?p* $\wedge$ *?P t))* $(\lambda$*-. L2-modify* $(\lambda s.$ *unpacked-C.heap-array-map ?p* $(\lambda$*-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)*

$[\![$*struct-rewrite-expr ?P* $?init_c'$ $?init_c$*; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q* $?init_a$ $?init_c'$*; THIN (*$\bigwedge$*p. L2Tcorres lift-global-heap* $(?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow$ *L2Tcorres lift-global-heap (L2-seq (L2-guard* $(\lambda s.$ $?P'\ s \wedge ?Q\ s)$*) (λ-. heap-unpacked-C'array-2.guard-with-fresh-stack-ptr ?n* $?init_a$ *(L2-VARS* $?f_a$ *?nm))) (globals.with-fresh-stack-ptr ?n* $?init_c$ *(L2-VARS* $?f_c$ *?nm))*

$[\![\bigwedge s\ p.\ ?f_c\ p \cdot s\ ?\{\!|\ \lambda r.$ *globals.typing.unchanged-typing-on* $\mathcal{S}\ s\ |\!\}$*; struct-rewrite-expr ?P* $?init_c'$ $?init_c$*; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q* $?init_a$ $?init_c'$*; THIN (*$\bigwedge$*p. L2Tcorres lift-global-heap* $(?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow$ *L2Tcorres lift-global-heap (L2-seq (L2-guard* $(\lambda s.$ $?P'\ s \wedge ?Q\ s)$*) (λ-. heap-unpacked-C'array-2.assume-with-fresh-stack-ptr ?n* $?init_a$ *(L2-VARS* $?f_a$ *?nm))) (globals.with-fresh-stack-ptr ?n* $?init_c$ *(L2-VARS* $?f_c$ *?nm))*

$[\![$*struct-rewrite-expr ?P* $?init_c'$ $?init_c$*; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q* $?init_a$ $?init_c'$*; THIN (*$\bigwedge$*p. L2Tcorres lift-global-heap* $(?f_a\ p)\ (?f_c\ p))]\!] \Longrightarrow$ *L2Tcorres lift-global-heap (L2-seq (L2-guard* $(\lambda s.\ ?P'\ s \wedge ?Q\ s)$*) (λ-. heap-array-C.guard-with-fresh-stack-ptr ?n* $?init_a$ *(L2-VARS* $?f_a$ *?nm))) (globals.with-fresh-stack-ptr ?n* $?init_c$ *(L2-VARS* $?f_c$ *?nm))*

$[\![\bigwedge s\ p.\ ?f_c\ p \cdot s\ ?\{\!|\ \lambda r.$ *globals.typing.unchanged-typing-on* $\mathcal{S}\ s\ |\!\}$*; struct-rewrite-expr ?P* $?init_c'$ $?init_c$*; abs-guard lift-global-heap ?P' ?P;*

abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN ($\bigwedge$p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-array-C.assume-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm)).

- The derived introduction rules are recursively applied to the program.

  The first step is the one that was significantly extended to support addressable fields in open structures. The second and third step remained almost unchanged.

thm *heap-abs*

When thoroughly inspecting the rules ⟦*abs-expr ?st ?X ?f1.0 ?f1'; abs-expr ?st ?Y ?f2.0 ?f2'*⟧ ⟹ *abs-expr ?st* (λs. *?X s ∧ ?Y s*) (λs. (*?f1.0 s*, *?f2.0 s*)) (λs. (*?f1' s*, *?f2' s*))

abs-expr ?st (λ-. *True*) (λs. *?a*) (λs. *?a*)

abs-expr ?st ?P ?a' ?a ⟹ *abs-guard ?st* (λs. *?P s ∧ ?a' s*) *?a*

abs-guard ?st (λ-. *?P*) (λ-. *?P*)

⟦*abs-guard ?st ?G ?G'; abs-guard ?st ?H ?H'*⟧ ⟹ *abs-guard ?st* (λs. *?G s ∧ ?H s*) (λs. *?G' s ∧ ?H' s*)

⟦*struct-rewrite-modifies ?P ?b ?c; abs-guard ?st ?P' ?P; abs-modifies ?st ?Q ?a ?b*⟧ ⟹ *L2Tcorres ?st* (*L2-seq* (*L2-guard* (λs. *?P' s ∧ ?Q s*)) (λ-. *L2-modify ?a*)) (*L2-modify ?c*)

⟦*struct-rewrite-expr ?P ?b ?c; abs-guard ?st ?P' ?P; abs-expr ?st ?Q ?a ?b*⟧ ⟹ *L2Tcorres ?st* (*L2-seq* (*L2-guard* (λs. *?P' s ∧ ?Q s*)) (λ-. *L2-gets ?a ?n*)) (*L2-gets ?c ?n*)

L2Tcorres ?st (*L2-gets* (λ-. *?a*) *?n*) (*L2-gets* (λ-. *?a*) *?n*)

⟦*struct-rewrite-guard ?b ?c; abs-guard ?st ?a ?b*⟧ ⟹ *L2Tcorres ?st* (*L2-guard ?a*) (*L2-guard ?c*)

⟦*THIN* ($\bigwedge$*x. L2Tcorres ?st* (*?B' x*) (*?B x*)); *THIN* ($\bigwedge$*r. struct-rewrite-expr* (*?G r*) (*?C' r*) (*?C r*)); *THIN* ($\bigwedge$*r. abs-guard ?st* (*?G' r*) (*?G r*)); *THIN* ($\bigwedge$*r. abs-expr ?st* (*?H r*) (*?C'' r*) (*?C' r*))⟧ ⟹ *L2Tcorres ?st* (*L2-seq* (*L2-guard* (λs. *?G' ?i s ∧ ?H ?i s*)) (λ-. *L2-while ?C''* (λi. *L2-seq* (*?B' i*) (λr. *L2-seq* (*L2-guard* (λs. *?G' r s ∧ ?H r s*)) (λ-. *L2-gets* (λ-. *r*) *?n*))) *?i ?n*)) (*L2-while ?C ?B ?i ?n*)

abs-spec ?st ?P ?A ?C ⟹ *L2Tcorres ?st* (*L2-seq* (*L2-guard ?P*) (λ-. *L2-spec ?A*)) (*L2-spec ?C*)

abs-assume ?st ?P ?A ?C ⟹ *L2Tcorres ?st* (*L2-seq* (*L2-guard ?P*) (λ-. *L2-assume ?A*)) (*L2-assume ?C*)

abs-spec ?st (λ-. *True*) {(*a*, *b*). *?C*} {(*a*, *b*). *?C*}

⟦*THIN* (*L2Tcorres ?st ?L ?L'*); *THIN* (*L2Tcorres ?st ?R ?R'*); *THIN* (*struct-rewrite-expr ?P ?C' ?C*); *THIN* (*abs-guard ?st ?P' ?P*); *THIN* (*abs-expr ?st ?Q ?C'' ?C'*)⟧ ⟹ *L2Tcorres ?st* (*L2-seq* (*L2-guard* (λs. *?P' s ∧ ?Q s*)) (λ-. *L2-condition ?C'' ?L ?R*)) (*L2-condition ?C ?L' ?R'*)

⟦*THIN* (*L2Tcorres ?st ?L′ ?L*); *THIN* (⋀*r*. *L2Tcorres ?st* (*?R′ r*) (*?R r*))⟧ ⟹ *L2Tcorres ?st* (*L2-seq ?L′ ?R′*) (*L2-seq ?L ?R*)

⟦*struct-rewrite-guard ?b ?c*; *abs-guard ?st ?a ?b*; ⋀*s s′*. ⟦*?c s*; *s′* = *?st s*⟧ ⟹ *L2Tcorres ?st ?f ?g*⟧ ⟹ *L2Tcorres ?st* (*L2-guarded ?a ?f*) (*L2-guarded ?c ?g*)

⟦*THIN* (*L2Tcorres ?st ?L ?L′*); *THIN* (⋀*r*. *L2Tcorres ?st* (*?R r*) (*?R′ r*))⟧ ⟹ *L2Tcorres ?st* (*L2-catch ?L ?R*) (*L2-catch ?L′ ?R′*)

L2Tcorres ?st ?L ?L′ ⟹ *L2Tcorres ?st* (*L2-try ?L*) (*L2-try ?L′*)

L2Tcorres ?st (*L2-unknown ?ns*) (*L2-unknown ?ns*)

L2Tcorres ?st (*L2-throw ?x ?n*) (*L2-throw ?x ?n*)

(⋀*x y*. *L2Tcorres ?st* (*?P x y*) (*?P′ x y*)) ⟹ *L2Tcorres ?st* (*case ?a of* (*x*, *y*) ⇒ *?P x y*) (*case ?a of* (*x*, *y*) ⇒ *?P′ x y*)

⟦*THIN* (*L2Tcorres ?st ?L ?L′*); *THIN* (*L2Tcorres ?st ?R ?R′*)⟧ ⟹ *L2Tcorres ?st* (*L2-seq ?L* (λ-. *?R*)) (*L2-seq ?L′* (λ-. *?R′*))

(⋀*a b*. *abs-expr ?st* (*?P a b*) (*?A a b*) (*?C a b*)) ⟹ *abs-expr ?st* (*case ?r of* (*a*, *b*) ⇒ *?P a b*) (*case ?r of* (*a*, *b*) ⇒ *?A a b*) (*case ?r of* (*a*, *b*) ⇒ *?C a b*)

(⋀*a b*. *abs-guard ?st* (*?A a b*) (*?C a b*)) ⟹ *abs-guard ?st* (*case ?r of* (*a*, *b*) ⇒ *?A a b*) (*case ?r of* (*a*, *b*) ⇒ *?C a b*)

L2Tcorres ?st L2-fail L2-fail

abs-expr id (λ-. *True*) *?A ?A*

abs-expr ?st ?P ?A ?C ⟹ *abs-expr ?st ?P* (λ*s r*. *?A s*) (λ*s r*. *?C s*)

abs-modifies id (λ-. *True*) *?A ?A*

L2Tcorres id ?A ?C ⟹ *L2Tcorres ?st* (*exec-concrete ?st* (*L2-call ?A ?emb ?ns*)) (*L2-call ?C ?emb ?ns*)

L2Tcorres id ?A ?C ⟹ *L2Tcorres ?st* (*exec-concrete ?st* (*L2-call-L1 ?arg-xf ?gs ?ret-xf ?A*)) (*L2-call-L1 ?arg-xf ?gs ?ret-xf ?C*)

L2Tcorres ?st ?A ?C ⟹ *L2Tcorres id* (*exec-abstract ?st* (*L2-call ?A ?emb ?ns*)) (*L2-call ?C ?emb ?ns*)

L2Tcorres ?st ?A ?C ⟹ *L2Tcorres ?st* (*L2-call ?A ?emb ?ns*) (*L2-call ?C ?emb ?ns*)

⟦*typ-heap-simulation ?st ?getter ?setter ?vgetter ?t-hrs ?t-hrs-update*; *abs-expr ?st ?P ?x′ ?x*⟧ ⟹ *abs-guard ?st* (λ*s*. *?P s* ∧ *?vgetter s* (*?x′ s*)) (λ*s*. *c-guard* (*?x s*))

⟦*typ-heap-simulation ?st ?r ?w ?v ?t-hrs ?t-hrs-update*; *abs-expr ?st ?P ?x′ ?x*⟧ ⟹ *abs-expr ?st* (λ*s*. *?P s* ∧ *?v s* (*?x′ s*)) (λ*s*. *?r s* (*?x′ s*)) (λ*s*. *h-val* (*hrs-mem* (*?t-hrs s*)) (*?x s*))

⟦*typ-heap-simulation ?st ?r ?w ?v ?t-hrs ?t-hrs-update*; *abs-expr ?st ?Pb ?b′ ?b*; ⋀*v*. *abs-expr ?st ?Pc* (*?c′ v*) (*?c v*)⟧ ⟹ *abs-modifies ?st* (λ*s*. *?Pb s* ∧ *?Pc s* ∧ *?v s* (*?b′ s*)) (λ*s*. *?w* (*?b′ s*) (λ-. *?c′* (*?r s* (*?b′ s*)) *s*) *s*) (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?b s*) (*?c* (*heap-lift--h-val* (*hrs-mem* (*?t-hrs s*)) (*?b s*)) *s*))) *s*)

struct-rewrite-expr ?P ?a′ ?a ⟹ *struct-rewrite-guard* (λ*s*. *?P s* ∧ *?a′ s*) *?a*

struct-rewrite-guard (λ-. *?P*) (λ-. *?P*)

⟦*struct-rewrite-guard ?b′ ?b*; *struct-rewrite-guard ?a′ ?a*⟧ ⟹ *struct-rewrite-guard* (λ*s*. *?a′ s* ∧ *?b′ s*) (λ*s*. *?a s* ∧ *?b s*)

(⋀*a b*. *struct-rewrite-guard* (*?A a b*) (*?C a b*)) ⟹ *struct-rewrite-guard* (*case ?r of* (*a*, *b*) ⇒ *?A a b*) (*case ?r of* (*a*, *b*) ⇒ *?C a b*)

⟦*valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update*; *struct-rewrite-expr ?P ?p′ ?p*; *struct-rewrite-guard ?Q* (λ*s*. *c-guard* (*?p′ s*))⟧ ⟹ *struct-rewrite-guard* (λ*s*. *?P s* ∧ *?Q s*) (λ*s*. *c-guard* (*PTR*(*?′f*) &(*?p s*→*?field-name*)))

⟦*valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update*; *struct-rewrite-expr ?P ?p′ ?p*; *struct-rewrite-guard ?Q* (λ*s*. *c-guard* (*?p′ s*))⟧ ⟹ *struct-rewrite-guard* (λ*s*. *?P s* ∧ *?Q s* ∧ 0 ≤ *?k* ∧ *nat ?k* < *CARD*(*?′n*)) (λ*s*. *c-guard* (*PTR-COERCE*(*?′f*[*?′n*] → *?′f*) (*PTR*(*?′f*[*?′n*]) &(*?p s*→*?field-name*)) +$_p$ *?k*))

struct-rewrite-expr (λ-. *True*) (λ-. *?a*) (λ-. *?a*)

struct-rewrite-expr ?P ?A ?C ⟹ *struct-rewrite-expr ?P* (λ*s* -. *?A s*) (λ*s* -. *?C s*)

(⋀*a b*. *struct-rewrite-expr* (*?P a b*) (*?A a b*) (*?C a b*)) ⟹ *struct-rewrite-expr* (*case ?r of* (*a*, *b*) ⇒ *?P a b*) (*case ?r of* (*a*, *b*) ⇒ *?A a b*) (*case ?r of* (*a*, *b*) ⇒ *?C a b*)

struct-rewrite-expr (λ-. *True*) (λ*s*. *h-val* (*?h s*) (*?p s*)) (λ*s*. *h-val* (*?h s*) (*?p s*))

struct-rewrite-expr ?P ?a ?c ⟹ *struct-rewrite-expr ?P* (λ*s*. *h-val* (*?h s*) (*?a s*)) (λ*s*. *h-val* (*?h s*) (*?c s*))

⟦*valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update*; *struct-rewrite-expr ?P ?p′ ?p*; *struct-rewrite-expr ?Q ?a* (λ*s*. *h-val* (*hrs-mem* (*?t-hrs s*)) (*?p′ s*))⟧ ⟹ *struct-rewrite-expr* (λ*s*. *?P s* ∧ *?Q s*) (λ*s*. *?field-getter* (*?a s*)) (λ*s*. *h-val* (*hrs-mem* (*?t-hrs s*)) (*PTR*(*?′f*) &(*?p s*→*?field-name*)))

⟦*valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update*; *abs-expr ?st ?P ?a ?c*⟧ ⟹ *abs-expr ?st ?P* (λ*s*. *?field-getter* (*?a s*)) (λ*s*. *?field-getter* (*?c s*))

⟦*valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update*; *struct-rewrite-expr ?P ?p′ ?p*; *struct-rewrite-expr ?Q ?a* (λ*s*. *h-val* (*hrs-mem* (*?t-hrs s*)) (*?p′ s*))⟧ ⟹ *struct-rewrite-expr* (λ*s*. *?P s* ∧ *?Q s* ∧ 0 ≤ *?k* ∧ *nat ?k* < *CARD*(*?′n*)) (λ*s*. *?field-getter* (*?a s*).[*nat ?k*]) (λ*s*. *h-val* (*hrs-mem* (*?t-hrs s*)) (*PTR-COERCE*(*?′f*[*?′n*] → *?′f*) (*PTR*(*?′f*[*?′n*]) &(*?p s*→*?field-name*)) +$_p$ *?k*))

⟦*valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update*; *struct-rewrite-expr ?P ?p′ ?p*; *struct-rewrite-expr ?Q ?a* (λ*s*. *h-val* (*hrs-mem* (*?t-hrs s*)) (*?p′ s*))⟧ ⟹ *struct-rewrite-expr* (λ*s*. *?P s* ∧ *?Q s* ∧ 0 ≤ *?k* ∧ *nat ?k* < *CARD*(*?′n*)) (λ*s*. *?field-getter* (*?a s*).[*nat ?k*]) (λ*s*. *h-val* (*hrs-mem* (*?t-hrs s*)) (*PTR*(*?′f*) &(*?p s*→*?field-name*) +$_p$ *?k*))

struct-rewrite-modifies (λ-. *True*) *?A ?A*

⟦*typ-heap-simulation ?st ?getter ?setter ?vgetter ?t-hrs ?t-hrs-update*;

struct-rewrite-expr ?P ?p′ ?p; *struct-rewrite-expr ?Q ?v′ ?v*⟧ ⟹ *struct-rewrite-modifies* (λ*s*. *?P s* ∧ *?Q s*) (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?p′ s*) (*?v′ s*))) *s*) (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?p s*) (*?v s*))) *s*)

heap-lift--wrap-h-val (*heap-lift--h-val ?s ?p*) (*h-val ?s ?p*)

heap-lift--wrap-h-val (*h-val ?s* (*PTR*(*?′a*) &(*?p*→*?f*))) (*h-val ?s* (*PTR*(*?′a*) &(*?p*→*?f*)))

heap-lift--wrap-h-val (*h-val ?s* (*PTR-COERCE*(*?′b* → *?′a*) *?p* +*p ?k*)) (*h-val ?s* (*PTR-COERCE*(*?′b* → *?′a*) *?p* +*p ?k*))

⟦*valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update*; *struct-rewrite-expr ?P ?p′ ?p*; *struct-rewrite-expr ?Q ?f′ ?f*; ⋀*s*. *heap-lift--wrap-h-val* (*?h-val-p′ s*) (*h-val* (*hrs-mem* (*?t-hrs s*)) (*?p′ s*)); *struct-rewrite-modifies ?R* (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?p″ s*) (*?u s* (*?field-setter* (*?f′ s*))))) *s*) (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?p′ s*) (*?field-setter* (*?f′ s*) (*?h-val-p′ s*)))) *s*); *struct-rewrite-guard ?S* (λ*s*. *c-guard* (*?p′ s*))⟧ ⟹ *struct-rewrite-modifies* (λ*s*. *?P s* ∧ *?Q s* ∧ *?R s* ∧ *?S s*) (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?p″ s*) (*?u s* (*?field-setter* (*?f′ s*))))) *s*) (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*PTR*(*?′f*) &(*?p s*→*?field-name*)) (*?f s* (*h-val* (*hrs-mem* (*?t-hrs s*)) (*PTR*(*?′f*) &(*?p s*→*?field-name*)))))) *s*)

⟦*valid-struct-field ?field-name ?field-getter ?field-setter ?t-hrs ?t-hrs-update*; *struct-rewrite-expr ?P ?p′ ?p*; *struct-rewrite-expr ?Q ?f′ ?f*; ⋀*s*. *heap-lift--wrap-h-val* (*?h-val-p′ s*) (*h-val* (*hrs-mem* (*?t-hrs s*)) (*?p′ s*)); *struct-rewrite-modifies ?R* (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?p″ s*) (*?u s* (*?field-setter* (λ*a*. *Arrays.update a* (*nat ?k*) (*?f′ s* (*a*.[*nat ?k*])))))) *s*) (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?p′ s*) (*?field-setter* (λ*a*. *Arrays.update a* (*nat ?k*) (*?f′ s* (*a*.[*nat ?k*]))) (*?h-val-p′ s*)))) *s*); *struct-rewrite-guard ?S* (λ*s*. *c-guard* (*?p′ s*))⟧ ⟹ *struct-rewrite-modifies* (λ*s*. *?P s* ∧ *?Q s* ∧ *?R s* ∧ *?S s* ∧ 0 ≤ *?k* ∧ *nat ?k* < *CARD*(*?′n*)) (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*?p″ s*) (*?u s* (*?field-setter* (λ*a*. *Arrays.update a* (*nat ?k*) (*?f′ s* (*a*.[*nat ?k*])))))) *s*) (λ*s*. *?t-hrs-update* (*hrs-mem-update* (*heap-update* (*PTR-COERCE*(*?′f*[*?′n*] → *?′f*) (*PTR*(*?′f*[*?′n*]) &(*?p s*→*?field-name*)) +*p ?k*) (*?f s* (*h-val* (*hrs-mem* (*?t-hrs s*)) (*PTR-COERCE*(*?′f*[*?′n*] → *?′f*) (*PTR*(*?′f*[*?′n*]) &(*?p s*→*?field-name*)) +*p ?k*))))) *s*)

valid-globals-field ?st ?old-getter ?old-setter ?new-getter ?new-setter ⟹ *abs-expr ?st* (λ*-*. *True*) *?new-getter ?old-getter*

⟦*valid-globals-field ?st ?old-getter ?old-setter ?new-getter ?new-setter*; ⋀*old*. *abs-expr ?st* (*?P old*) (*?v old*) (*?v′ old*)⟧ ⟹ *abs-modifies ?st* (λ*s*. ∀ *old*. *?P old s*) (λ*s*. *?new-setter* (λ*old*. *?v old s*) *s*) (λ*s*. *?old-setter* (λ*old*. *?v′ old s*) *s*)

valid-globals-field ?st ?old-getter ?old-setter ?new-getter ?new-setter ⟹ *struct-rewrite-expr* (λ*-*. *True*) *?old-getter ?old-getter*

⟦*valid-globals-field ?st ?old-getter ?old-setter ?new-getter ?new-setter*; ⋀*old*. *struct-rewrite-expr* (*?P old*) (*?v′ old*) (*?v old*)⟧ ⟹ *struct-rewrite-modifies* (λ*s*. ∀ *old*. *?P old s*) (λ*s*. *?old-setter* (λ*old*. *?v′ old s*) *s*) (λ*s*. *?old-setter*

(λold. ?v old s) s)

abs-spec ?st (λ-. True) {(a, b). b may-not-modify-globals a} {(a, b). b may-not-modify-globals a}

⟦valid-globals-field ?st ?old-getter ?old-setter ?new-getter ?new-setter; abs-spec ?st (λ-. True) {(a, b). ?C a b} {(a, b). ?C′ a b}⟧ ⟹ abs-spec ?st (λ-. True) {(a, b). mex (λx. ?C (?new-setter (λ-. x) a) b)} {(a, b). mex (λx. ?C′ (?old-setter (λ-. x) a) b)}

⟦typ-heap-simulation ?st ?r ?w ?v ?t-hrs ?t-hrs-update; abs-expr ?st ?P ?x′ (λs. PTR-COERCE(?′a[?′b] → ?′a) (?x s))⟧ ⟹ abs-guard ?st (λs. ?P s ∧ (∀ a∈set (array-addrs (?x′ s) CARD(?′b)). ?v s a)) (λs. c-guard (?x s))

⟦typ-heap-simulation ?st ?r ?w ?v ?t-hrs ?t-hrs-update; abs-expr ?st ?Pb ?b′ ?b; abs-expr ?st ?Pn ?n′ ?n; abs-expr ?st ?Pv ?y′ ?y⟧ ⟹ abs-modifies ?st (λs. ?Pb s ∧ ?Pn s ∧ ?Pv s ∧ ?n′ s < CARD(?′b) ∧ (∀ ptr∈set (array-addrs (PTR-COERCE(?′a[?′b] → ?′a) (?b′ s)) CARD(?′b)). ?v s ptr)) (λs. ?w (PTR-COERCE(?′a[?′b] → ?′a) (?b′ s) +$_p$ int (?n′ s)) (λv. ?y′ s) s) (λs. ?t-hrs-update (hrs-mem-update (heap-update (?b s) (Arrays.update (h-val (hrs-mem (?t-hrs s)) (?b s)) (?n s) (?y s)))) s)

⟦typ-heap-simulation ?st ?r ?w ?v ?t-hrs ?t-hrs-update; abs-expr ?st ?Pb ?b′ ?b; abs-expr ?st ?Pn ?n′ ?n⟧ ⟹ abs-expr ?st (λs. ?Pb s ∧ ?Pn s ∧ ?n′ s < CARD(?′b) ∧ ?v s (PTR-COERCE(?′a[?′b] → ?′a) (?b′ s) +$_p$ int (?n′ s))) (λs. ?r s (PTR-COERCE(?′a[?′b] → ?′a) (?b′ s) +$_p$ int (?n′ s))) (λs. h-val (hrs-mem (?t-hrs s)) (?b s).[?n s])

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(32 word) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-w32-update (λh. h(?p := ?a s)) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(32 word) t (PTR-COERCE(32 signed word → 32 word) ?p) ∧ ?P t)) (λ-. L2-modify (λs. heap-w32-update (λh. h(PTR-COERCE(32 signed word → 32 word) ?p := UCAST(32 signed → 32) (?a s))) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(8 word) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-w8-update (λh. h(?p := ?a s)) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(8 word) t (PTR-COERCE(8 signed word → 8 word) ?p) ∧ ?P t)) (λ-. L2-modify (λs. heap-w8-update (λh. h(PTR-COERCE(8 signed word → 8 word) ?p := UCAST(8 signed → 8) (?a s))) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(32 word ptr) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-w32′ptr-update (λh. h(?p := ?a s)) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(unit ptr) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-unit′ptr-update

$(\lambda h.\ h(?p := ?a\ s))\ s)))\ (globals.IO\text{-}modify\text{-}heap\text{-}paddingE\ ?p\ ?c)$
$abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?P\ ?a\ ?c \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq$
$(L2\text{-}guard\ (\lambda t.\ IS\text{-}VALID(data\text{-}C)\ t\ ?p \wedge ?P\ t))\ (\lambda\text{-}.\ L2\text{-}modify\ (\lambda s.\ heap\text{-}data\text{-}C\text{-}update$
$(\lambda h.\ h(?p := ?a\ s))\ s)))\ (globals.IO\text{-}modify\text{-}heap\text{-}paddingE\ ?p\ ?c)$
$abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?P\ ?a\ ?c \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq$
$(L2\text{-}guard\ (\lambda t.\ IS\text{-}VALID(closed\text{-}C)\ t\ ?p \wedge ?P\ t))\ (\lambda\text{-}.\ L2\text{-}modify\ (\lambda s.\ heap\text{-}closed\text{-}C\text{-}update$
$(\lambda h.\ h(?p := ?a\ s))\ s)))\ (globals.IO\text{-}modify\text{-}heap\text{-}paddingE\ ?p\ ?c)$
$abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?P\ ?a\ ?c \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq$
$(L2\text{-}guard\ (\lambda t.\ IS\text{-}VALID(unpacked\text{-}C)\ t\ ?p \wedge ?P\ t))\ (\lambda\text{-}.\ L2\text{-}modify\ (\lambda s.$
$heap\text{-}unpacked\text{-}C\text{-}update\ (\lambda h.\ h(?p := ?a\ s))\ s)))\ (globals.IO\text{-}modify\text{-}heap\text{-}paddingE$
$?p\ ?c)$
$abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?P\ ?a\ ?c \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq$
$(L2\text{-}guard\ (\lambda t.\ IS\text{-}VALID(arr\text{-}struct\text{-}C)\ t\ ?p \wedge ?P\ t))\ (\lambda\text{-}.\ L2\text{-}modify\ (\lambda s.$
$heap\text{-}arr\text{-}struct\text{-}C\text{-}update\ (\lambda h.\ h(?p := ?a\ s))\ s)))\ (globals.IO\text{-}modify\text{-}heap\text{-}paddingE$
$?p\ ?c)$
$abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?P\ ?a\ ?c \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq$
$(L2\text{-}guard\ (\lambda t.\ valid\text{-}array\text{-}base.valid\text{-}array\ (\lambda h.\ IS\text{-}VALID(unpacked\text{-}C)\ h)$
$t\ ?p \wedge ?P\ t))\ (\lambda\text{-}.\ L2\text{-}modify\ (\lambda s.\ unpacked\text{-}C.heap\text{-}array\text{-}map\ ?p\ (\lambda\text{-}.\ ?a\ s)$
$s)))\ (globals.IO\text{-}modify\text{-}heap\text{-}paddingE\ ?p\ ?c)$
$abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?P\ ?a\ ?c \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq$
$(L2\text{-}guard\ (\lambda t.\ valid\text{-}array\text{-}base.valid\text{-}array\ (valid\text{-}array\text{-}base.valid\text{-}array\ (\lambda h.$
$IS\text{-}VALID(unpacked\text{-}C)\ h))\ t\ ?p \wedge ?P\ t))\ (\lambda\text{-}.\ L2\text{-}modify\ (\lambda s.\ unpacked\text{-}C.outer.heap\text{-}array\text{-}map$
$?p\ (\lambda\text{-}.\ ?a\ s)\ s)))\ (globals.IO\text{-}modify\text{-}heap\text{-}paddingE\ ?p\ ?c)$
$abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?P\ ?a\ ?c \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq$
$(L2\text{-}guard\ (\lambda t.\ IS\text{-}VALID(two\text{-}dimensional\text{-}C)\ t\ ?p \wedge ?P\ t))\ (\lambda\text{-}.\ L2\text{-}modify$
$(\lambda s.\ heap\text{-}two\text{-}dimensional\text{-}C\text{-}map\ ?p\ (\lambda\text{-}.\ ?a\ s)\ s)))\ (globals.IO\text{-}modify\text{-}heap\text{-}paddingE$
$?p\ ?c)$
$abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?P\ ?a\ ?c \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq$
$(L2\text{-}guard\ (\lambda t.\ IS\text{-}VALID(data\text{-}struct1\text{-}C)\ t\ ?p \wedge ?P\ t))\ (\lambda\text{-}.\ L2\text{-}modify\ (\lambda s.$
$heap\text{-}data\text{-}struct1\text{-}C\text{-}map\ ?p\ (\lambda\text{-}.\ ?a\ s)\ s)))\ (globals.IO\text{-}modify\text{-}heap\text{-}paddingE$
$?p\ ?c)$
$abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?P\ ?a\ ?c \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq$
$(L2\text{-}guard\ (\lambda t.\ IS\text{-}VALID(data\text{-}struct2\text{-}C)\ t\ ?p \wedge ?P\ t))\ (\lambda\text{-}.\ L2\text{-}modify\ (\lambda s.$
$heap\text{-}data\text{-}struct2\text{-}C\text{-}map\ ?p\ (\lambda\text{-}.\ ?a\ s)\ s)))\ (globals.IO\text{-}modify\text{-}heap\text{-}paddingE$
$?p\ ?c)$
$abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?P\ ?a\ ?c \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq$
$(L2\text{-}guard\ (\lambda t.\ IS\text{-}VALID(inner\text{-}C)\ t\ ?p \wedge ?P\ t))\ (\lambda\text{-}.\ L2\text{-}modify\ (\lambda s.\ heap\text{-}inner\text{-}C\text{-}map$
$?p\ (\lambda\text{-}.\ ?a\ s)\ s)))\ (globals.IO\text{-}modify\text{-}heap\text{-}paddingE\ ?p\ ?c)$
$abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?P\ ?a\ ?c \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq$
$(L2\text{-}guard\ (\lambda t.\ valid\text{-}array\text{-}base.valid\text{-}array\ (\lambda h.\ IS\text{-}VALID(inner\text{-}C)\ h)\ t$
$?p \wedge ?P\ t))\ (\lambda\text{-}.\ L2\text{-}modify\ (\lambda s.\ inner\text{-}C.heap\text{-}array\text{-}map\ ?p\ (\lambda\text{-}.\ ?a\ s)\ s)))$
$(globals.IO\text{-}modify\text{-}heap\text{-}paddingE\ ?p\ ?c)$
$abs\text{-}expr\ lift\text{-}global\text{-}heap\ ?P\ ?a\ ?c \Longrightarrow L2Tcorres\ lift\text{-}global\text{-}heap\ (L2\text{-}seq$
$(L2\text{-}guard\ (\lambda t.\ IS\text{-}VALID(outer\text{-}array\text{-}C)\ t\ ?p \wedge ?P\ t))\ (\lambda\text{-}.\ L2\text{-}modify\ (\lambda s.$

heap-outer-array-C-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. valid-array-base.valid-array (λh. IS-VALID(data-C) h) t ?p ∧ ?P t)) (λ-. L2-modify (λs. data-C.heap-array-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(data-array-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-data-array-C-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(outer-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-outer-C-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(other-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-other-C-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. valid-array-base.valid-array (λh. IS-VALID(unpacked-C) h) t ?p ∧ ?P t)) (λ-. L2-modify (λs. unpacked-C.heap-array-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(array-C) t ?p ∧ ?P t)) (λ-. L2-modify (λs. heap-array-C-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

⟦struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN (⋀p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-w32.guard-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))

⟦⋀s p. ?f$_c$ p ∙ s ?{ λr. globals.typing.unchanged-typing-on 𝒮 s }; struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN (⋀p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-w32.assume-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))

⟦struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN (⋀p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-w8.guard-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm))

⟦⋀s p. ?f$_c$ p ∙ s ?{ λr. globals.typing.unchanged-typing-on 𝒮 s }; struct-rewrite-expr ?P ?init$_c$' ?init$_c$; abs-guard lift-global-heap ?P' ?P; abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'; THIN (⋀p. L2Tcorres lift-global-heap (?f$_a$ p) (?f$_c$ p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P' s ∧ ?Q s)) (λ-. heap-w8.assume-with-fresh-stack-ptr ?n ?init$_a$ (L2-VARS ?f$_a$?nm))) (globals.with-fresh-stack-ptr

?n ?init$_c$ (L2-VARS ?f$_c$?nm))

⟦*struct-rewrite-expr ?P ?init$_c$′ ?init$_c$*; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P′ s* ∧ *?Q s*)) (λ*-. heap-w32′ptr.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦⋀*s p. ?f$_c$ p* ⋅ *s ?*{ λ*r. globals.typing.unchanged-typing-on* 𝒮 *s* }; *struct-rewrite-expr ?P ?init$_c$′ ?init$_c$*; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P′ s* ∧ *?Q s*)) (λ*-. heap-w32′ptr.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦*struct-rewrite-expr ?P ?init$_c$′ ?init$_c$*; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P′ s* ∧ *?Q s*)) (λ*-. heap-unit′ptr.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦⋀*s p. ?f$_c$ p* ⋅ *s ?*{ λ*r. globals.typing.unchanged-typing-on* 𝒮 *s* }; *struct-rewrite-expr ?P ?init$_c$′ ?init$_c$*; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P′ s* ∧ *?Q s*)) (λ*-. heap-unit′ptr.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦*struct-rewrite-expr ?P ?init$_c$′ ?init$_c$*; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P′ s* ∧ *?Q s*)) (λ*-. heap-data-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦⋀*s p. ?f$_c$ p* ⋅ *s ?*{ λ*r. globals.typing.unchanged-typing-on* 𝒮 *s* }; *struct-rewrite-expr ?P ?init$_c$′ ?init$_c$*; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P′ s* ∧ *?Q s*)) (λ*-. heap-data-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦*struct-rewrite-expr ?P ?init$_c$′ ?init$_c$*; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P′ s* ∧ *?Q s*)) (λ*-. heap-closed-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦⋀*s p. ?f$_c$ p* ⋅ *s ?*{ λ*r. globals.typing.unchanged-typing-on* 𝒮 *s* }; *struct-rewrite-expr ?P ?init$_c$′ ?init$_c$*; *abs-guard lift-global-heap ?P′ ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$′*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P′ s* ∧ *?Q s*)) (λ*-. heap-closed-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*)))

(*globals.with-fresh-stack-ptr ?n ?init$_c$ (L2-VARS ?f$_c$?nm)*)
⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* ∧ *?Q s*)) (λ*-. heap-unpacked-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))
⟦⋀*s p. ?f$_c$ p* ∙ *s ?*{ λ*r. globals.typing.unchanged-typing-on* 𝒮 *s* }; *struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* ∧ *?Q s*)) (λ*-. heap-unpacked-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))
⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* ∧ *?Q s*)) (λ*-. heap-arr-struct-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))
⟦⋀*s p. ?f$_c$ p* ∙ *s ?*{ λ*r. globals.typing.unchanged-typing-on* 𝒮 *s* }; *struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* ∧ *?Q s*)) (λ*-. heap-arr-struct-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))
abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t. IS-VALID*(*unpacked-C*[*3*][*2*]) *t ?p* ∧ *?P t*)) (λ*-. L2-modify* (λ*s. unpacked-C.outer.heap-array-map ?p* (λ*-. ?a s*) *s*))) (*globals.IO-modify-heap-paddingE ?p ?c*)
⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* ∧ *?Q s*)) (λ*-. heap-unpacked-C'array-3'array-2.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))
⟦⋀*s p. ?f$_c$ p* ∙ *s ?*{ λ*r. globals.typing.unchanged-typing-on* 𝒮 *s* }; *struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* ∧ *?Q s*)) (λ*-. heap-unpacked-C'array-3'array-2.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))
⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* ∧ *?Q s*)) (λ*-. heap-two-dimensional-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$*

?nm))

⟦$\bigwedge$*s p. ?f$_c$ p* ⋅ *s ?*{ *λr. globals.typing.unchanged-typing-on S s* }; *struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (*⋀p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (*λs. ?P' s ∧ ?Q s*)) (*λ-. heap-two-dimensional-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (*⋀p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (*λs. ?P' s ∧ ?Q s*)) (*λ-. heap-data-struct1-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦$\bigwedge$*s p. ?f$_c$ p* ⋅ *s ?*{ *λr. globals.typing.unchanged-typing-on S s* }; *struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (*⋀p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (*λs. ?P' s ∧ ?Q s*)) (*λ-. heap-data-struct1-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (*⋀p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (*λs. ?P' s ∧ ?Q s*)) (*λ-. heap-data-struct2-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦$\bigwedge$*s p. ?f$_c$ p* ⋅ *s ?*{ *λr. globals.typing.unchanged-typing-on S s* }; *struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (*⋀p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (*λs. ?P' s ∧ ?Q s*)) (*λ-. heap-data-struct2-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (*⋀p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (*λs. ?P' s ∧ ?Q s*)) (*λ-. heap-inner-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦$\bigwedge$*s p. ?f$_c$ p* ⋅ *s ?*{ *λr. globals.typing.unchanged-typing-on S s* }; *struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (*⋀p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (*λs. ?P' s ∧ ?Q s*)) (*λ-. heap-inner-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (*λt. IS-VALID*(*inner-C*[*5*]) *t ?p ∧ ?P t*)) (*λ-. L2-modify* (*λs. inner-C.heap-array-map ?p* (*λ-. ?a s*) *s*))) (*globals.IO-modify-heap-paddingE ?p ?c*)

⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-inner-C'array-5.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦⋀*s p*. *?f$_c$ p* ⋅ *s ?*{ λ*r*. *globals.typing.unchanged-typing-on* $\mathcal{S}$ *s* }; *struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-inner-C'array-5.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-outer-array-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦⋀*s p*. *?f$_c$ p* ⋅ *s ?*{ λ*r*. *globals.typing.unchanged-typing-on* $\mathcal{S}$ *s* }; *struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-outer-array-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

abs-expr lift-global-heap ?P ?a ?c ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*t*. *IS-VALID*(*data-C*[*10*]) *t ?p* ∧ *?P t*)) (λ-. *L2-modify* (λ*s*. *data-C.heap-array-map ?p* (λ-. *?a s*) *s*))) (*globals.IO-modify-heap-paddingE ?p ?c*)

⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-data-C'array-10.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦⋀*s p*. *?f$_c$ p* ⋅ *s ?*{ λ*r*. *globals.typing.unchanged-typing-on* $\mathcal{S}$ *s* }; *struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-data-C'array-10.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* (⋀*p*. *L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))⟧ ⟹ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s*. *?P' s* ∧ *?Q s*)) (λ-. *heap-data-array-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))

⟦⋀s p. ?f_c p ⋅ s ?{ λr. globals.typing.unchanged-typing-on $\mathcal{S}$ s }; struct-rewrite-expr ?P ?$init_c$′ ?$init_c$; abs-guard lift-global-heap ?P′ ?P; abs-expr lift-global-heap ?Q ?$init_a$?$init_c$′; THIN (⋀p. L2Tcorres lift-global-heap (?f_a p) (?f_c p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P′ s ∧ ?Q s)) (λ-. heap-data-array-C.assume-with-fresh-stack-ptr ?n ?$init_a$ (L2-VARS ?f_a ?nm))) (globals.with-fresh-stack-ptr ?n ?$init_c$ (L2-VARS ?f_c ?nm))

⟦struct-rewrite-expr ?P ?$init_c$′ ?$init_c$; abs-guard lift-global-heap ?P′ ?P; abs-expr lift-global-heap ?Q ?$init_a$?$init_c$′; THIN (⋀p. L2Tcorres lift-global-heap (?f_a p) (?f_c p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P′ s ∧ ?Q s)) (λ-. heap-outer-C.guard-with-fresh-stack-ptr ?n ?$init_a$ (L2-VARS ?f_a ?nm))) (globals.with-fresh-stack-ptr ?n ?$init_c$ (L2-VARS ?f_c ?nm))

⟦⋀s p. ?f_c p ⋅ s ?{ λr. globals.typing.unchanged-typing-on $\mathcal{S}$ s }; struct-rewrite-expr ?P ?$init_c$′ ?$init_c$; abs-guard lift-global-heap ?P′ ?P; abs-expr lift-global-heap ?Q ?$init_a$?$init_c$′; THIN (⋀p. L2Tcorres lift-global-heap (?f_a p) (?f_c p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P′ s ∧ ?Q s)) (λ-. heap-outer-C.assume-with-fresh-stack-ptr ?n ?$init_a$ (L2-VARS ?f_a ?nm))) (globals.with-fresh-stack-ptr ?n ?$init_c$ (L2-VARS ?f_c ?nm))

⟦struct-rewrite-expr ?P ?$init_c$′ ?$init_c$; abs-guard lift-global-heap ?P′ ?P; abs-expr lift-global-heap ?Q ?$init_a$?$init_c$′; THIN (⋀p. L2Tcorres lift-global-heap (?f_a p) (?f_c p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P′ s ∧ ?Q s)) (λ-. heap-other-C.guard-with-fresh-stack-ptr ?n ?$init_a$ (L2-VARS ?f_a ?nm))) (globals.with-fresh-stack-ptr ?n ?$init_c$ (L2-VARS ?f_c ?nm))

⟦⋀s p. ?f_c p ⋅ s ?{ λr. globals.typing.unchanged-typing-on $\mathcal{S}$ s }; struct-rewrite-expr ?P ?$init_c$′ ?$init_c$; abs-guard lift-global-heap ?P′ ?P; abs-expr lift-global-heap ?Q ?$init_a$?$init_c$′; THIN (⋀p. L2Tcorres lift-global-heap (?f_a p) (?f_c p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P′ s ∧ ?Q s)) (λ-. heap-other-C.assume-with-fresh-stack-ptr ?n ?$init_a$ (L2-VARS ?f_a ?nm))) (globals.with-fresh-stack-ptr ?n ?$init_c$ (L2-VARS ?f_c ?nm))

abs-expr lift-global-heap ?P ?a ?c ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λt. IS-VALID(unpacked-C[2]) t ?p ∧ ?P t)) (λ-. L2-modify (λs. unpacked-C.heap-array-map ?p (λ-. ?a s) s))) (globals.IO-modify-heap-paddingE ?p ?c)

⟦struct-rewrite-expr ?P ?$init_c$′ ?$init_c$; abs-guard lift-global-heap ?P′ ?P; abs-expr lift-global-heap ?Q ?$init_a$?$init_c$′; THIN (⋀p. L2Tcorres lift-global-heap (?f_a p) (?f_c p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P′ s ∧ ?Q s)) (λ-. heap-unpacked-C′array-2.guard-with-fresh-stack-ptr ?n ?$init_a$ (L2-VARS ?f_a ?nm))) (globals.with-fresh-stack-ptr ?n ?$init_c$ (L2-VARS ?f_c ?nm))

⟦⋀s p. ?f_c p ⋅ s ?{ λr. globals.typing.unchanged-typing-on $\mathcal{S}$ s }; struct-rewrite-expr ?P ?$init_c$′ ?$init_c$; abs-guard lift-global-heap ?P′ ?P; abs-expr lift-global-heap ?Q ?$init_a$?$init_c$′; THIN (⋀p. L2Tcorres lift-global-heap (?f_a p) (?f_c p))⟧ ⟹ L2Tcorres lift-global-heap (L2-seq (L2-guard (λs. ?P′ s ∧ ?Q s)) (λ-. heap-unpacked-C′array-2.assume-with-fresh-stack-ptr ?n ?$init_a$ (L2-VARS ?f_a ?nm))) (globals.with-fresh-stack-ptr ?n ?$init_c$ (L2-VARS ?f_c ?nm))

$[\![$*struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* ($\bigwedge$*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))$]\!] \Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* $\wedge$ *?Q s*)) (λ-. *heap-array-C.guard-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*))
$[\![\bigwedge$*s p. ?f$_c$ p* $\cdot$ *s ?*$\{\!|$ λ*r. globals.typing.unchanged-typing-on* $\mathcal{S}$ *s* $|\!\}$; *struct-rewrite-expr ?P ?init$_c$' ?init$_c$*; *abs-guard lift-global-heap ?P' ?P*; *abs-expr lift-global-heap ?Q ?init$_a$?init$_c$'*; *THIN* ($\bigwedge$*p. L2Tcorres lift-global-heap* (*?f$_a$ p*) (*?f$_c$ p*))$]\!]$ $\Longrightarrow$ *L2Tcorres lift-global-heap* (*L2-seq* (*L2-guard* (λ*s. ?P' s* $\wedge$ *?Q s*)) (λ-. *heap-array-C.assume-with-fresh-stack-ptr ?n ?init$_a$* (*L2-VARS ?f$_a$?nm*))) (*globals.with-fresh-stack-ptr ?n ?init$_c$* (*L2-VARS ?f$_c$?nm*)) and keeping in mind that they are used by recursively applying them as introduction rules to a concrete program C and a schematic variable for the abstract program A one can see the following strategy:

- First some normalisation of expressions involving pointers is performed on the concrete program C. In particular an update to a dereferenced pointer $\&(p{\rightarrow}f)$ by a value v is transformed to an update on p, where first the value of p is fetched from memory and then field f of this value is updated by v and the resulting compound value is put back to memory.

- Only after the normalisation on C, the actual translation to an abstract A on the split heap is performed.

  The normalisation step is guided by predicates *struct-rewrite-guard*, *struct-rewrite-expr* and *struct-rewrite-modifies*. After that normalisation the atomic building block for the simulation is provided by an instance of *typ-heap-simulation*, which tells how a lookup and update of a pointer of a given type in the byte heap is simulated in the split heap.

  The proofs of the necessary instances of *typ-heap-simulation* are provided once and forall in the initialisation phase of **autocorres** or **init−autocorres**. They are provided for atomic and shared heaps as well as derived heaps.

Proving *typ-heap-simulation*

As stated before this step was extended quite excessively and quite some infrastructure of both theorems as well as ML code were built to support it.

Typed vs. Untyped Dialects/ Records vs. Byte-Lists First some general remarks on the general concepts and setup. The UMM was designed

in the spirit to take advantage of HOL type and class inference to support reasoning about C-types and pointers. As a consequence some of the definitions and theorems are rather subtle and somehow live an the edge of what can be typed and expressed in the HOL type system. Sometimes the essence or the limitations of a theorem only become 'visible' when also looking at the types of expressions, especially the pointer types. Unfortunately writing about these concepts is rather ambiguous with respect to the notion type. 'Type' might refer to the original C-structure in the C program, this type is related to an abstract HOL-type, which is a record. This record is an instance of various type-classes: *c-type*, *mem-type*, *xmem-type*. For types of *c-type* we define overloaded *typ-info-t*, which associates a HOL-term describing the type (referred to as type-description or type-information). Moreover, pointers represented in HOL carry the HOL-type they point to as phantom type.

An illustration for the subtle typing is the two terms $\&(p{\rightarrow}f) \neq \&(q{\rightarrow}g)$ vs. $p \neq q \Longrightarrow \&(p{\rightarrow}f) \neq \&(q{\rightarrow}g)$. Whereas in first term the pointers p and q have a different type, in the second term they both have the same type as the inequation in the precondition also makes the types equal. This might not always be the intended meaning. When the type of the pointer is irrelevant one might resort to plain addresses instead of pointers *ptr-val p* $=$ *ptr-val q* $\Longrightarrow \&(p{\rightarrow}f) = \&(q{\rightarrow}g)$. This general theme occurs in various places. There often is a typed variant of a concept and also an untyped variant, based on addresses or byte lists, like p vs. *ptr-val p*. Both are closely related. The typed variant somehow reflects the 'API' of the concept and is more abstract, whereas during a proof one actually resort to the untyped variant to gain flexibility. These two views on a concepts leads to a duplication of lemmas. Moreover, a lemma might not immediately be available in the form one expects, but can be 'easily' derived from some related lemmas.

The central UMM HOL-datatype to reflect C-types into HOL terms is $('a, 'b)$ *typ-desc* and is defined mutually recursive with $('a, 'b)$ *typ-struct*. It is basically a tree describing the nested structure of an aggregate type. Field names, alignment and size information is encoded, and lookup and update of sub-fields can be derived from this information.

The most prominent instances of this type are the typed variant $'a$ *xtyp-info* vs. the untyped variant *typ-uinfo*. The former can be transformed to the latter by *export-uinfo*. As with pointer types the type variable in $'a$ *xtyp-info* denotes the abstract HOL-C-type that is described by the type information. So to derive a field-lookup or field-update on HOL-types from the type information one needs to resort to the typed variant. Exporting the type information via *export-uinfo* maintains the shape of the tree, but removes all the HOL-C-type dependent components. The resulting *typ-uinfo* has two main use cases:

- It can be seen as a type-tag identifying a C-type. It can be used to relate two C-types on the HOL-term level, e.g. ask if they are equal or if one is contained in the other.
- It can be used to normalise a byte-list representing a value of that type. Normalisation means that all padding bytes in the byte-list are set to zero.

  For each C-type the C-Parser also generates the type information which can be accessed via the overloaded function: *typ-info-t TYPE('a)*.

  The main use cases for *typ-info-t TYPE('a)* are
- Provide lenses for fields (access / update) on the abstract value (record), via *access-ti*, *update-ti* and *field-lookup* The main use cases for *typ-uinfo-t TYPE('a)* are
- comparison of type descriptors: equality and order $s \leq_\tau t$
- normalisation of byte-lists *norm-tu*, in particular setting all padding bytes to zero.

thm *sub-typ-def*
thm *typ-tag-le-def*

So *typ-info-t* is more related to the abstract view on a value (as a HOL record), whereas *typ-uinfo-t* is more related to the byte-list encoding of the value. This duality somehow also reflects the dual nature of types in C. On the one hand C is a statically typed language and on the other hand it allows to break the abstraction and switch to a low-level byte oriented view.

context
  fixes *p*::*'a*::*c-type ptr*
  fixes *f*::*qualified-field-name*
  fixes *t*::*'a xtyp-info*
  fixes *n*:: *nat*
begin

A central function on both typed and untyped typ-information is the function *field-lookup* which retrieves the type-information for a field of a type. A common application of this function is to state some property on a dereferenced pointer: *PTR('b)* &(*p*→*f*). Note that *p* is an pointer to an *'a*: *p*. A typical precondition is to retrieve the type-information for field *f* from the type information of *'a*: *field-lookup* (*typ-info-t TYPE('a)*) *f 0* = *Some* (*t*, *n*)

Note that the retrieved type information *t* is still tagged with *'a*. The number *n* is the offset of the selected field. Typically we want to relate *t* to the type of the selected field *'b* (which only happens to be the same *'a* in case the field is the empty list). The relation can be established via

export-uinfo, namely *export-uinfo t* = *export-uinfo* (*typ-info-t TYPE*('*b*)). Note that directly equating *t* to *typ-info-t TYPE*('*b*) is not even a well-typed expression in HOL, as '*a* is not equal to '*b*.

The right hand side abbreviates to constant *typ-uinfo-t TYPE*('*b*).

end

A concrete example might give some further insight to the relation of *typ-info-t* and *typ-uinfo-t* and how the type-information is constructed. Let us consider the field *''inner-C''* of *outer-C*, which selects a field of *inner-C*. We can retrieve the information for the field:

lemma *field-lookup* (*typ-info-t TYPE*(*outer-C*)) [*''inner-C''*] *0* =
Some (*adjust-ti* (*typ-info-t TYPE*(*inner-C*)) *outer-C.inner-C* (*inner-C-update* ∘ (λ*x* -. *x*)), *0*)
⟨*proof*⟩

Note that the retrieved type information is constructed from the nested type information for *inner-C*, by adjusting it. Adjusting means that we say how we can lookup and update the sub-field of the record *outer-C*. This adjustment only affects the typed-view of the type information. Exporting the type information collapses to the adjusted inner type:

lemma *export-uinfo* (*adjust-ti* (*typ-info-t TYPE*(*inner-C*)) *outer-C.inner-C* (*inner-C-update* ∘ (λ*x* -. *x*)))
= *export-uinfo* (*typ-info-t TYPE*(*inner-C*))
⟨*proof*⟩

In the realm of 'lenses", *adjust-ti* can be viewed as a form of composition of lenses. A lense for an inner type is transformed to a lense on the outer type. The lense assotiated to type information is captured in *access-ti*, for the 'lense-lookup' aka. 'lense-get' part, and *update-ti* for the 'lense-update' aka 'lense-put'.

Function *field-lookup* is the essential building block to relate field names e.g. as part of dereferencing a pointer to their abstracts operations on the associated HOL-type. One can convert between the typed and untyped version:

thm *field-lookup-export-uinfo-Some-rev*
thm *field-lookup-export-uinfo-Some*

Here are some closely related, mostly derived concepts around *field-lookup*:

- *field-lookup*, *field-ti*, *field-offset*, *field-of*
- *td-set*, *sub-typ* (*s* ≤$_\tau$ *t*
- Family of field name functions: *all-field-names*, *TypHeap.field-names*, *field-names-u*, *field-names-no-padding*, *all-field-names-no-padding*

  Often lemmas might not be available for all variants, but via some simple indirections, via definitions or conversion lemmas. In an Isar-proof, **sledgehammer** can often help to find the connections.

thm *td-set-field-lookupD*
thm *td-set-field-lookup*
thm *all-field-names-union-field-names-export-uinfo-conv*
thm *set-field-names-all-field-names-conv*
thm *field-names-u-field-names-export-uinfo-conv(1)*
thm *set-field-names-no-padding-all-field-names-no-padding-conv*
thm *all-field-names-no-padding-typ-uinfo-t-conv*

Doing the Proof The fundamental goal is to derive interpretations *typ-heap-simulation* for every relevant type. It states that the virtual heap for a type as obtained from *lifted-globals* simulates the original UMM heap in *globals*. Both states are connected by the abstraction function *lift-global-heap* which is the abstract function *st* in *typ-heap-simulation*. To facilitate the construction of the proofs we introduce intermediate helper locale *typ-heap-simulation-open-types* and make use of the infrastructure of lenses and scenes that we mentioned before.

To optimize the construction (in particular to minimise the number of heaps we have to put into into *lifted-globals*) we distinguish three main cases:

- A structure is closed and has no addressable fields: ⟦*open-types ?𝒯*; *heap-typing-simulation ?𝒯 ?hrs ?hrs-upd ?heap-typing ?heap-typing-upd ?l*; *lense ?R ?u*; *map-of ?𝒯* (*typ-uinfo-t TYPE*(*?'a*)) = *None*; ⋀*p x s*. *open-types.ptr-valid ?𝒯* (*hrs-htd* (*?hrs s*)) *p* ⟹ *?l* (*?hrs-upd* (*open-types.write-dedicated-heap ?𝒯 p x*) *s*) = *?u* (*upd-fun p* (λ*old*. *merge-ti-list* (*map snd* (*open-types.addressable-fields ?𝒯 TYPE*(*?'a*))) *old x*)) (*?l s*); ⋀*x d h*. *?heap-typing-upd d* (*?u x h*) = *?u x* (*?heap-typing-upd d h*); ⋀*p s*. ∀ *a*∈*ptr-span p*. *root-ptr-valid* (*hrs-htd* (*?hrs s*)) (*PTR*(*stack-byte*) *a*) ⟹ *?R* (*?l s*) *p* = *ZERO*(*?'a*); ⋀*h p*. *?V h p* = *open-types.ptr-valid ?𝒯* (*?heap-typing h*) *p*; ⋀*p f h*. *?W p f h* = *?u* (λ*h'*. *h'*(*p* := *f* (*h' p*))) *h*⟧ ⟹ *typ-heap-simulation-open-types ?𝒯 ?l ?R ?W ?V ?hrs ?hrs-upd ?heap-typing ?heap-typing-upd* In this case there is only one relevant heap in *lifted-globals*. No overlay of a dedicated heap with some common heaps is necessary.

- A structure is completely open, meaning that all fields are addressable: ⟦*open-types ?𝒯*; *heap-typing-simulation ?𝒯 ?hrs ?hrs-upd ?heap-typing ?heap-typing-upd ?l*; *map-of ?𝒯* (*typ-uinfo-t TYPE*(*?'a*)) = *Some ?fs*; *length ?rs* = *length ?fs*; *length ?ws* = *length ?fs*; *list-all* (λ(*f*, *r*, *w*). *open-types.typ-heap-simulation-of-field ?𝒯 ?l ?hrs ?hrs-upd ?heap-typing-upd f r w*) (*zip ?fs* (*zip ?rs ?ws*)); *distinct-prop* (λ(*f1*, *w1*) (*f2*, *w2*). *disj-fn f1 f2* ⟶ *pointer-writer-disjnt-eq w1 w2*) (*zip ?fs ?ws*); ⋀*a b*. *fold* (λ*x*. *merge-ti* (*the* (*field-ti TYPE*(*?'a*) *x*)) *a*) *?fs b* = *a*; ⋀*h p*. *?V h p* = *open-types.ptr-valid ?𝒯* (*?heap-typing h*) *p*; ⋀*h p x*. *?R h p* = *fold* (λ*r*. *r h p*) *?rs x*; ⋀*p f h*. *?W p f h* = *fold* (λ*w*. *w p* (*f* (*?R h p*))) *?ws h*⟧ ⟹ *typ-heap-simulation-open-types ?𝒯 ?l ?R ?W ?V*

?hrs ?hrs-upd ?heap-typing ?heap-typing-upd $\wedge$ ($\forall$ *w*. ($\forall$ *x*. *list-all* (λ*w'*. *pointer-writer-disjnt* (λ*p*. *w' p x*) *w*) *?ws*) $\longrightarrow$ ($\forall$ *f*. *pointer-writer-disjnt* (λ*p*. *?W p f*) *w*)) $\wedge$ ($\forall$ *w p*. ($\forall$ *x*. *list-all* (λ*w'*. *w' p x* $\circ$ *w* = *w* $\circ$ *w' p x*) *?ws*) $\longrightarrow$ ($\forall$ *f*. *?W p f* $\circ$ *w* = *w* $\circ$ *?W p f*)). In this case we do not need a dedicated heap. The structure is described by a overlay of common heaps.

- A structure is partially open, some fields are addressable and some not: ⟦*open-types ?𝒯*; *heap-typing-simulation ?𝒯 ?hrs ?hrs-upd ?heap-typing ?heap-typing-upd ?l*; *map-of ?𝒯* (*typ-uinfo-t TYPE*(*?'a*)) = *Some ?fs*; *lense ?g ?u*; *length ?rs* = *length ?fs*; *length ?ws* = *length ?fs*; *list-all* (λ(*f*, *r*, *w*). *open-types.typ-heap-simulation-of-field ?𝒯 ?l ?hrs ?hrs-upd ?heap-typing-upd f r w*) (*zip ?fs* (*zip ?rs ?ws*)); *distinct-prop* (λ(*f1*, *w1*) (*f2*, *w2*). *disj-fn f1 f2* $\longrightarrow$ *pointer-writer-disjnt-eq w1 w2*) (*zip ?fs ?ws*); *list-all* (λ*w*. $\forall$ *p a f*. *w p a* $\circ$ *?u f* = *?u f* $\circ$ *w p a*) *?ws*; $\bigwedge$*p x s*. *open-types.ptr-valid ?𝒯* (*hrs-htd* (*?hrs s*)) *p* $\Longrightarrow$ *?l* (*?hrs-upd* (*open-types.write-dedicated-heap ?𝒯 p x*) *s*) = *?u* (*upd-fun p* (λ*old*. *merge-ti-list* (*map snd* (*open-types.addressable-fields ?𝒯 TYPE*(*?'a*))) *old x*)) (*?l s*); $\bigwedge$*p s*. $\forall$ *a*$\in$*ptr-span p*. *root-ptr-valid* (*hrs-htd* (*?hrs s*)) (*PTR*(*stack-byte*) *a*) $\Longrightarrow$ *?g* (*?l s*) *p* = *ZERO*(*?'a*); $\bigwedge$*x d h*. *?heap-typing-upd d* (*?u x h*) = *?u x* (*?heap-typing-upd d h*); $\bigwedge$*h p*. *?V h p* = *open-types.ptr-valid ?𝒯* (*?heap-typing h*) *p*; $\bigwedge$*h p*. *?R h p* = *fold* (λ*r*. *r h p*) *?rs* (*?g h p*); $\bigwedge$*p f h*. *?W p f h* = *fold* (λ*w*. *w p* (*f* (*?R h p*))) *?ws* (*?u* (*upd-fun p* (λ*old*. *merge-ti-list* (*map snd* (*open-types.addressable-fields ?𝒯 TYPE*(*?'a*))) *old* (*f* (*?R h p*)))) *h*)⟧ $\Longrightarrow$ *typ-heap-simulation-open-types ?𝒯 ?l ?R ?W ?V ?hrs ?hrs-upd ?heap-typing ?heap-typing-upd* $\wedge$ *pointer-lense ?g* (λ*p f*. *?u* (*upd-fun p f*)) $\wedge$ ($\forall$ *w*. ($\forall$ *x*. *list-all* (λ*w'*. *pointer-writer-disjnt* (λ*p*. *w' p x*) *w*) *?ws*) $\longrightarrow$ ($\forall$ *x*. *pointer-writer-disjnt* (λ*p*. *?u* (*upd-fun p* (λ-. *x*))) *w*) $\longrightarrow$ ($\forall$ *f*. *pointer-writer-disjnt* (λ*p*. *?W p f*) *w*)) $\wedge$ ($\forall$ *w p*. ($\forall$ *x*. *list-all* (λ*w'*. *w' p x* $\circ$ *w* = *w* $\circ$ *w' p x*) *?ws*) $\longrightarrow$ ($\forall$ *x*. *?u* (*upd-fun p* (λ-. *x*)) $\circ$ *w* = *w* $\circ$ *?u* (*upd-fun p* (λ-. *x*))) $\longrightarrow$ ($\forall$ *f*. *?W p f* $\circ$ *w* = *w* $\circ$ *?W p f*)). In that case we need an overlay of a dedicated heap and some common heaps.

  On an intuitive abstract level the lemmas and proof argue about composing field-level lookup and updates within some heap(s) to lookup and updates of the complete compound structure. We want to argue that lookup and update in the UMM heap is simulated by the overlayed updates of a dedicated heap and some common heaps in the split heap. To argue about different fields of a structure we build on the idea of scenes and also extend it to reads. Note that the general problem is that a structure is represented as a HOL record and each field has a distinct type. Because of the limitations of the HOL type system we cannot directly combine e.g. functions like readers for different fields like *'a* $\Rightarrow$ *'b1* and *'a* $\Rightarrow$ *'b2* in a HOL list. Here the scene idea to

express everything via a merge or update on $'a$ is a helpful 'trick'. E.g. 'reading' the value of a field can in a sense be as well expressed as a function $'a \Rightarrow 'a \Rightarrow 'a$ that reads the field of the first argument and puts it into any structure you give it as second argument:

thm *lift-global-heap-def*
thm *open-types.typ-heap-simulationI-all-addressable*
thm *open-types.typ-heap-simulationI-part-addressable*
thm *open-types.typ-heap-simulationI-no-addressable*
thm *pointer-lense-def*
thm *partial-pointer-lense-def*
thm *typ-heap-simulation-of-field-def*

Other important building blocks are:

- *pointer-lense* which describes a virtual heap.
- *partial-pointer-lense* which describes the effect of field lookup / updates on a virtual heap by combining a scene identifying the field and the pointer lense for the virtual heap.
- *typ-heap-simulation-of-field ?st ?t-hrs ?t-hrs-update ?heap-typing-upd ?f' ?r' ?w'* = $((\forall d\ p\ f.$ *?heap-typing-upd* $d \circ$ *?w'* $p\ f =$ *?w'* $p\ f \circ$ *?heap-typing-upd* $d) \wedge (\forall t\ u\ n.$ *field-ti* $TYPE(?'a)$ *?f'* $=$ *Some* $t \longrightarrow$ *field-lookup* (*typ-uinfo-t* $TYPE(?'a)$) *?f'* $0 =$ *Some* $(u, n) \longrightarrow$ *partial-pointer-lense* (*merge-ti* t) *?r' ?w'* $\wedge (\forall p\ s.\ (\forall a \in$ *ptr-span* $p.$ *root-ptr-valid* (*hrs-htd* (*?t-hrs* s)) ($PTR($*stack-byte*$)\ a$)) $\longrightarrow$ *?r'* (*?st* s) p $ZERO(?'a) = ZERO(?'a)) \wedge (\forall p\ x\ h.$ *ptr-valid-u* u (*hrs-htd* (*?t-hrs* h)) $\&(p \rightarrow ?f') \longrightarrow$ *?st* (*?t-hrs-update* (*hrs-mem-update* (*heap-upd-list* (*size-td* u) $\&(p \rightarrow ?f')$ (*access-ti* $t\ x$))) h) $=$ *?w'* $p\ x$ (*?st* h)))) which is used to break down *typ-heap-simulation* to the level of fields in the structure.

The automation is implemented in `HeapLiftBase.gen_new_heap`.

The following paragraphs describe some abstract arguments and lemma collections. In a previous version the proof *typ-heap-simulation* was more directly based on those arguments implemented in ML. Meanwhile we were able to replace most parts of the ML code by lemmas described before based on scenes and pointer lenses. Nevertheless the arguments are still valid and the lemma collections might be useful in other places.

Fundamental heaps We start with the fundamental heaps. The simulation property for heap-updates is captured in *write-simulation*. It is a commutation property. Provided we have a valid pointer in the split heap (obtained from lifting from the byte-level heap), we can either first perform the heap update in the byte-level heap and then lift the state via

lift-global-heap, or first lift the byte-level heap into the split heap and apply the corresponding update there. Both ways yield the same final state. So we have to prove equality of two states of *lifted-globals*. The first question is which parts of the state did change? Intuitively only the affected split heap did change, all other heaps stayed the same. But all split heaps are derived from the same byte level heap. So the proof decomposes into two main arguments, one is for the affected heap where we have to show that both updates, the one on the byte-level heap and and the one on the affected split heap, are the same. The other argument is that all other split heaps remain unchanged. Actually the first case is the more straight forward one and could be handled with theorems like *PTR-VALID*(*?'a*) (*hrs-htd ?h*) *?p* $\Longrightarrow$ *plift* (*hrs-mem-update* (*heap-update ?p ?v*) *?h*) = (*plift ?h*)(*?p* $\mapsto$ *?v*).

thm *plift-heap-update*

The second case, to prove what is not changed is more involved. As a split heap might not only contain root pointers but also valid pointers that are nested in other types we have to distinguish several cases. We developed a general theory in *open-types* and the essential theorems for the commutation proof are collected in ⟦*PTR-VALID*(*?'a*) *?d ?p*; *PTR-VALID*(*?'a*) *?d ?q*⟧ $\Longrightarrow$ *h-val* (*heap-update ?p ?v ?h*) *?q* = ((*h-val ?h*)(*?p* := *?v*)) *?q*

⟦*PTR-VALID*(*?'a*) *?d ?p*; *PTR-VALID*(*?'a*) *?d ?q*; *length ?bs* = *size-of TYPE*(*?'a*)⟧ $\Longrightarrow$ *h-val* (*heap-update-padding ?p ?v ?bs ?h*) *?q* = ((*h-val ?h*)(*?p* := *?v*)) *?q*

PTR-VALID(*?'a*) (*hrs-htd ?h*) *?p* $\Longrightarrow$ *plift ?h ?p* = *Some* (*h-val* (*hrs-mem ?h*) *?p*). When we update the heap at pointer *p* and lookup the value of pointer *q* we can distinguish various cases by analysing the type relation (e.g. if one type is nested in the other). Note that by introducing guards into the code and the design of *lift-global-heap* we only have to care about the case were both pointers are valid according to *PTR-VALID*('*a*). To be more specific. That the pointer that is updated is *PTR-VALID*('*a*) is ensured by the corresponding guard on the update. That the pointer we read from is actually *PTR-VALID*('*a*) is more subtle. Here we rely on the construction of *lift-global-heap* were a split heap is constructed by λp. *the-default ZERO*('*a*) (*plift* (*t-hrs-'* *g*) *p*). In case the pointer is invalid we always obtain *ZERO*('*a*) and hence only the valid pointers may be affected by an heap update. This reduction to valid pointers and *h-val* is encapsulated in theorems *root-ptr-valid* (*hrs-htd ?h*) *?p* $\Longrightarrow$ *the-default ZERO*(*?'a*) (*simple-lift* (*hrs-mem-update* (*heap-update ?p ?v*) *?h*) *?q*) = ((λp. *the-default ZERO*(*?'a*) (*simple-lift ?h p*))(*?p* := *?v*)) *?q*

⟦*root-ptr-valid* (*hrs-htd ?h*) *?p*; *length ?bs* = *size-of TYPE*(*?'a*)⟧ $\Longrightarrow$ *the-default ZERO*(*?'a*) (*simple-lift* (*hrs-mem-update* (*heap-update-padding ?p ?v ?bs*) *?h*) *?q*) = ((λp. *the-default ZERO*(*?'a*) (*simple-lift ?h p*))(*?p* := *?v*)) *?q*

root-ptr-valid (*hrs-htd ?h*) *?p* $\wedge$ *?f ?x* = *?v* $\Longrightarrow$ *?f* (*the-default ZERO*(*?'a*)

(*simple-lift* (*hrs-mem-update* (*heap-update ?p ?x*) *?h*) *?q*)) = ((λ*p*. *?f* (*the-default ZERO*(*?'a*) (*simple-lift ?h p*)))(*?p* := *?v*)) *?q*

⟦*root-ptr-valid* (*hrs-htd ?h*) *?p* ∧ *?f ?x* = *?v*; *length ?bs* = *size-of TYPE*(*?'a*)⟧ ⟹ *?f* (*the-default ZERO*(*?'a*) (*simple-lift* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*) *?h*) *?q*)) = ((λ*p*. *?f* (*the-default ZERO*(*?'a*) (*simple-lift ?h p*)))(*?p* := *?v*)) *?q*

(⟦*root-ptr-valid* (*hrs-htd ?h*) *?q*; *hrs-htd ?h* ⊨$_t$ *?q*⟧ ⟹ *h-val* (*heap-update ?p ?v* (*hrs-mem ?h*)) *?q* = *h-val* (*hrs-mem ?h*) *?q*) ⟹ *the-default ZERO*(*?'a*) (*simple-lift* (*hrs-mem-update* (*heap-update ?p ?v*) *?h*) *?q*) = *the-default ZERO*(*?'a*) (*simple-lift ?h ?q*)

⟦*length ?bs* = *size-of TYPE*(*?'b*); ⟦*root-ptr-valid* (*hrs-htd ?h*) *?q*; *hrs-htd ?h* ⊨$_t$ *?q*⟧ ⟹ *h-val* (*heap-update-padding ?p ?v ?bs* (*hrs-mem ?h*)) *?q* = *h-val* (*hrs-mem ?h*) *?q*⟧ ⟹ *the-default ZERO*(*?'a*) (*simple-lift* (*hrs-mem-update* (*heap-update-padding ?p ?v ?bs*) *?h*) *?q*) = *the-default ZERO*(*?'a*) (*simple-lift ?h ?q*)

(⟦*root-ptr-valid* (*hrs-htd ?h*) *?q*; *hrs-htd ?h* ⊨$_t$ *?q*⟧ ⟹ *?f* (*h-val* (*heap-update ?p ?v* (*hrs-mem ?h*)) *?q*) = *?f* (*h-val* (*hrs-mem ?h*) *?q*)) ⟹ *?f* (*the-default ZERO*(*?'a*) (*simple-lift* (*hrs-mem-update* (*heap-update ?p ?v*) *?h*) *?q*)) = *?f* (*the-default ZERO*(*?'a*) (*simple-lift ?h ?q*))

⟦*length ?bs* = *size-of TYPE*(*?'b*); ⟦*root-ptr-valid* (*hrs-htd ?h*) *?q*; *hrs-htd ?h* ⊨$_t$ *?q*⟧ ⟹ *?f* (*h-val* (*heap-update-padding ?p ?v ?bs* (*hrs-mem ?h*)) *?q*) = *?f* (*h-val* (*hrs-mem ?h*) *?q*)⟧ ⟹ *?f* (*the-default ZERO*(*?'a*) (*simple-lift* (*hrs-mem-update* (*heap-update-padding ?p ?v ?bs*) *?h*) *?q*)) = *?f* (*the-default ZERO*(*?'a*) (*simple-lift ?h ?q*))

PTR-VALID(*?'a*) (*hrs-htd ?h*) *?p* ⟹ *the-default ZERO*(*?'a*) (*plift* (*hrs-mem-update* (*heap-update ?p ?v*) *?h*) *?q*) = ((λ*p*. *the-default ZERO*(*?'a*) (*plift ?h p*))(*?p* := *?v*)) *?q*

⟦*PTR-VALID*(*?'a*) (*hrs-htd ?h*) *?p*; *length ?bs* = *size-of TYPE*(*?'a*)⟧ ⟹ *the-default ZERO*(*?'a*) (*plift* (*hrs-mem-update* (*heap-update-padding ?p ?v ?bs*) *?h*) *?q*) = ((λ*p*. *the-default ZERO*(*?'a*) (*plift ?h p*))(*?p* := *?v*)) *?q*

PTR-VALID(*?'a*) (*hrs-htd ?h*) *?p* ∧ *?f ?x* = *?v* ⟹ *?f* (*the-default ZERO*(*?'a*) (*plift* (*hrs-mem-update* (*heap-update ?p ?x*) *?h*) *?q*)) = ((λ*p*. *?f* (*the-default ZERO*(*?'a*) (*plift ?h p*)))(*?p* := *?v*)) *?q*

⟦*PTR-VALID*(*?'a*) (*hrs-htd ?h*) *?p* ∧ *?f ?x* = *?v*; *length ?bs* = *size-of TYPE*(*?'a*)⟧ ⟹ *?f* (*the-default ZERO*(*?'a*) (*plift* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*) *?h*) *?q*)) = ((λ*p*. *?f* (*the-default ZERO*(*?'a*) (*plift ?h p*)))(*?p* := *?v*)) *?q*

(⟦*PTR-VALID*(*?'a*) (*hrs-htd ?h*) *?q*; *hrs-htd ?h* ⊨$_t$ *?q*⟧ ⟹ *h-val* (*heap-update ?p ?v* (*hrs-mem ?h*)) *?q* = *h-val* (*hrs-mem ?h*) *?q*) ⟹ *the-default ZERO*(*?'a*) (*plift* (*hrs-mem-update* (*heap-update ?p ?v*) *?h*) *?q*) = *the-default ZERO*(*?'a*) (*plift ?h ?q*)

⟦*length ?bs* = *size-of TYPE*(*?'b*); ⟦*PTR-VALID*(*?'a*) (*hrs-htd ?h*) *?q*; *hrs-htd ?h* ⊨$_t$ *?q*⟧ ⟹ *h-val* (*heap-update-padding ?p ?v ?bs* (*hrs-mem ?h*)) *?q* = *h-val* (*hrs-mem ?h*) *?q*⟧ ⟹ *the-default ZERO*(*?'a*) (*plift* (*hrs-mem-update*

(*heap-update-padding ?p ?v ?bs*) *?h*) *?q*) = *the-default ZERO*(*?'a*) (*plift ?h ?q*)

(⟦*PTR-VALID*(*?'a*) (*hrs-htd ?h*) *?q*; *hrs-htd ?h* ⊨$_t$ *?q*⟧ ⟹ *?f* (*h-val* (*heap-update ?p ?v* (*hrs-mem ?h*)) *?q*) = *?f* (*h-val* (*hrs-mem ?h*) *?q*)) ⟹ *?f* (*the-default ZERO*(*?'a*) (*plift* (*hrs-mem-update* (*heap-update ?p ?v*) *?h*) *?q*)) = *?f* (*the-default ZERO*(*?'a*) (*plift ?h ?q*))

⟦*length ?bs* = *size-of TYPE*(*?'b*); ⟦*PTR-VALID*(*?'a*) (*hrs-htd ?h*) *?q*; *hrs-htd ?h* ⊨$_t$ *?q*⟧ ⟹ *?f* (*h-val* (*heap-update-padding ?p ?v ?bs* (*hrs-mem ?h*)) *?q*) = *?f* (*h-val* (*hrs-mem ?h*) *?q*)⟧ ⟹ *?f* (*the-default ZERO*(*?'a*) (*plift* (*hrs-mem-update* (*heap-update-padding ?p ?v ?bs*) *?h*) *?q*)) = *?f* (*the-default ZERO*(*?'a*) (*plift ?h ?q*)).

thm *lift-global-heap-def*
thm *the-plift-hval-eqI*
thm *plift-eqI*
thm *plift-simps*

Treatment of Array Types At the core there is no special type-information for array types. It is treated analogously to structures. Each index gets an individual field name which is the unary encoding of the index. The good thing is that one does not need any new fundamental lemmas to deal with array types. But it is not a good idea to unfold the type-information for arrays and to work on that expanded version. On the one hand things like simplification might become slow, on the other hand we can make use of the regular structure of array types and once and forall derive general lemmas.

The central theorems to work with field-lookup in arrays are *?n* < *CARD*(*?'b*) ⟹ *field-lookup* (*typ-info-t TYPE*(*?'a*[*?'b*])) [*replicate ?n CHR ''1''*] *?i* = *Some* (*adjust-ti* (*typ-info-t TYPE*(*?'a*)) (λ*x*. *x*.[*?n*]) (λ*x f*. *Arrays.update f ?n x*), *?i* + *?n* ∗ *size-of TYPE*(*?'a*)) and *?i* < *CARD*(*?'b*) ⟹ *array-ptr-index ?p False ?i* = *PTR*(*?'a*) &(*?p*→[*replicate ?i CHR ''1''*]). The latter one allows to convert between an array-index-arithmetic based view of array pointers *array-ptr-index p False i* and the field-name view *PTR*('*a*) &(*p*→[*replicate i CHR ''1''*]).

thm *field-lookup-array*
thm *array-ptr-index-field-lvalue-conv*

These theorems are used to normalise array accesses towards symbolic field name accesses of the format term>*Ptr* &(*p*→[*replicate i CHR ''1''*]). Note that we don't unfold the definition of *replicate* here, and index *i* stays abstract, constrained by precondition that limits its range.

thm *unpacked-C.ptr-valid.unfold*
thm *unpacked-C'array-2.ptr-valid.unfold*

The instances of *typ-heap-simulation* can be established from *typ-heap-simulation* of the element types. We introduce the locales *array-typ-heap-simulation* and *two-dimensional-array-typ-heap-simulation* to automatically provide the

sublocale relations when the element type falls into *array-outer-max-size* or *array-inner-max-size* respectively.

Derived Heaps The commutation proofs for derived heaps follow another path. For derived heaps we can assume that we have all the instances of *typ-heap-simulation* of the component types already available. Unaddressable fields are never shared. Pointers only appear within a dedicated enclosing parent structure. Validity of the pointer coincides with validity of the parent pointer. Pointers that are mapped by the field heap coincide with the parent pointers, there is no need to calculate the offset of the field. So deriving the commutation proof from the available theorems boils down to their composition. The central lemma connects an update of a derived heap to an fold over the updates of the toplevel fields: *c-guard ?p* $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*32 word*) &(*?p*→[*''y4-C''*])) (*y4-C ?x*) ∘ (*heap-update* (*PTR*(*32 word*) &(*?p*→[*''y3-C''*])) (*y3-C ?x*) ∘ (*heap-update* (*PTR*(*32 word*) &(*?p*→[*''y2-C''*])) (*y2-C ?x*) ∘ (*heap-update* (*PTR*(*32 word*) &(*?p*→[*''y1-C''*])) (*y1-C ?x*) ∘ *heap-update* (*PTR*(*32 signed word*) &(*?p*→[*''x-C''*])) (*x-C ?x*))))

c-guard ?p $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*32 word*) &(*?p*→[*''c2-C''*])) (*c2-C ?x*) ∘ *heap-update* (*PTR*(*32 word*) &(*?p*→[*''c1-C''*])) (*c1-C ?x*)

c-guard ?p $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*closed-C*) &(*?p*→[*''fld4-C''*])) (*fld4-C ?x*) ∘ (*heap-update* (*PTR*(*32 word*) &(*?p*→[*''fld3-C''*])) (*fld3-C ?x*) ∘ (*heap-update* (*PTR*(*32 word*) &(*?p*→[*''fld2-C''*])) (*fld2-C ?x*) ∘ *heap-update* (*PTR*(*32 word*) &(*?p*→[*''fld1-C''*])) (*fld1-C ?x*)))

c-guard ?p $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*32 word*) &(*?p*→[*''lng-C''*])) (*lng-C ?x*) ∘ *heap-update* (*PTR*(*8 word*) &(*?p*→[*''chr-C''*])) (*chr-C ?x*)

c-guard ?p $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*32 word*) &(*?p*→[*''fld-C''*])) (*outer-C.fld-C ?x*) ∘ *heap-update* (*PTR*(*inner-C*) &(*?p*→[*''inner-C''*])) (*outer-C.inner-C ?x*)

c-guard ?p $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*32 word*) &(*?p*→[*''count-C''*])) (*count-C ?x*) ∘ *heap-update* (*PTR*(*unpacked-C*[*2*]) &(*?p*→[*''elements-C''*])) (*elements-C ?x*)

c-guard ?p $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*unpacked-C*) &(*?p*→[*''fy-C''*])) (*fy-C ?x*) ∘ (*heap-update* (*PTR*(*closed-C*) &(*?p*→[*''fz-C''*])) (*fz-C ?x*) ∘ *heap-update* (*PTR*(*32 word*) &(*?p*→[*''fx-C''*])) (*fx-C ?x*))

c-guard ?p $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*unpacked-C*[*3*][*2*]) &(*?p*→[*''matrix-C''*])) (*matrix-C ?x*)

c-guard ?p $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*32 word*) &(*?p*→[*''fld-C''*])) (*outer-array-C.fld-C ?x*) ∘ *heap-update* (*PTR*(*inner-C*[*5*]) &(*?p*→[*''inner-array-C''*])) (*inner-array-C ?x*)

c-guard ?p $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*data-C*[*10*]) &(*?p*→[*''array-C''*])) (*data-array-C.array-C ?x*)

c-guard ?p $\Longrightarrow$ *heap-update ?p ?x = heap-update* (*PTR*(*data-C*) &(*?p*→[*''d2-C''*]))

(*d2-C ?x*) ∘ *heap-update* (*PTR*(*data-C*) &(*?p*→[''*d1-C*'']))
(*d1-C ?x*)

c-guard ?p ⟹ *heap-update ?p ?x* = *heap-update* (*PTR*(*data-struct1-C*) &(*?p*→[''*d-C*'']))
(*d-C ?x*)

c-guard ?p ⟹ *heap-update ?p ?x* = *heap-update* (*PTR*(*32 word*[*12*])
&(*?p*→[''*arr2-C*'']))
(*arr2-C ?x*) ∘ *heap-update* (*PTR*(*32 word*[*12*]) &(*?p*→[''*arr1-C*'']))
(*arr1-C ?x*). Moreover, we establish that *heap-update-padding* is equivalent to *heap-update* under the state lifting function. Padding bytes become irrelevant in the split heap. For each toplevel field we have the commutation proof connecting the monolithic byte-level heap update to the split-heap update collected in ⟦*IS-VALID*(*32 word*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*32 word*)⟧ ⟹ *lift-global-heap* (*t-hrs-'-update*
(*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-w32-update*
(λ*h*. *h*(*?p* := *?x*)) (*lift-global-heap ?s*)

IS-VALID(*32 word*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-'-update*
(*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-w32-update* (λ*h*. *h*(*?p* :=
?x)) (*lift-global-heap ?s*)

⟦*IS-VALID*(*32 word*) (*lift-global-heap ?s*) (*PTR-COERCE*(*32 signed word*
→ *32 word*) *?p*); *length ?bs* = *size-of TYPE*(*32 signed word*)⟧ ⟹ *lift-global-heap*
(*t-hrs-'-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) =
heap-w32-update (λ*h*. *h*(*PTR-COERCE*(*32 signed word* → *32 word*) *?p* :=
UCAST(*32 signed* → *32*) *?x*)) (*lift-global-heap ?s*)

IS-VALID(*32 word*) (*lift-global-heap ?s*) (*PTR-COERCE*(*32 signed word*
→ *32 word*) *?p*) ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update*
?p ?x)) *?s*) = *heap-w32-update* (λ*h*. *h*(*PTR-COERCE*(*32 signed word* →
32 word) *?p* := *UCAST*(*32 signed* → *32*) *?x*)) (*lift-global-heap ?s*)

⟦*IS-VALID*(*8 word*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*8*
word)⟧ ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update-padding*
?p ?x ?bs)) *?s*) = *heap-w8-update* (λ*h*. *h*(*?p* := *?x*)) (*lift-global-heap ?s*)

IS-VALID(*8 word*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-'-update*
(*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-w8-update* (λ*h*. *h*(*?p* :=
?x)) (*lift-global-heap ?s*)

⟦*IS-VALID*(*8 word*) (*lift-global-heap ?s*) (*PTR-COERCE*(*8 signed word*
→ *8 word*) *?p*); *length ?bs* = *size-of TYPE*(*8 signed word*)⟧ ⟹ *lift-global-heap*
(*t-hrs-'-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) =
heap-w8-update (λ*h*. *h*(*PTR-COERCE*(*8 signed word* → *8 word*) *?p* :=
UCAST(*8 signed* → *8*) *?x*)) (*lift-global-heap ?s*)

IS-VALID(*8 word*) (*lift-global-heap ?s*) (*PTR-COERCE*(*8 signed word*
→ *8 word*) *?p*) ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update*
?p ?x)) *?s*) = *heap-w8-update* (λ*h*. *h*(*PTR-COERCE*(*8 signed word* → *8*
word) *?p* := *UCAST*(*8 signed* → *8*) *?x*)) (*lift-global-heap ?s*)

⟦*IS-VALID*(*32 word ptr*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of*
TYPE(*32 word ptr*)⟧ ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update*
(*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-w32'ptr-update* (λ*h*. *h*(*?p* :=
?x)) (*lift-global-heap ?s*)

IS-VALID(*32 word ptr*) (*lift-global-heap ?s*) *?p* $\Longrightarrow$ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-w32'ptr-update* ($\lambda h.$ *h*(*?p* := *?x*)) (*lift-global-heap ?s*)

⟦*IS-VALID*(*unit ptr*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*unit ptr*)⟧ $\Longrightarrow$ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-unit'ptr-update* ($\lambda h.$ *h*(*?p* := *?x*)) (*lift-global-heap ?s*)

IS-VALID(*unit ptr*) (*lift-global-heap ?s*) *?p* $\Longrightarrow$ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-unit'ptr-update* ($\lambda h.$ *h*(*?p* := *?x*)) (*lift-global-heap ?s*)

⟦*IS-VALID*(*data-C*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*data-C*)⟧ $\Longrightarrow$ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-data-C-update* ($\lambda h.$ *h*(*?p* := *?x*)) (*lift-global-heap ?s*)

IS-VALID(*data-C*) (*lift-global-heap ?s*) *?p* $\Longrightarrow$ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-data-C-update* ($\lambda h.$ *h*(*?p* := *?x*)) (*lift-global-heap ?s*)

⟦*IS-VALID*(*closed-C*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*closed-C*)⟧ $\Longrightarrow$ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-closed-C-update* ($\lambda h.$ *h*(*?p* := *?x*)) (*lift-global-heap ?s*)

IS-VALID(*closed-C*) (*lift-global-heap ?s*) *?p* $\Longrightarrow$ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-closed-C-update* ($\lambda h.$ *h*(*?p* := *?x*)) (*lift-global-heap ?s*)

⟦*IS-VALID*(*unpacked-C*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*unpacked-C*)⟧ $\Longrightarrow$ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-unpacked-C-update* ($\lambda h.$ *h*(*?p* := *?x*)) (*lift-global-heap ?s*)

IS-VALID(*unpacked-C*) (*lift-global-heap ?s*) *?p* $\Longrightarrow$ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-unpacked-C-update* ($\lambda h.$ *h*(*?p* := *?x*)) (*lift-global-heap ?s*)

⟦*IS-VALID*(*arr-struct-C*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*arr-struct-C*)⟧ $\Longrightarrow$ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-arr-struct-C-update* ($\lambda h.$ *h*(*?p* := *?x*)) (*lift-global-heap ?s*)

IS-VALID(*arr-struct-C*) (*lift-global-heap ?s*) *?p* $\Longrightarrow$ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-arr-struct-C-update* ($\lambda h.$ *h*(*?p* := *?x*)) (*lift-global-heap ?s*)

⟦*valid-array-base.valid-array* ($\lambda h.$ *IS-VALID*(*unpacked-C*) *h*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*unpacked-C*[*3*])⟧ $\Longrightarrow$ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *unpacked-C.heap-array-map ?p* (λ*-. ?x*) (*lift-global-heap ?s*)

valid-array-base.valid-array ($\lambda h.$ *IS-VALID*(*unpacked-C*) *h*) (*lift-global-heap ?s*) *?p* $\Longrightarrow$ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *unpacked-C.heap-array-map ?p* (λ*-. ?x*) (*lift-global-heap ?s*)

⟦*valid-array-base.valid-array* (*valid-array-base.valid-array* (λ*h*. *IS-VALID*(*unpacked-C*) *h*)) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*unpacked-C*[*3*][*2*])⟧ ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *unpacked-C.outer.heap-array-map ?p* (λ*-. ?x*) (*lift-global-heap ?s*)

valid-array-base.valid-array (*valid-array-base.valid-array* (λ*h*. *IS-VALID*(*unpacked-C*) *h*)) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *unpacked-C.outer.heap-array-map ?p* (λ*-. ?x*) (*lift-global-heap ?s*)

⟦*IS-VALID*(*two-dimensional-C*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*two-dimensional-C*)⟧ ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-two-dimensional-C-map ?p* (λ*-. ?x*) (*lift-global-heap ?s*)

IS-VALID(*two-dimensional-C*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-two-dimensional-C-map ?p* (λ*-. ?x*) (*lift-global-heap ?s*)

⟦*IS-VALID*(*data-struct1-C*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*data-struct1-C*)⟧ ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-data-struct1-C-map ?p* (λ*-. ?x*) (*lift-global-heap ?s*)

IS-VALID(*data-struct1-C*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-data-struct1-C-map ?p* (λ*-. ?x*) (*lift-global-heap ?s*)

⟦*IS-VALID*(*data-struct2-C*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*data-struct2-C*)⟧ ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-data-struct2-C-map ?p* (λ*-. ?x*) (*lift-global-heap ?s*)

IS-VALID(*data-struct2-C*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-data-struct2-C-map ?p* (λ*-. ?x*) (*lift-global-heap ?s*)

⟦*IS-VALID*(*inner-C*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*inner-C*)⟧ ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *heap-inner-C-map ?p* (λ*-. ?x*) (*lift-global-heap ?s*)

IS-VALID(*inner-C*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *heap-inner-C-map ?p* (λ*-. ?x*) (*lift-global-heap ?s*)

⟦*valid-array-base.valid-array* (λ*h*. *IS-VALID*(*inner-C*) *h*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*inner-C*[*5*])⟧ ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *inner-C.heap-array-map ?p* (λ*-. ?x*) (*lift-global-heap ?s*)

valid-array-base.valid-array (λ*h*. *IS-VALID*(*inner-C*) *h*) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-′-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *inner-C.heap-array-map ?p* (λ*-. ?x*) (*lift-global-heap ?s*)

⟦*IS-VALID*(*outer-array-C*) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of*

$TYPE(\textit{outer-array-C})\rrbracket \Longrightarrow \textit{lift-global-heap}\ (\textit{t-hrs-}'\textit{-update}\ (\textit{hrs-mem-update}$
$(\textit{heap-update-padding}\ ?p\ ?x\ ?bs))\ ?s) = \textit{heap-outer-array-C-map}\ ?p\ (\lambda\text{-}.\ ?x)$
$(\textit{lift-global-heap}\ ?s)$

$\textit{IS-VALID}(\textit{outer-array-C})\ (\textit{lift-global-heap}\ ?s)\ ?p \Longrightarrow \textit{lift-global-heap}\ (\textit{t-hrs-}'\textit{-update}$
$(\textit{hrs-mem-update}\ (\textit{heap-update}\ ?p\ ?x))\ ?s) = \textit{heap-outer-array-C-map}\ ?p$
$(\lambda\text{-}.\ ?x)\ (\textit{lift-global-heap}\ ?s)$

$\llbracket \textit{valid-array-base.valid-array}\ (\lambda h.\ \textit{IS-VALID}(\textit{data-C})\ h)\ (\textit{lift-global-heap}$
$?s)\ ?p;\ \textit{length}\ ?bs = \textit{size-of}\ TYPE(\textit{data-C}[10])\rrbracket \Longrightarrow \textit{lift-global-heap}\ (\textit{t-hrs-}'\textit{-update}$
$(\textit{hrs-mem-update}\ (\textit{heap-update-padding}\ ?p\ ?x\ ?bs))\ ?s) = \textit{data-C.heap-array-map}$
$?p\ (\lambda\text{-}.\ ?x)\ (\textit{lift-global-heap}\ ?s)$

$\textit{valid-array-base.valid-array}\ (\lambda h.\ \textit{IS-VALID}(\textit{data-C})\ h)\ (\textit{lift-global-heap}$
$?s)\ ?p \Longrightarrow \textit{lift-global-heap}\ (\textit{t-hrs-}'\textit{-update}\ (\textit{hrs-mem-update}\ (\textit{heap-update}\ ?p$
$?x))\ ?s) = \textit{data-C.heap-array-map}\ ?p\ (\lambda\text{-}.\ ?x)\ (\textit{lift-global-heap}\ ?s)$

$\llbracket \textit{IS-VALID}(\textit{data-array-C})\ (\textit{lift-global-heap}\ ?s)\ ?p;\ \textit{length}\ ?bs = \textit{size-of}$
$TYPE(\textit{data-array-C})\rrbracket \Longrightarrow \textit{lift-global-heap}\ (\textit{t-hrs-}'\textit{-update}\ (\textit{hrs-mem-update}$
$(\textit{heap-update-padding}\ ?p\ ?x\ ?bs))\ ?s) = \textit{heap-data-array-C-map}\ ?p\ (\lambda\text{-}.\ ?x)$
$(\textit{lift-global-heap}\ ?s)$

$\textit{IS-VALID}(\textit{data-array-C})\ (\textit{lift-global-heap}\ ?s)\ ?p \Longrightarrow \textit{lift-global-heap}\ (\textit{t-hrs-}'\textit{-update}$
$(\textit{hrs-mem-update}\ (\textit{heap-update}\ ?p\ ?x))\ ?s) = \textit{heap-data-array-C-map}\ ?p\ (\lambda\text{-}.$
$?x)\ (\textit{lift-global-heap}\ ?s)$

$\llbracket \textit{IS-VALID}(\textit{outer-C})\ (\textit{lift-global-heap}\ ?s)\ ?p;\ \textit{length}\ ?bs = \textit{size-of}\ TYPE(\textit{outer-C})\rrbracket$
$\Longrightarrow \textit{lift-global-heap}\ (\textit{t-hrs-}'\textit{-update}\ (\textit{hrs-mem-update}\ (\textit{heap-update-padding}$
$?p\ ?x\ ?bs))\ ?s) = \textit{heap-outer-C-map}\ ?p\ (\lambda\text{-}.\ ?x)\ (\textit{lift-global-heap}\ ?s)$

$\textit{IS-VALID}(\textit{outer-C})\ (\textit{lift-global-heap}\ ?s)\ ?p \Longrightarrow \textit{lift-global-heap}\ (\textit{t-hrs-}'\textit{-update}$
$(\textit{hrs-mem-update}\ (\textit{heap-update}\ ?p\ ?x))\ ?s) = \textit{heap-outer-C-map}\ ?p\ (\lambda\text{-}.\ ?x)$
$(\textit{lift-global-heap}\ ?s)$

$\llbracket \textit{IS-VALID}(\textit{other-C})\ (\textit{lift-global-heap}\ ?s)\ ?p;\ \textit{length}\ ?bs = \textit{size-of}\ TYPE(\textit{other-C})\rrbracket$
$\Longrightarrow \textit{lift-global-heap}\ (\textit{t-hrs-}'\textit{-update}\ (\textit{hrs-mem-update}\ (\textit{heap-update-padding}$
$?p\ ?x\ ?bs))\ ?s) = \textit{heap-other-C-map}\ ?p\ (\lambda\text{-}.\ ?x)\ (\textit{lift-global-heap}\ ?s)$

$\textit{IS-VALID}(\textit{other-C})\ (\textit{lift-global-heap}\ ?s)\ ?p \Longrightarrow \textit{lift-global-heap}\ (\textit{t-hrs-}'\textit{-update}$
$(\textit{hrs-mem-update}\ (\textit{heap-update}\ ?p\ ?x))\ ?s) = \textit{heap-other-C-map}\ ?p\ (\lambda\text{-}.\ ?x)$
$(\textit{lift-global-heap}\ ?s)$

$\llbracket \textit{valid-array-base.valid-array}\ (\lambda h.\ \textit{IS-VALID}(\textit{unpacked-C})\ h)\ (\textit{lift-global-heap}$
$?s)\ ?p;\ \textit{length}\ ?bs = \textit{size-of}\ TYPE(\textit{unpacked-C}[2])\rrbracket \Longrightarrow \textit{lift-global-heap}$
$(\textit{t-hrs-}'\textit{-update}\ (\textit{hrs-mem-update}\ (\textit{heap-update-padding}\ ?p\ ?x\ ?bs))\ ?s) = \textit{un-}$
$\textit{packed-C.heap-array-map}\ ?p\ (\lambda\text{-}.\ ?x)\ (\textit{lift-global-heap}\ ?s)$

$\textit{valid-array-base.valid-array}\ (\lambda h.\ \textit{IS-VALID}(\textit{unpacked-C})\ h)\ (\textit{lift-global-heap}$
$?s)\ ?p \Longrightarrow \textit{lift-global-heap}\ (\textit{t-hrs-}'\textit{-update}\ (\textit{hrs-mem-update}\ (\textit{heap-update}\ ?p$
$?x))\ ?s) = \textit{unpacked-C.heap-array-map}\ ?p\ (\lambda\text{-}.\ ?x)\ (\textit{lift-global-heap}\ ?s)$

$\llbracket \textit{IS-VALID}(\textit{array-C})\ (\textit{lift-global-heap}\ ?s)\ ?p;\ \textit{length}\ ?bs = \textit{size-of}\ TYPE(\textit{array-C})\rrbracket$
$\Longrightarrow \textit{lift-global-heap}\ (\textit{t-hrs-}'\textit{-update}\ (\textit{hrs-mem-update}\ (\textit{heap-update-padding}$
$?p\ ?x\ ?bs))\ ?s) = \textit{heap-array-C-map}\ ?p\ (\lambda\text{-}.\ ?x)\ (\textit{lift-global-heap}\ ?s)$

$\textit{IS-VALID}(\textit{array-C})\ (\textit{lift-global-heap}\ ?s)\ ?p \Longrightarrow \textit{lift-global-heap}\ (\textit{t-hrs-}'\textit{-update}$
$(\textit{hrs-mem-update}\ (\textit{heap-update}\ ?p\ ?x))\ ?s) = \textit{heap-array-C-map}\ ?p\ (\lambda\text{-}.\ ?x)$

(*lift-global-heap ?s*)
⟦*IS-VALID*(*unpacked-C*[*3*][*2*]) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*unpacked-C*[*3*][*2*])⟧ ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *unpacked-C.outer.heap-array-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)
IS-VALID(*unpacked-C*[*3*][*2*]) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *unpacked-C.outer.heap-array-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)
⟦*IS-VALID*(*inner-C*[*5*]) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*inner-C*[*5*])⟧ ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *inner-C.heap-array-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)
IS-VALID(*inner-C*[*5*]) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *inner-C.heap-array-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)
⟦*IS-VALID*(*data-C*[*10*]) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*data-C*[*10*])⟧ ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *data-C.heap-array-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)
IS-VALID(*data-C*[*10*]) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *data-C.heap-array-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)
⟦*IS-VALID*(*unpacked-C*[*2*]) (*lift-global-heap ?s*) *?p*; *length ?bs* = *size-of TYPE*(*unpacked-C*[*2*])⟧ ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update-padding ?p ?x ?bs*)) *?s*) = *unpacked-C.heap-array-map ?p* (λ-. *?x*) (*lift-global-heap ?s*)
IS-VALID(*unpacked-C*[*2*]) (*lift-global-heap ?s*) *?p* ⟹ *lift-global-heap* (*t-hrs-'-update* (*hrs-mem-update* (*heap-update ?p ?x*)) *?s*) = *unpacked-C.heap-array-map ?p* (λ-. *?x*) (*lift-global-heap ?s*).

thm *heap-update-fold-toplevel-fields-pointless*
thm *plift-heap-update-padding-heap-update-pointless-conv*
thm *lift-heap-update-padding-heap-update-conv*
thm *write-commutes*

No restriction to *packed-type*!

In the original version of AutoCorres there was a restriction of heap lifting to types that are also in *packed-type*. Those are types that internally have no padding fields at all. The reason was that certain lemmas especially regarding heap update become more involved in the presence of padding fields. A *heap-update* preserves the value of all padding bytes, whereas the abstract value obtained from a corresponding *h-val* is independent of the value of the padding bytes. So in principle these notions fit together well but it seems that some formal language was missing to reason about padding bytes. Meanwhile we added a theory to reason about those padding bytes in *AutoCorres2.Padding-Equivalence*. This made it possible to liberate

heap lifting from the restriction to *packed-type*. For example *heap-update ?p ?arr* = (λs. *foldl* ($\lambda s\ n$. *heap-update* (*array-ptr-index ?p False n*) (*?arr*.[*n*]) *s*) *s* [*0..<CARD*(*?'b*)]) for *packed-type* holds in general *c-guard ?p* $\Longrightarrow$ *heap-update ?p ?arr ?h* = *fold* (λi. *heap-update* (*array-ptr-index ?p False i*) (*?arr*.[*i*])) [*0..<CARD*(*?'b*)] *?h*.

thm *heap-update-array*
thm *heap-update-Array*

Padding Equivalence and *xmem-type*

Padding bytes and padding fields are introduced via the construction of new C-Types via the combinators *ti-pad-combine* and *ti-typ-pad-combine* to satisfy alignment properties. However, the concept of a padding byte is not a first-class citizen of a ($'a$, $'b$) *typ-desc*, but just happens to be a field with some special properties. In practice these fields are generated by *ti-pad-combine*, so we know that the field name starts with $''!pad''$, and the associated 'lense' for lookup and update have the 'passthrough' properties of a padding field.

We made these properties explicit by supplying a theory to identify padding-bytes in a list of bytes associated with an C-Type, and having notions to compare such byte-lists, e.g. telling if two byte lists are equal up-to the padding bytes, or if they agree on the padding bytes. This information is also backed into the properties of *xmem-type*, which is a subclass of *mem-type*. All primitive word and pointer types as well as all types that are constructed by the UMM module of the C-Parser are proved to belong to that class. The construction of array types propagates the class as expected.

Again the notions come in pairs of a 'typed' and a 'untyped' version. The 'typed' version associated with *typ-info-t* is a lense based formalisation, the 'untyped' version associated with *typ-uinfo-t* is based on byte lists.

The lense based version is introduced by *padding-lense*. Access and update follow the approach of $'a$ *field-desc*. Lookup / access has type $'a \Rightarrow$ *byte list* $\Rightarrow$ *byte list*. It takes an abstract value and a supply list of padding bytes and retrieves the bytes encoding the field. The intuition of the padding byte list is the following. It is supposed to bridge the gap between the abstract value and the byte representation. The abstract value is independent of the padding bytes. Hence in general there is no one-to-one correspondence between a byte list encoding of an abstract value and the abstract value. When converting from a byte list to an abstract value this is fine, the padding bytes are just ignored. But when we go the other way we have to invent the padding bytes. One solution could be to just fill up with zeros. Another solution could be to yield a non-deterministic result for the padding bytes. The UMM model chose another way. The possible valuation for padding bytes is supplied as an additional argument. So whenever a padding byte is needed we just take it from the position in that list. The update has type

byte list $\Rightarrow$ *'a* $\Rightarrow$ *'a*. It takes the value of a field, encoded as a byte list, and transforms it to an update on the abstract value.

context *padding-lense*
begin

The concept of a *padding-lense* follows the signature of *'a field-desc* and describes the notions of padding bytes as semantic properties of the access function *acc* and the update function *upd*. Based on them it introduces the notions of

- *local.is-padding-byte*, is a byte a padding with respect to the lense?
- *local.is-value-byte*, is a byte a value byte with respect to the lense, i.e. does the abstract value associated with the byte list depend on that byte.
- *local.eq-padding*, access cannot distinguish the byte lists.
- *local.eq-upto-padding*, update cannot distinguish the byte list.

thm *is-padding-byte-def*
thm *is-value-byte-def*
thm *eq-padding-def*
thm *eq-upto-padding-def*

As the definitions are semantically defined the effect on *acc* and *upd* are rather immediate. For example, if two byte lists are equal upto padding, then an update with *upd* yields the same result. If the padding bytes within two supply byte lists are the same, then *acc v* yields the same result for both byte lists.

end

In the untyped *typ-uinfo-t* we define the corresponding notions, as properties of the byte list instead of the access and update functions.

- *is-padding-byte*, a byte is a padding byte in a byte list, if normalisation of the byte list is independent of its value. Normalisation *norm-tu* is defined on *typ-uinfo-t* and sets all padding bytes to zero.
- *is-value-byte*, normalisation depends on the value of the byte.
- *eq-padding*, all padding bytes are the same.
- *eq-upto-padding*, all value bytes are the same.

thm *is-padding-byte-def*
thm *is-value-byte-def*
thm *eq-padding-def*

thm *eq-upto-padding-def*

For instances of *xmem-type* we can go back and forth between both characterisations.

thm *is-padding-byte-lense-conv*
thm *field-lookup-is-padding-byte-lense-conv*
thm *is-value-byte-lense-conv*
thm *field-lookup-is-value-byte-lense-conv*
thm *eq-padding-lense-conv*
thm *field-lookup-eq-padding-lense-conv*
thm *eq-upto-padding-lense-conv*
thm *field-lookup-eq-upto-padding-lense-conv*

With those theorems in place it is easy to show that padding fields only consist of padding bytes and thus do not account to the abstract value.

thm *field-lookup-access-ti-to-bytes-field-conv*
thm *access-ti-update-ti-eq-upto-padding*
thm *field-lookup-qualified-padding-field-name*(*1*)
thm *is-padding-tag-def padding-tag-def padding-desc-def*

UMM-Simprocs Cache

To solve sideconditions on UMM-Types we have implemented some simprocs within `UMM_Proofs`. Currently their scope is limited to the usecases we have in the proofs described above. It should be straightforward to extend them to more usecases. Their purpose is not to support abstract reasoning on types but to provide properties of concrete instances of C-types, for example deciding whether $TYPE(32\ word) \leq_\tau TYPE(outer\text{-}C)$ holds.

The benefit of the simprocs is that we do not have to guess and prove lots of lemmas already during the definition of a new UMM type, but can postpone it until we actually need them. Once they are proven they are added to the cache, so they are only proven once. In our use-case the lemmas we need depend on the addressable fields the user specifies on an **autocorres** invocation.

⟨*ML*⟩

The simprocs make use of a common cache. The cache itself is implemented as a simpset. Cache lookup means we try to rewrite with the cache:

- cache miss: term is unchanged.
- cache hit: term is changed.

  In case of a cache hit we are finished and return the resulting equation. In case of a cache miss we invoke the simplifier with a taylored simpset to derive a new equation. This equation is then added to the cache as well as returned from the simproc.

Some fine points of this setup are related to context management. Conceptually we only prove theory level theorems about an UMM type. So even if we prove them after the definition of the UMM type they should all be applicable as if we have immediately proven them at the point where the UMM type was generated. However, the simproc is invoked in some context later on where the theory has already advanced. In order to produce a result after a cache miss we have to somehow 'travel back in time' to the original theory context after the definition of the UMM type, such that the simplifier properly works on it and the result is reusable in other contexts. We maintain that original theory state and recertify terms before simplifying them.

Another point is, that the terms we attempt to simplify and cache might depend on each other. In order not to miss to cache intermediate results one has to carefully craft the simpsets.

In general the simplifier tries simprocs only after unconditional and conditional rules. So when a rewrite rule has already transformed the redex the simproc will never see that redex. This has two main implications:

- In order to have the simproc-based cache applied, there must not be a rule in the simpset that removes the redex before the cache actually has a chance to see it. This is why we maintain several simpsets to control that behaviour.
- When a cache miss was encountered and we apply the simplifier recursively to rewrite the redex we have to take care that the simproc is not invoked again on the same redex. When there is an unconditional rule in the simpset that rewrites that redex we are on the safe side. But beware of *conditional* rules. When the solver fails to solve the conditions the rule can fail and then the simproc is invoked again on the same redex. To prevent the setup from looping in these cases we maintain the redexes the simproc has already seen.

Here are some principles in the design of the simprocs and their simpsets:

- The type descriptions obtained by *typ-info-t* are not fully expanded. Instead we use a simplifier setup that works on the combinators, like *empty-typ-info*, *final-pad*, *ti-typ-pad-combine*. This maintains the more compact representation of a type-descriptor as a combinator expression, which is anyways the way it is originally defined.
- Fieldnames for array indexes stay symbolic: *replicate i ''1''*. We employ derived rules for array indexes and do not have to expand the type-descriptor of arrays. For these rules to apply we typically need the information that the index is in the bounds of the array type. In those cases the simproc generates a conditional rewrite rule.

- To test equality on *export-uinfo* t = *export-uinfo* s we normalise towards *export-uinfo* (*typ-info-t* $TYPE('a)$) = *export-uinfo* (*typ-info-t* $TYPE('b)$), where both $'a$ and $'b$ are concrete instances. Then when both expressions happen to be the same we have proved equality. Note that this approach is incomplete and in particular not sufficient to disprove the equality and prove the inequality. However, as these equality often appears as sidecondition in a conditional rewrite rule, not being able to prove the equality is somehow "equivalent" to disproving it: If we cannot prove the sidecondition the rule cannot be applied. The most prominent pattern for this is a *field-lookup* yielding the type-description of the selected field which is then compared to some type-description that is derived from a pointer type. To also disprove equality we make use of rule (*export-uinfo* (*typ-info-t* $TYPE(?'a)$) = *export-uinfo* (*typ-info-t* $TYPE(?'b)$)) = ($TYPE(?'a) \leq_\tau TYPE(?'b)$ $\wedge$ $TYPE(?'b) \leq_\tau TYPE(?'a)$) and use the simproc on *sub-typ*.

- To decide whether e.g. $TYPE(32\ word) \leq_\tau TYPE(outer\text{-}C)$ holds or not, we first try to disprove it by using type name information $[\![typ\text{-}name\ (typ\text{-}info\text{-}t\ TYPE(?'a)) \neq pad\text{-}typ\text{-}name; typ\text{-}name\ (typ\text{-}info\text{-}t\ TYPE(?'a)) \notin td\text{-}names\ (typ\text{-}info\text{-}t\ TYPE(?'b))]\!] \Longrightarrow \neg\ TYPE(?'a) \leq_\tau TYPE(?'b)$. both *typ-name* and *td-names* are supplied by the UMM package: *typ-name* (*typ-info-t* $TYPE(arr\text{-}struct\text{-}C)) \equiv ''arr\text{-}struct\text{-}C''$

  typ-name (*typ-info-t* $TYPE(data\text{-}struct2\text{-}C)) \equiv ''data\text{-}struct2\text{-}C''$

  typ-name (*typ-info-t* $TYPE(data\text{-}struct1\text{-}C)) \equiv ''data\text{-}struct1\text{-}C''$

  typ-name (*typ-info-t* $TYPE(data\text{-}array\text{-}C)) \equiv ''data\text{-}array\text{-}C''$

  typ-name (*typ-info-t* $TYPE(outer\text{-}array\text{-}C)) \equiv ''outer\text{-}array\text{-}C''$

  typ-name (*typ-info-t* $TYPE(two\text{-}dimensional\text{-}C)) \equiv ''two\text{-}dimensional\text{-}C''$

  typ-name (*typ-info-t* $TYPE(other\text{-}C)) \equiv ''other\text{-}C''$

  typ-name (*typ-info-t* $TYPE(array\text{-}C)) \equiv ''array\text{-}C''$

  typ-name (*typ-info-t* $TYPE(outer\text{-}C)) \equiv ''outer\text{-}C''$

  typ-name (*typ-info-t* $TYPE(unpacked\text{-}C)) \equiv ''unpacked\text{-}C''$

  typ-name (*typ-info-t* $TYPE(inner\text{-}C)) \equiv ''inner\text{-}C''$

  typ-name (*typ-info-t* $TYPE(closed\text{-}C)) \equiv ''closed\text{-}C''$

  typ-name (*typ-info-t* $TYPE(data\text{-}C)) \equiv ''data\text{-}C''$, *td-names* (*typ-info-t* $?x) \equiv \{''word0000010'', ''arr\text{-}struct\text{-}C'', ''word0000010\text{-}array\text{-}00110''\}$

  td-names (*typ-info-t* $?x) \equiv \{''data\text{-}C'', ''word0000010'', ''data\text{-}struct2\text{-}C'', ''data\text{-}struct1\text{-}C''\}$

  td-names (*typ-info-t* $?x) \equiv \{''data\text{-}C'', ''word0000010'', ''data\text{-}struct1\text{-}C''\}$

  td-names (*typ-info-t* $?x) \equiv \{''data\text{-}C'', ''word0000010'', ''data\text{-}array\text{-}C'', ''data\text{-}C\text{-}array\text{-}01010''\}$

td-names (*typ-info-t ?x*) ≡ {*''inner-C''*, *''closed-C''*, *''word0000010''*, *''outer-array-C''*, *''inner-C-array-1010''*}

td-names (*typ-info-t ?x*) ≡ {*''word00010''*, *''unpacked-C''*, *''word0000010''*, *''two-dimensional-C''*, *''unpacked-C-array-110''*, *''unpacked-C-array-110-array-010''*}

td-names (*typ-info-t ?x*) ≡ {*''other-C''*, *''closed-C''*, *''word00010''*, *''unpacked-C''*, *''word0000010''*}

td-names (*typ-info-t ?x*) ≡ {*''array-C''*, *''word00010''*, *''unpacked-C''*, *''word0000010''*, *''unpacked-C-array-010''*}

td-names (*typ-info-t ?x*) ≡ {*''inner-C''*, *''outer-C''*, *''closed-C''*, *''word0000010''*}

td-names (*typ-info-t ?x*) ≡ {*''word00010''*, *''unpacked-C''*, *''word0000010''*}

td-names (*typ-info-t ?x*) ≡ {*''inner-C''*, *''closed-C''*, *''word0000010''*}

td-names (*typ-info-t ?x*) ≡ {*''closed-C''*, *''word0000010''*}

td-names (*typ-info-t ?x*) ≡ {*''data-C''*, *''word0000010''*}. By construction every distinct type gets an individual name. If that does not work we try proving it instead, by using a tansitivity prover on the single step sub-type relations provided by $TYPE(\textit{32 signed word}) \leq_\tau TYPE(\textit{data-C})$

$TYPE(\textit{32 word}) \leq_\tau TYPE(\textit{data-C})$

$TYPE(\textit{32 word}) \leq_\tau TYPE(\textit{closed-C})$

$TYPE(\textit{32 word}) \leq_\tau TYPE(\textit{inner-C})$

$TYPE(\textit{closed-C}) \leq_\tau TYPE(\textit{inner-C})$

$TYPE(\textit{8 word}) \leq_\tau TYPE(\textit{unpacked-C})$

$TYPE(\textit{32 word}) \leq_\tau TYPE(\textit{unpacked-C})$

$TYPE(\textit{inner-C}) \leq_\tau TYPE(\textit{outer-C})$

$TYPE(\textit{32 word}) \leq_\tau TYPE(\textit{outer-C})$

$TYPE(\textit{unpacked-C}[2]) \leq_\tau TYPE(\textit{array-C})$

$TYPE(\textit{32 word}) \leq_\tau TYPE(\textit{array-C})$

$TYPE(\textit{32 word}) \leq_\tau TYPE(\textit{other-C})$

$TYPE(\textit{closed-C}) \leq_\tau TYPE(\textit{other-C})$

$TYPE(\textit{unpacked-C}) \leq_\tau TYPE(\textit{other-C})$

$TYPE(\textit{unpacked-C}[3][2]) \leq_\tau TYPE(\textit{two-dimensional-C})$

$TYPE(\textit{inner-C}[5]) \leq_\tau TYPE(\textit{outer-array-C})$

$TYPE(\textit{32 word}) \leq_\tau TYPE(\textit{outer-array-C})$

$TYPE(\textit{data-C}[10]) \leq_\tau TYPE(\textit{data-array-C})$

$TYPE(\textit{data-C}) \leq_\tau TYPE(\textit{data-struct1-C})$

$TYPE(data\text{-}struct1\text{-}C) \leq_\tau TYPE(data\text{-}struct2\text{-}C)$

$TYPE(32\ word[12]) \leq_\tau TYPE(arr\text{-}struct\text{-}C)$ and the rule for arrays $TYPE(?'a) \leq_\tau TYPE(?'a[?'b])$.

thm *not-sub-typ-via-td-name*
thm *typ-name-simps*
thm *td-names-simps*
thm *sub-typ-simps*
thm *element-typ-subtyp-array-typ*

Here some examples

lemma *export-uinfo* (*typ-info-t TYPE*(*array-C*)) = *export-uinfo* (*typ-info-t TYPE*(*other-C*))
⟨*proof*⟩

schematic-goal *field-lookup* (*typ-info-t TYPE*(*outer-C*)) [''*inner-C*'', ''*fld1-C*''] *0* = *?X*
⟨*proof*⟩

lemma *TYPE*(*8 word*) $\leq_\tau$ *TYPE*(*outer-C*) = *False*
⟨*proof*⟩

schematic-goal $i < 2 \Longrightarrow j < 3 \Longrightarrow$
field-ti TYPE(*two-dimensional-C*) [''*matrix-C*'', *replicate i CHR* ''*1*'', *replicate j CHR* ''*1*''] = *?X*
⟨*proof*⟩

lemma *TYPE*(*8 word*) $\leq_\tau$ *TYPE*(*8 word*[*42*])
⟨*proof*⟩

⟨*ML*⟩

26.21.6 Examples for normalisation of array indexes

lemma
fixes *p*::*array-C ptr*
assumes *bnd*[*simp*]: *i* < *2*
shows (*PTR*(*unpacked-C*) &(*p*→[''*elements-C*'', *replicate i CHR* ''*1*'']) = *PTR*(*unpacked-C*) &(*p*→[''*elements-C*'']) +$_p$ *int i*
⟨*proof*⟩

lemma
fixes *p*::*array-C ptr*
assumes *bnd*: (*i*::*32 word*) < *2*
shows (*PTR*(*unpacked-C*) &(*p*→[''*elements-C*'', *replicate* (*unat i*) *CHR* ''*1*'']) = *PTR*(*unpacked-C*) &(*p*→[''*elements-C*'']) +$_p$ *uint i*
⟨*proof*⟩

lemma
fixes *p*::*array-C ptr*
shows *do* {- <− *guard* (λ-. *unat* (*i*::*32 word*) < *2*);
return ((*PTR*(*unpacked-C*) &(*p*→[''*elements-C*'', *replicate* (*unat i*) *CHR* ''*1*'']))}
=
do {
- ← *guard* (λ*s*. *unat i* < *2*);
return (*PTR*(*unpacked-C*) &(*p*→[''*elements-C*'']) +$_p$ *uint i*)
}
⟨*proof*⟩

lemma
fixes *p*::*array-C ptr*
shows *do* {- <− *guard* (λ-. (*i*::*32 word*) < *2*);
return ((*PTR*(*unpacked-C*) &(*p*→[''*elements-C*'', *replicate* (*unat i*) *CHR* ''*1*'']))}
=
do {
- ← *guard* (λ*s*. *i* < *2*);
return (*PTR*(*unpacked-C*) &(*p*→[''*elements-C*'']) +$_p$ *uint i*)
}
⟨*proof*⟩

lemma
fixes *p*::*array-C ptr*
assumes *bnd*[*simp*]: *i* < *2*
shows (*PTR*(*unpacked-C*) &(*p*→[''*elements-C*'', *replicate i CHR* ''*1*'', ''*chr-C*'']))
=
PTR(*unpacked-C*)
&(*PTR*(*unpacked-C*) &(*p*→[''*elements-C*'']) +$_p$ *int i*→[''*chr-C*''])
⟨*proof*⟩

lemma (*i*::*32 word*) < *4* ⟹ *unat i* < *4*
⟨*proof*⟩

lemma (*i*::*32 signed word*) <*s* *4* ∧ *0* ≤*s* *i* ⟹ *nat* (*sint i*) < *4*
⟨*proof*⟩

26.21.7 Essence of Heap Lifting

Some birds eye view on what heap lifting is about. Consider the following C program.

```
typedef struct foo {
  int myint;
  bool mybool;
} foo;

int * p;
```

```
foo * q;

(*p) = 42;
i = q->myint;
b = q->mybool;
```

Is value i and b affected by update to $*p$
From the C-Parser we only get *c-guard* p and *c-guard* q:

```
{c_guard p \<and> c_guard q}
(*p) = 42;
i = q->myint;
b = q->mybool;
```

This only tells us something about alignment of pointers and that the pointer span does not overflow, but nothing about disjointness of the pointer spans.

What about stronger guarantees, if we assume well-typedness of the pointers:

```
{d \<Turnstile>\<^sub>t p \<and> d \<Turnstile>\<^sub>t p q}
(*p) = 42;
i = q->myint;
b = q->mybool;
```

Now we know that p might alias with $q->myint$ as they have the same type, but that pointer span p is actually disjoint form pointer span $q->mybool$. Hence for the read of b we can skip the heap update via p.

Now consider that the structure *foo* is a closed structure in the split heap model. This means that we have even stronger assumptions (which are guaranteed by guards in the lifted code):

```
{root_ptr_valid d p \<and> root_ptr_valid p q}
(*p) = 42;
i = q->myint;
b = q->mybool;
```

From this we can infer that the pointer spans p and the entire pointer span q don't overlap. hence reading i as well as b can skip the heap update via p.

In general the heap lifting phase introduces the potentially weaker guards for open types:

```
{ptr_valid d p \<and> ptr_valid p q}
(*p) = 42;
i = q->myint;
b = q->mybool;
```

So whether i might be affected by p depends on whether field *myint* is addressable or not.

The essential semantic part of heap lifting is to introduce these guards at every pointer access. In a second step these guards then allow a change in a representation from the monolitic heap to the split heap, with a separate heap for each atomic type. But in a sense this second step could be viewed as optional or cosmetic in the semantic sense. There is all the information available to perform the "skipping" steps as sketched above in the monolithic heap. This is what is performed in the simulation proof. So the question is if we could omit introduction of "lifted globals" completely and provide the necessary automation to the user to simplify accesses on the monolithic heap that mimic that behaviour in the split heap. In a sense instead of doing the simulation proof upfront it is done adhoc at every heap access / update. This avoids to provide quadratically many commutation lemmas (in the number of fields of open types) that are introduces for the lifted globals.

From the perspective of the simulation proof alone such a "virtual" split heap, which does not introduce a new lifted-globals type, seems to be equivalent to the one which introduces a new type with concrete distinct record fields for each fundamental heap. The simulation proof only works when every pointer access is guarded and under that assumption the two representations behave the same.

However, besides the simulation there are other aspects of the split heap that might pay off for verification or further abstractions. The two models behave differently for invalid pointers: In the split-heap (with different record fields), updates to one heap still do not affect other heaps but we cannot properly model such updates in the monolithic heap at all.

Another aspect was introduced later on for stack allocation: non-deterministic padding bytes. The *typ-heap-simulation* was generalised to state that lifting to the split heap is invariant for arbitrary padding bytes, i.e. *write-simulation*. The lifted heap only cares about the abstract values were the values of the padding bytes become irrelevant.

thm *h-val-heap-update-padding*
thm *plift-heap-update-padding-heap-update-pointless-conv*
thm *lift-heap-update-padding-heap-update-conv*

context *open-struct-all-corres*
begin
thm *ts-def*
thm *ac-corres*
end

context *ts-definition-access-arr-struct*

begin
thm *access-arr-struct'-def* [*no-vars*]
lemma *access-arr-struct' p* ≡ *do* {
  i ← *whileLoop* (λ*i s*. *i* < *6*)
        (λ*i*. *do* {
              - ← *guard* (λ*s*. *IS-VALID*(*arr-struct-C*) *s p*);
              - ← *modify* (*heap-arr-struct-C-update* (λ*a*. *a*(*p* := *arr2-C-update* (λ*a*. *Arrays.update a* (*unat i*) *3*) (*a p*))));
              - ← *guard* (λ*s*. *i* + *1* < *0xC*);
                  - ← *modify* (*heap-arr-struct-C-update* (λ*a*. *a*(*p* := *arr2-C-update* (*fupdate* (*unat* (*i* + *1*)) (λ*v*. *v* + *i*)) (*a p*))));
              return (*i* + *1*)
            })
        0;
  return 1
}
  ⟨*proof*⟩
end

end
theory *AutoCorres-Documentation*
imports
  AutoCorresInfrastructure
  pointers-to-locals
  In-Out-Parameters-Ex
  fnptr
  union-ac
  open-struct
begin

end

Chapter 27

C-Translation Infrastructure

theory *CTranslationInfrastructure*
imports
CTranslation
begin

term *guarded-spec-body*

27.1 Local Variables

Local variables in the SIMPL outcome of the C-Parser are represented in *locals*. The natural number in the domain encodes the canonical position in the scope of the C-function: input arguments, then return variable, then local variables. The byte list in the range of the function is the encoding of the value according to the UMM (unified memory model) *c-type*. The typing of variables is maintained as data in `CLocals` and a typed view is achieved via *clookup* and *cupdate*.

In previous attempts local variables were represented first as a **record** and more recently as **statespace**. The representation as *locals* tries to combine the best of all previous approaches:

- Uniform (program independent) representation of local variables.
- Positional 'naming' enables a canonical parameter passing for function calls, even if the names of parameters are unknown, as in the case of function-pointers.
- Lightweight "typing" via ML infrastructure instead of the 'fixes' of **statespace**. Unfortunately, the more flexible **statespace** representation for local variables turned out to reveal some performance bottle neck in the locale infrastructure, which itself boils down to the resources consumed by the omnipresent instantiation of types and terms.

declare [[*ML-print-depth = 1000*]]

27.1.1 Basic ML primitives

To define new variables the basic primitive is `CLocals.define_locals` which takes a scope (currently a two element list of qualifiers: program-name followed by function-name) and the list of local variable declarations (name, type and kind). Note that the order is relevant as it represents the canonical positional ordering.

⟨*ML*⟩

For each local variable a constant is defined with the position of the variable. The constant name encodes the original name of the local variable. Whenever available we use this symbolic name in favour of the plain literal number. However, at some places we use the plain numbers, in particular for parameter passing. There is some simplifier setup which we explain later that allows to use either representation.

To enable syntax translations from short names to the symbolic constants together with their typing one has to enter the correct scope, e.g.:

⟨*ML*⟩

term *prog.myfun.p-′*
thm *prog.myfun.p-′-def*

Some remarks on the naming scheme for those symbolic constants. The qualifiers for both program-name and function-name are mandatory. The name mangling with the suffix serves two purposes:

- Avoid name clashes with user input via variable names in the C sources (in particular for the artificial return variable)
- Avoid name clashes for bound or free variables (or new constants) in HOL. Note that even a constant with mandatory qualifier e.g. *foo.c* would force the pretty printer to avoid a bound variable being named just as *c*. The pretty printer only takes the `Long_Name.base_name` into account.

  The general suffix *-′* avoids clashes with C variable names as the prime is not a legal character for C identifiers. Syntax translations will truncate this suffix. The artificial return variable *ret* that the C-Parser introduces is internally *ret′-′* to avoid a potential conflict with a local variable named *ret*.

⟨*ML*⟩

27.1.2 Syntax

Plain lookup and update in locals.

term ‹*l* · *p*›

⟨*ML*⟩

term ‹$s\langle x := y\rangle$›
term ‹$s\langle x := y,\ p := k\rangle$›

⟨*ML*⟩

Lookup and update in a complete Simpl state, selecting the *locals*

term ‹$s\ \cdot_{\mathcal{L}}\ p$›
⟨*ML*⟩

term ‹$s\langle x :=_{\mathcal{L}}\ y\rangle$›
term ‹$s\langle x :=_{\mathcal{L}}\ y,\ p :=_{\mathcal{L}}\ k\rangle$›

27.1.3 Simplifier setup

The simplifier is setup to be able to deal with symbolic as well as literal numerals. Symbolic ones are expanded on the fly employing the infrastructure form the code generator. The trigger for this expansion is via a simplification procedure that. Conditional rules can be configured via the attribute:

⟨*ML*⟩

The simplification procedure first checks wheter the precondition evaluates to true, via the code-generator. In case of success the simplifier is invoked to replay that proof and trigger the rule. `Code_Simproc.code_simp_prems`, `Code_Simproc.code_prove`.

Note that the attribute also consumes an optional name to index the simprocs. This name is taken from **named-theorems**.

The relevant setup of the rules can be found in `CLocals.thy`.

schematic-goal ‹(*clookup 2* (*cupdate prog.myfun.z-′* (λ-. *2*::*64 word*) *l*)) = (*?X*::*64 word*)›
⟨*proof*⟩

schematic-goal ‹(*clookup prog.myfun.z-′* (*cupdate prog.myfun.z-′* (λ-. *2*::*64 word*) *l*)) = (*?X*::*64 word*)›
⟨*proof*⟩
schematic-goal $s\langle x := \lambda\text{-}.\ 2\rangle \cdot x = ?X$
⟨*proof*⟩

schematic-goal $s\langle x := \lambda\text{-}.\ 2\rangle \cdot p = ?X$
⟨*proof*⟩

install-C-file *ex.c*

thm *size-td-simps*
thm *my-struct-C-update-const*

The root locale storing the global content *ex-global-addresses*. This is also the locale ware the bodies of the imported functions are defined.

context *ex-global-addresses*
begin

The usage of global variables is analysed and three cases are distinguesed:

- static variables do not change there value. So they are defined as HOL constant with their initial value. In case no initialiser is given they are zero initialised.
- 'ordinary' global variables that are also updated somewhere in the code are stored in the record of global variables *globals*. They are accessed by the lookup and update function of the **record** package.
- In case the address of a global variable is taken somewhere in the code, we declare a global constant for a pointer to that variable. Lookup and update is then indirectly via the heap. Note that the pointer is only declared. There is no defining equation. Currently there are also no assumptions maintained about distinctness of global variable pointers. This is up to the user.

thm *global-const-defs*
term *g-static*
term *g-ordinary-'* **term** *g-ordinary-'-update*
term *g-addressed-'*
thm *main-body-def*
thm *globals.typing.get-upd*
thm *state.typing.get-upd*

Functions also result in a declaration of a constant representing the global function pointer. *ex.add*. They have the type *unit ptr*.

term *ex.add*

When the code actually uses function pointers to perform indirect calls, some more infrastructure is provided, cf `../doc/fnptr.thy`

When looking into the function bodies the symbolic names for the positional local variables are displayed, fully qualified with program name as well as function name.

thm *add-body-def*

To have short names printed, one can either enter the scope of the function by activating the corresponding *variables* bundle, or use the option to always display short names.

context includes *add-variables*
begin

thm *add-body-def*
end

declare [[*clocals-short-names*]]
thm *add-body-def*
end

The canonical locale to do verification of a procedure is the *impl* locale. This activates the scope of the function and also stores the equation that the lookup of the function pointer in the environment Γ retrieves the expected body.

context *add-impl*
begin
thm *add-body-def*
thm *add-impl*

The symbolic names can be folded and unfolded by an attribute.

thm *add-body-def* [*unfold-locals*]
thm *add-body-def* [*unfold-locals*, *fold-locals*]

These attributes can also take qualifier in case an alternative scope should be added

thm *call-add-body-def*
thm *call-add-body-def* [*unfold-locals*] — nothing happens as we are in the wrong scope
thm *call-add-body-def* [*unfold-locals call-add*] — now they are expanded as we qualify the scope

end

context *call-add-impl*
begin
thm *call-add-body-def*
declare [[*hoare-use-call-tr′* = *false*]]
thm *call-add-body-def*
end

All the implementations of the program are gathered in the *simpl* locale

context *ex-simpl*
begin
thm *add-impl*
thm *call-add-impl*
end

27.2 Infrastructure for states

type-synonym *state* = (*globals*, *locals*, *32 signed word*) *CProof.state*

context *add-impl*
begin

As a final part of the verification generator generator terms containing local and global variables are generalised to a 'splitted' view, where each component becomes has a bound variable.

lemma
$\exists s{::}state.\ s \cdot_{\mathcal{L}} n = 3$
⟨*proof*⟩

lemma
$\exists s{::}state.\ s \cdot_{\mathcal{L}} n = 3$
⟨*proof*⟩

lemma
$\exists s{::}state.\ s \cdot_{\mathcal{L}} n = 3 \wedge s \cdot_{\mathcal{L}} m = 3$
⟨*proof*⟩

lemma
$\exists s{::}state.\ s \cdot_{\mathcal{L}} n = 3 \wedge s \cdot_{\mathcal{L}} m = 3 \wedge g\text{-}'\ (globals\ s) = 4 \wedge global\text{-}exn\text{-}var'\text{-}'\ s = Break$
⟨*proof*⟩

Simpl syntax

⟨*ML*⟩
term ‹{| ´n = x |}›
⟨*ML*⟩
term ‹´n :== x ›

term ‹´n :== CALL add(x, y)›
term ‹´n :== CALL ex.add(x, y)›
⟨*ML*⟩

term ‹´ret' :== PROC add(2, 3)›

end

27.3 Cached simproc examples

Some more explanation on this is in `../doc/AutoCorresInfrastructure.thy`

schematic-goal $TYPE(addr\text{-}bitsize\ word) \leq_\tau TYPE(unpacked\text{-}C[12])$
⟨*proof*⟩

lemma $TYPE(8\ word) \leq_\tau TYPE(array\text{-}C)$
⟨*proof*⟩

lemma $TYPE(8\ word) <_\tau TYPE(array\text{-}C)$
⟨*proof*⟩

lemma $TYPE(addr\text{-}bitsize\ word) \leq_\tau TYPE(matrix\text{-}C)$
⟨*proof*⟩

lemma $TYPE(8\ word) \leq_\tau TYPE(matrix\text{-}C)$
⟨*proof*⟩

⟨*ML*⟩

schematic-goal ‹
field-names-no-padding (*typ-info-t* $TYPE(outer\text{-}C)$) (*export-uinfo* (*typ-info-t* $TYPE(inner\text{-}C)$))
= *?XX*›
⟨*proof*⟩
⟨*ML*⟩

schematic-goal ‹
set (*field-names-no-padding* (*typ-info-t* $TYPE(outer\text{-}C)$) (*export-uinfo* (*typ-info-t* $TYPE(inner\text{-}C)$))) = *?XX*›
⟨*proof*⟩
⟨*ML*⟩

lemma *export-uinfo* (*typ-info-t* $TYPE(unpacked\text{-}C)$) = *export-uinfo* (*typ-info-t* ($TYPE(32\ word)$))
⟨*proof*⟩

lemma *typ-uinfo-t* $TYPE(unpacked\text{-}C)$ = *export-uinfo* (*typ-info-t* ($TYPE(32\ word)$))
⟨*proof*⟩

lemma *typ-uinfo-t* $TYPE(32\ signed\ word)$ = *export-uinfo* (*typ-info-t* ($TYPE(32\ word)$))
⟨*proof*⟩

lemma *typ-uinfo-t* $TYPE(32\ signed\ word[3])$ = *export-uinfo* (*typ-info-t* ($TYPE(32\ word[3])$))
⟨*proof*⟩

schematic-goal ‹
set (*field-names-no-padding* (*typ-info-t* $TYPE(unpacked\text{-}C)$) (*export-uinfo* (*typ-info-t* $TYPE(8\ word)$))) = *?XX*›
⟨*proof*⟩

lemma *export-uinfo* (*typ-info-t* $TYPE(unpacked\text{-}C)$) = *export-uinfo* (*typ-info-t* ($TYPE(unpacked\text{-}C)$))
⟨*proof*⟩

⟨*ML*⟩
lemma *TYPE* (*32 word*) $\leq_\tau$ *TYPE*(*32 word*[*256*])
⟨*proof*⟩

lemma *TYPE* (*32 word*) $\leq_\tau$ *TYPE*(*32 word*[*256*])
⟨*proof*⟩
⟨*ML*⟩

schematic-goal *n* < *12* ⟹ *field-lookup* (*typ-info-t TYPE*(*array-C*)) [*''elements-C''*, *replicate n CHR ''1''*, *''chr-C''*] *0* = *?X*
⟨*proof*⟩

schematic-goal *n* < *12* ⟹ *field-lookup* (*typ-uinfo-t TYPE*(*array-C*)) [*''elements-C''*, *replicate n CHR ''1''*, *''chr-C''*] *0* = *?X*
⟨*proof*⟩

schematic-goal *n* < *12* ⟹ *field-ti TYPE*(*array-C*) [*''elements-C''*, *replicate n CHR ''1''*, *''chr-C''*] = *?X*
⟨*proof*⟩

schematic-goal *n* < *12* ⟹ *field-ti TYPE*(*array-C*) [*''elements-C''*, *replicate n CHR ''1''*, *''chr-C''*] = *?X*
⟨*proof*⟩

lemma *TYPE*(*unpacked-C*[*12*]) $\leq_\tau$ *TYPE*(*array-C*[*2*])
⟨*proof*⟩

lemma ¬ *TYPE*(*unpacked-C*[*12*]) $\leq_\tau$ *TYPE*(*unpacked-C*[*24*])
⟨*proof*⟩

lemma *TYPE*(*unpacked-C*[*12*]) $\leq_\tau$ *TYPE*(*array-C*)
⟨*proof*⟩

lemma *typ-uinfo-t TYPE*(*unit ptr*) ≠ *typ-uinfo-t TYPE*(*32 word ptr*)
⟨*proof*⟩

Matching of the interpreteted locale *stack-heap-raw-state*

⟨*ML*⟩

thm *zero-simps*
thm *make-zero*
thm *size-simps*
thm *size-align-simps*

thm *h-val-fields*
thm *heap-update-fields*
thm *fg-cons-simps*

thm *fl-ti-simps*
thm *fl-Some-simps*

end

Chapter 28

Translation of the StrictC Dialect (Outdated)

Michael Norrish

28.1 Introduction

This report describes the translation program that imports C source into a running Isabelle/HOL process, making a series of Isabelle definitions in the process, as well as discharging a number of (relatively minor) proof obligations that arise along the way.

The source code for the translator is found in the directory `c-parser`. Source files, *e.g.*, `file.ML`, are found in this directory unless otherwise noted.

The translation expects its input to be a well-formed C source file. Such a source file must additionally satisfy a number of other constraints, giving rise to a subset of C that is here called "StrictC". Files in StrictC may also include special annotations intended only for consumption by Isabelle (and the human code-verifier).

In fact, there are two important StrictC programs: the translation program, and the analysis program. The analysis program is entirely independent of Isabelle, and can be used to check that a source file conforms to the StrictC subset. It also implements a number of analyses that can be performed on source code. For example, it can output the input file's call-graph, and can list the globals that are read or modified in each function. The additional source-files supporting the analysis program are found in

```
c-parser/standalone-parser
```

The rest of this report describes both the functionality provided by these programs (focussing on the translator), how this functionality is implemented, and *where* it is implemented. The aim is to give a picture of the systems' design in a way that should make future modification of the code possible.

28.1.1 StrictC Subset Summary

This is a brief list summarising the simplest restrictions imposed by the StrictC subset.

- No `goto` statements.
- No fall-through cases in `switch` statements. Cases can be terminated with `continue` and `return`, as well as `break`.
- Labels for `switch` statement cases must all appear at the syntactic level immediately below the block statement that must appear below the `switch` statement.
- No unions. (These are handled by a separate tool; see Cock [1].)
- No `struct`, `enum` or `typedef` declarations anywhere except at the top, global, level.

28.2 Abstract Syntax

The core data types in the translator represent the input program. These abstract syntax values are the product of parsing the concrete syntax (see Appendix 28.6 for the grammar used), and is the subsequent input to all further analyses and translation. The abstract syntax declarations are given in `Absyn.ML`. For example, the definition of the syntax type corresponding to C statements is given in Figure 28.1. There are also definitions for C types, expressions, and declarations in `Absyn.ML`.

The type `statement` is actually a `statement_node` wrapped inside a "region" (see `Region.ML` and Section 28.2.1 below), which provides information about where the original concrete syntax originated. This is used for providing error messages.

All strings in the statement declaration correspond to Isabelle terms (*e.g.*, the loop invariant in the `While` case). These will be parsed as such later in the translation process, but are just uninterpreted strings when the C parser finishes. Function calls can return their results into l-values, have the return value ignored, or have the return value itself `return`-ed. The first option corresponds to having the `expr option` argument be `SOME e` in the `AssignFnCall` constructor, the second would have that parameter be `NONE`, and the last is handled by the `ReturnFnCall` constructor.

Syntactically, these options correspond to writing

```
var = f(x,y);
```

or

```
f(x,y);
```

```
datatype statement_node =
      Assign of expr * expr
    | AssignFnCall of expr option * expr * expr list
    | EmbFnCall of expr * expr * expr list
    | Block of block_item list
    | While of expr * string wrap option * statement
    | Trap of trappable * statement
    | Return of expr option
    | ReturnFnCall of expr * expr list
    | Break
    | Continue
    | IfStmt of expr * statement * statement
    | Switch of expr * (expr option list * block_item list) list
    | EmptyStmt
    | Auxupd of string
    | Spec of ((string * string) * statement list * string)
    | AsmStmt of {volatilep : bool, asmblock : asmblock}
and statement = Stmt of statement_node Region.Wrap.t
and block_item =
    BI_Stmt of statement
  | BI_Decl of declaration wrap
```

Figure 28.1: The Abstract Syntax Data Type for C Statements

or

```
return f(x,y);
```

C99 Block Items Conforming to the C99 grammar, the input language allows declarations at any point inside a block, not just in a sequence at the head of the block. In other words, the following

```
{
  x = f(z);
  int y = x + 1;
  while (x < y) { ... }
}
```

is legal in C99. This means that a block has to take a list of `block_item` values as an argument, where a `block_item` is either a statement or a declaration.

Syntactic Sugar for Loops The abstract syntax has just one form of loop, the `While` constructor. The optional string argument to `While` is used to represent any user-supplied invariant. Together with the `Trap` constructor, this is used to implement all three forms of C loop. The translation follows the model from Norrish's PhD [6, p60]. It also supports `for`-loops with declarations in the first position. This latter, for example,

```
for (int i = 3; i < 10; i++) ...
```

is a feature of C99.

Breaking from C, the grammar has the third component of the `for` loop form be a restricted form of statement, rather than an expression. The parser syntax only allows comma-separated increments (*i.e.*, `++`), decrements and assignments.

The Isabelle translation eventually compiles all loops to one underlying the loop primitive in the VCG environment called `While`. This form does not handle exceptional control-flow forms like `break` and `continue`. These are instead handled by the `Trap` constructor, mapping to the VCG language's `TRY-CATCH` form.

28.2.1 Regions

The `Region` module implements a method for annotating arbitrary data types with location information. This module has been taken from the `MLton` compiler project (which has a BSD-style open source licence). It is used a great deal in the system, and its use could probably be extended still further. Region information is used to produce good error messages.

The basic type is that of the `region` which is essentially a pair of "source positions" (which are in turn implemented in `SourcePos.ML`). One useful source position is `SourcePos.bogus`, corresponding to "nowhere" (perhaps because some syntax has been conjured out of nowhere and doesn't really exist in a file).

Regions are then used to implement the concept of a "wrap" (SML type `Region.Wrap.t`), a polymorphic data type. The file `Absyn.ML` declares the following abbreviation:

```
type 'a wrap = 'a Region.Wrap.t
```

An `'a wrap` (read "alpha wrap") is an `'a` value coupled with a region. The important functions for manipulating wraps are

```
val wrap    : 'a * SourcePos.t * SourcePos.t -> 'a wrap
val bogwrap : 'a -> 'a wrap
val left    : 'a wrap -> SourcePos.t
val right   : 'a wrap -> SourcePos.t
val node    : 'a wrap -> 'a
val apnode  : ('a -> 'b) -> 'a wrap -> 'b wrap
```

For example, the grammar code in `StrictC.grm` manipulates a number of values of type `string wrap`. If an error relating to this value arises, both the string and its position can be reported to the user.

Things become more complicated when the type to be wrapped is recursive. The standard idiom in the project is illustrated with the definition of the `statement` data type (shown in Figure 28.1). The constructors for values of the type are actually given in an auxiliary type (`statement_node`), but the recursive constructors take arguments of type `statement`. The type `statement` is then a type that is mutually recursive with `statement_node`, and which has just one constructor, `Stmt`.[1]

Because a `statement` is not a wrap, the project cannot directly call functions like `node` on `statement` values. Instead, helper functions are declared:

```
val sleft  : statement -> SourcePos.t
val sright : statement -> SourcePos.t
val snode  : statement -> statement_node
val swrap  : SourcePos.t * SourcePos.t * statement_node ->
             statement
```

When code wishes to pattern-match against the multiple possible forms a statement `s` may have, the idiom is to write

[1] It would be nice if one could just declare `statement` to be an abbreviation of `statement_node wrap`, but SML doesn't permit this. It must be a `datatype` itself, and thus must have at least one constructor.

```
case snode s of
  EmptyStmt => ...
| While(g,inv,body) => ...
| IfStmt(g,ts,es) => ...
| ...
```

The strength of this idiom is that one *always* manipulates values of type `statement`. In particular, if the case analysis above is to make recursive calls of its analysis on sub-statements such as `body`, `ts` and `es`, these values are of the correct type for this to be done immediately.

The `expr` (expression) type (home to constructors such as `Deref`, `Var` and `TypeCast`) is set up in the same style, giving rise to functions `eleft`, `eright`, `enode`, `ewrap` and `ebogwrap`.

The type of C types The type of C types is `'a ctype`, with constructors such as `Void` and `Ptr`. Though recursive, it doesn't use the wrap idiom. On the other hand, this type is polymorphic. The `'a` parameter is instantiated with an SML type that corresponds to the forms that give the size of arrays when they are declared. This type parameter is instantiated with `expr` when the input file is first parsed. In this way, a declaration like

```
unsigned char array[EnumConst1 * sizeof(int*)];
```

can be handled. Thus, the `VarDecl` constructor of the `declaration` type

```
val VarDecl :
      expr ctype * string wrap * bool * initializer option ->
      declaration
```

takes an `expr ctype` as its first parameter. Within subsequent phases of the analysis and translation, it is much more convenient to work with values of type `int ctype`, where the (constant) expression has been evaluated. For example, the `get_rettype` function from `program_analysis.ML`, takes a `csenv` value (see Section 28.3 below) and the name of a function, and returns the return type of the function. The value returned is an `int ctype`. The conversion of an `expr ctype` into an `int ctype` is done by the function `Absyn.constify_abtype`.

28.3 The Symbol Table

All of the work done in analysis and translation of StrictC revolves around the information stored in two important data structures implemented in the module `program-analysis.ML`. The first of these is the `var_info` type. This stores information about individual variables. The second type, `csenv` ("C state environment"), stores information about the program as a whole,

including its collection of variables, but also recording details such as where variables are read and modified, and the program's call-graph structure.

The `var_info` type stores information about declared identifiers living in the name-space that encompasses normal objects, functions and enumeration constants. In addition to the type of the variable (*e.g.*, `int`, `char *` *etc.*), the `var_info` also includes information about where the variable was declared in terms of program locations, and also in terms of scope (it might be global, or declared local to a particular function).

The `csenv` type accumulates its information about the program by performing traversals of the abstract syntax tree. **The StrictC translator makes no effort to be a one-pass compiler**, but the number of traversals is no greater than three, and will probably be reduced in future versions of the implementation. These traversals are performed after the parser has constructed all of the tree. There are also places in this analysis where **the translator assumes that it has seen the whole program**. In particular, the translator cannot be used to translate *translation units* that are to be separately compiled. It must be presented with a concatenation of the complete sources.

The bulk of the API for manipulating values of type `csenv` is concerned with pulling information out of the symbol table. For example, it is possible to calculate the type of a C expression with the function

```
val cse_typing : csenv -> expr -> int ctype
```

In addition, `program-analysis.ML` contains the one entry-point for taking a sequence of external declarations (once parsed) and creating a `csenv` value:

```
val process_decls :
  Absyn.ext_decl list ->
  ((Absyn.ext_decl list * Absyn.statement list) * csenv)
```

The return type includes a modified version of the syntax that was provided as input, a list of the initialising assignments for the global variables, and the `csenv` value.

28.3.1 Functional Record Updates in SML

The code in `program-analysis.ML` uses a powerful, but cryptic SML idiom that makes it easy to define SML records along with functions for updating their fields. Done naïvely, writing code to do this represents a quadratic amount of work for the programmer. Put another way, the naïve approach requires $O(n)$ much typing whenever a field is added to or removed from a record definition of n fields.

The cryptic technique is fully described at

http://mlton.org/FunctionalRecordUpdate

and allows the addition or deletion of a field to be done with $O(1)$ much typing. Supporting code is in `FunctionalRecordUpdate.ML`.

The cryptic code is isolated within `program-analysis.ML`, where it is used to define update functions that are subsequently used exclusively. In general, when one of the two types has a field `fld` of type τ, then there will typically be a function {`cse`|`vi`}`_fupd_fld` defined, with type

$$(\tau \to \tau) \to \mathsf{rcd} \to \mathsf{rcd}$$

where rcd is either `var_info` or `csenv`. Such functions can be used to update the fields of a record: the user provides a function that is given the old value of the field, and which returns the new value.

28.4 Creation of the Hoare Environment State

The major oddity about Norbert Schirmer's Hoare environment, into which we are translating our programs, is that all local variables, including function parameters, have to be part of the "global state". This state must be declared before any functions can be translated, because a function becomes an Isabelle definition (conceptually at least) that operates over that state space.

Slightly simplifying, the state space is an Isabelle type that is a record with fields for every local and global variable. Each field has a type (Isabelle/HOL is a typed logic after all), which means that all local variables of the same name in the same StrictC translation unit must have the same type. This is rather an arbitrary requirement, but easy both to enforce and to comply with. Thankfully, signed and unsigned variants of the same underlying type (such as `signed short` and `unsigned short`) are given the same Isabelle/HOL type, so there is a little leeway. Nonetheless, if `i` is of type `int` in one function, it can not be of type `char` in another.[2]

The arrangement of global and local variables is actually slightly complicated. In essence, the state-space is set up to look like:

```
statespace = record
   globals :: global_var_type
   local_var1 :: lvar1_type
   local_var2 :: lvar2_type
   ...
   local_varn :: lvarn_type
```

[2]If this is attempted, the system will "munge" one of the variables so that it has a different name when translated into Isabelle. The "munged" name is stored in the variable's `var_info` record.

where the field `globals` is of a custom record type `global_var_type` that in turn contains all of the global variables. In addition to the user program's own globals, the translation process adds two extra global variables of its own. These variables are used for handling "exceptional exits" (such as those caused by the `break`, `continue` and `return` statements), and for modelling the global heap. The exact names of these variables is not important, here we will refer to them as `global_exn_var`, and `global_heap`.

These two special variables are part of `global_var_type` and so must have Isabelle types themselves. The type of `global_exn_var` is the enumerated type `c_exntype`, defined in the Isabelle theory `CProof` to have three possible values, `Break`, `Return` and `Continue`. The type of the global heap `global_heap` field is a product of underlying heap contents (a map of type `addr` $\rightarrow$ `word8`) and a special-purpose data type to store type information about the heap memory (see Harvey Tuch's PhD thesis [9] for more on this).

28.4.1 Representing Values in Memory

There is one important requirement that must be met by all object types that occur in C programs: they must be representable in memory. Alternatively, it must be possible to encode values of the types in a program as sequences of bytes, and to then decode those same bytes back into the original values. One approach to modelling this fundamental requirement might be to have Isabelle functions that manipulated only lists of bytes. Working at the level of this untyped, and very concrete, view of the program state would be an extremely poor state of affairs (the C programmer would have a more abstract view of the program than the verifier).

Our approach is to use Isabelle type-classes to encode the fact that an Isabelle type can be represented in a consistent amount of "C memory". When an Isabelle type `'a` is in the class `mem_type`[3], it supports functions

```
to_bytes   :: "'a::mem_type => byte list"
from_bytes :: "byte list => 'a::mem_type"
```

as well as a number of other supporting functions that record details like the fields that occur in compound `struct` types, and the (constant) length of the byte-lists that encode the values in the type.

All of the atomic types manipulated by our programs are fixed-width words (*e.g.*, 32 bit words for integers). It is easy to demonstrate that these types are indeed in the `mem_type` class. In particular, we do *not* pretend that programs manipulate infinite precision integers, and then worry about whether or not these integers can be pushed into and pulled out of memory. All of the arithmetic performed by the programs we verify is done at fixed widths, respecting the underlying machine's operations. Additionally, using

[3] See `umm_heap/CTypes.thy`.

the techniques from Section 28.5.1, we trap the undefined behaviour caused by overflow on signed values.

28.4.2 Pointers

Pointers are always represented as words of a particular size (regardless of type being pointed to). This is not required by the C standard, which only requires pointers to `void` be capable of storing all other non-function pointer values, and that all function pointer values be inter-convertible. Again, our decision to specialise on particular, and reasonable, target architectures makes life simpler.

Pointers retain type information by using "phantom" type variables. The Isabelle declaration is

```
datatype 'a ptr = Ptr addr
```

Then, if the C program under analysis calls for a variable of type `char *`, the Isabelle environment will include a variable of Isabelle type `byte ptr`. In this way our pointers are typed, even if their underlying encoding makes it trivial to view a pointer to one type as a pointer to another type.

Because the underlying representation is always the same, all pointer types are proved to be in the class `mem_type` once and for all (in theory `CTypes`). Pointers to `void` are represented as values in the Isabelle type `unit Ptr`, where `unit` is the standard singleton type. The `unit` type is not shown to be in the class `mem_type`.

28.4.3 Arrays

C arrays are lists of values of fixed length. A faithful representation of this type requires a novel Isabelle type. We build on Anthony Fox's implementation of John Harrison's "finite Cartesian products" idea [4]. Syntactic trickery within Isabelle allows us to write types like

```
nat[10]
```

which is an array of 10 natural numbers. There are operators for updating and indexing into arrays. Note that the type `10` that appears above is an Isabelle *type*, not a term.

If type τ is a `mem_type`, then an array of τ values will also be a `mem_type`, as long as the size of the array is not so large that the array would not fit into memory. This condition is discharged as the StrictC program is translated.

For technical reasons due to the implementation of type classes in Isabelle, we need to fix separate limits, ahead of time, on the number of elements in an array and the size of each element. Currently, for 32-bit ARM, our model fixes a maximum of:

```
struct listnode {
   int node_data;
   struct listnode *next;
};
```

```
struct node2;
struct node1 {
   struct node2 *data;
   struct node1 *next, *prev;
   int someflag;
};
struct node2 {
   struct node1 *owner;
   char stringdata[100];
};
```

Figure 28.2: Examples of Recursive `struct` Declarations Accepted in StrictC

- 2^{13} (8192) elements in each array; and
- 2^{19} bytes (512 KiB) in each array element

For x86-64, the limits are:

- 2^{20} (1048576) elements in each array; and
- 2^{26} bytes (64 MiB) in each array element

One would prefer to be able to multiply the size of the element type by the number of the elements, but the type system does not permit this (for good technical reasons).

28.4.4 C `struct` Types

From the point of view of the translation into Isabelle, `struct` types do not pose any great conceptual difficulties. A `struct` type is clearly very similar to an Isabelle record, which in turn is conceptually the same as a tuple. The first complication that arises is that Isabelle tuples can not be recursive, whereas C `struct` types are often recursive (as when implementing linked structures in the heap).

This required the implementation of an alternative record definition package, allowing (possibly multiple) recursive types. This then allows Isabelle types to be defined that correspond to C declarations such as those shown in Figure 28.2.

Confirming that a `struct` type really is representable in memory, requires the definition of functions for converting Isabelle records into lists of bytes and *vice versa*. The size of the converted value must also be checked to be no bigger than the size of memory. Both of these actions require knowledge of how the fields of the `struct` are laid out in memory, which is in turn a function of the padding that can be inserted between the fields. Such calculations are architecture dependent.

28.5 Translating Expressions

Expression translation is implemented in `expression_translation.ML`.

Fundamental Concepts StrictC expressions are essentially a subset of C expressions, and are fairly easy to translate to corresponding Isabelle expressions that manipulate Isabelle-encoded values. There are two fundamental concepts to grasp of expression translation. First, when being evaluated ("read to determine a value") a C expression of type τ becomes an Isabelle expression of type `statespace` $\rightarrow [\![\tau]\!]$, where $[\![\tau]\!]$ is the translated, Isabelle version of C type τ.

This must be done to make sense of expressions that read memory: an expression such as `s.arrayfld[3]` only has a specific value in the context of a specific state of memory. This use of a "lifted" function space to represent the expression is a standard technique in denotational semantics. In the example given, the value of the expression is a function that looks at the statespace to determine what data is stored at variable `s`. As the statespace evolves, the value returned from this function changes, but the function's value is the same.

Expressions do not just determine values however, they can also denote an *l-value*, something denoting a "place in memory" that is to be updated. This is done when an address is taken, or when an assignment is to be performed. In a simple language, l-values might only be variable names, but C allows for complicated expressions on both sides of an assignment. The example above (`s.arrayfld[3]`) might just as well be assigned to as read. So, the l-value of an expression e that has type τ will be an Isabelle value of type `statespace` $\rightarrow \tau \rightarrow$ `statespace`, a function that takes a new value and a statespace to change, and returns the updated statespace.

Not all expressions are l-values (the expression `3` is not, for example), so the translation of any one expression can return one or two different values, an "r-value" as well as an optional l-value.

In addition, because of the way the translation does not put local variables into memory, the translation provides another separate optional value, that of the expression's address. If everything did live in memory, all l-values would have addresses, and one could dispense with the separate calculation of l-values. Thus the first three lines of Figure 28.3.

The SML types given to the `lval`, `addr` and `rval` fields in the declaration of `expr_info` are themselves function spaces at the SML level. These function spaces manipulate values of SML type `term`. The way in which typing at the C level is reflected at the Isabelle level is mainly hidden at the SML level, where the programmer just manipulates terms (the Isabelle types are internal to those values).

However, the function spaces *can* be made visible at the SML level. This is done mainly to reduce the number of β-redexes that would otherwise be

```
datatype expr_info =
  EI of {lval   : (term -> term -> term) option,
         addr   : (term -> term) option,
         rval   : term -> term,
         cty    : int ctype,
         ibool  : bool,
         lguard : (term -> term * term) list,
         guard  : (term -> term * term) list,
         left   : SourcePos.t,
         right  : SourcePos.t }
```

Figure 28.3: The `expr_info` type, into which C expressions are translated internally.

created in the resulting term. For example, translating the C expression `x + 3` will first create r-values

$$\begin{aligned} [\![\mathtt{x}]\!] &= (\lambda\sigma.\ \mathtt{x}(\sigma)) \\ [\![\mathtt{3}]\!] &= (\lambda\sigma.\ 3) \end{aligned}$$

where the application of the `x` function to a statespace σ pulls out the value of variable `x` in σ.

When translating an expression $e_1 + e_2$, one naturally creates the value

$$\lambda\sigma.\ [\![e_1]\!](\sigma) + [\![e_2]\!](\sigma)$$

If this was done within Isabelle `term` values, the result would be a `term` full of expressions of the form $(\lambda v.\ M)N$. By lifting the Isabelle λ to the SML level, only one abstraction need to be created, at the very top-level in the translation. Thus, the translation of the addition becomes the SML expression

```
(fn s => mk_plus (rval_of e1 s) (rval_of e2 s))
```

where `rval_of` returns the `rval` field of an `expr_info` and the β-reduction happens within SML.

The fourth line of the record in `expr_info` values stores the type of the expression, something that informs the translation of many different C expressions. For example, additions are not as simple as just presented because of the possibility that one of the arguments might be a pointer. The `left` and `right` fields of the `expr_info` record store the source-code position of the original expression. The two guard fields are explained below in Section 28.5.1.

The `ibool` field of the `expr_info` records whether or not the term being generated in the r-value is of Isabelle's boolean type. This is done so that

translation can avoid some conversions between Isabelle words and booleans. For example, if the expression being translated is `x < 6 && y > 10`, the resulting Isabelle term will include a use of the two comparison operators on words. Strictly, one should then turn the boolean results of these operators into either a one or zero. But, as these results are then combined with boolean conjunction, the values will immediately be converted back into booleans.

Of course, in an expression such as `x && x->fld > 6`, the first operand to the conjunction does not have Isabelle boolean type, and will be converted to a boolean value by comparing it with the null pointer. (In fact, this particular example causes a more complicated translation effect to occur; see Section 28.5.1 below on undefined behaviour.) Dually, if the code puts a boolean value into memory, the translation has to make sure that the Isabelle term has the appropriate word type once more. There are two functions used here:

```
mk_isabool : expr_info -> expr_info
strip_kb   : expr_info -> expr_info
```

The function `mk_isabool` produces an `expr_info` value that does have Isabelle boolean type (it will be the identity function on an r-value that is already known to be boolean). The function `strip_kb` reverses this, turning an Isabelle boolean into an Isabelle word if necessary.

28.5.1 Undefined Behaviour

There are three classes of under-specification in the C standard. Those classed as *implementation defined* are behaviours or values that are supposed to be fixed and documented by particular implementations. For example, the number of bits in an `int` (a number that must be at least 16), is a fixed value that implementations are allowed to choose themselves. Because StrictC targets a particular architecture, most implementation-defined aspects of C can be "baked into" the translation. (The design attempts to have the translation sources be easy to modify to account for different architectures; see for example the `ImplementationNumbers` structure in each of the `TargetNumbers.ML` files.)

The second class of under-specification comprises *unspecified* behaviours. The most important of these is the order of evaluation of arguments to operators and function calls. Therefore, one should avoid writing code that depends on unspecified evaluation order. Otherwise, the compiled executable might have different semantics compared to the Simpl specification that we generate. For example:

```
int x = f() + g();
```

where `f` and `g` both have side effects, is currently allowed but should be avoided. In the future, we plan to forbid or restrict programs that may have unspecified evaluation order. The standalone analysis tool can check for this problem using the `-embedded_fncalls` option.

The third class of under-specification is *undefined behaviour*. In essence, undefined behaviours are runtime errors (such as dereferencing a null pointer). They are undefined because the standard does not require the implementation to catch them, or to realise that they have occurred. Rather, if an undefined behaviour occurs, the user can no longer rely on their program to do anything sensible. In effect, implementations are given licence to blunder on however they like when an undefined behaviour occurs.

Thus, it is critical that the C code we verify never exhibits any undefined behaviours. We do this with a feature of the Hoare environment called the *guard*. A guard is a boolean condition g attached to a statement s, with the combination written $g \to s$. If the guard g is true when the combined statement is to be executed, then s is allowed to execute. If it is false, the underlying semantics defines the result to be a "fault". When a program faults, nothing more can happen, so it has effectively aborted. Verifying a program with guarded statements thus requires proofs that the guards are always true.

Most guards arise in expressions, and the translation process accumulates these in one top-level guard at the statement level. For example, in the statement

```
x = *p >> i;
```

there will be four guards attached to the statement that will need to be discharged: that `p` is not null, that `*p` is not negative, that `i` is not negative, and that `i` is less than 32.

One elegant feature of guards in the Hoare environment is that they can be selectively disabled for the purposes of verification. For example, the C standard's requirement that right shift operations might not be performed on negative numbers (that it results in undefined behaviour) is too strict, given a particular C compiler and target architecture. In this situation, it is possible to prove Hoare-triples where that particular guard is not used, so that the semantics of $g \to s$ is simply that of s.

28.6 Concrete Syntax: Parsing and Lexing

The grammar for StrictC is given in the file `StrictC.grm`, which is given as input to the standard tool `mlyacc`. The format of an `mlyacc` file looks quite similar to the format accepted by `yacc`. A typical grammar rule is

```
init_declarator_list
: init_declarator
```

```
                ([init_declarator])
| init_declarator YCOMMA init_declarator_list
                (init_declarator :: init_declarator_list)
```

where a non-terminal appears before a colon, and multiple possible right-hand sides are separated by the pipe or vertical bar character. The code for a production appears in parentheses after each right-hand-side. The code's convention is to have token names (*e.g.*, `YCOMMA` above) be upper-case.

Apart from the changes that turn assignment expressions into statements, the grammar for StrictC attempts to be as close as possible to the grammar of the C standard. In general, the names for non-terminals in `StrictC.grm` are taken from the standard, so it should be fairly clear how the standard's grammar has been mapped into `StrictC.grm`.

28.6.1 Lexing and `typedef` Names

The standard problem in lexing and parsing C is that the grammar is ambiguous: the non-terminal *typedef-name* is defined to simply be the same as an *identifier*. When the parser encounters

```
x * f(y);
```

it can't know if this is meant to be a multiplication of variable `x` by a function call expression, or whether it is the (prototype) declaration of a function called `f` taking an argument of type `y` and returning a value of type `x`.

To resolve this, the lexer must be able to classify identifier tokens as normal identifiers or *typedef-names*. The StrictC translator's approach to this problem is not typical, because of the strange way in which `mlyacc` combines handling of side effects with its error correction. Normally, one would have some sort of updatable symbol table that the parser would write to when it encountered a `typedef` declaration. The lexer would read from the same table as it encountered identifiers, allowing appropriate categorisation.

The approach taken in the StrictC translator is to have the lexer do all of the work, without reference to the parser (see `StrictC.lex`). This *is* possible, but is also convoluted, and involves many updatable variables that are internal to the lexer. The basic idea is that when the lexer sees a `typedef` token, it switches to the `TDEF` state. When an identifier is seen in this state, the identifier is added to the list of *typedef-names*, and lexing can continue. The complications arrive in declarations like

```
typedef struct s { int fld1; char fld2; } s_t;
```

where the identifiers `s`, `fld1` and `fld2` have to be ignored, and `s_t` taken as the new *typedef-name*. This requires the lexer to handle the matching brace characters, and partly motivates the requirement that `typedef` declarations all occur at the top-level (and not be nested).

28.6.2 GCC `__attribute__` Declarations

The parser handles, but mostly ignores, various `gcc`-specific extensions, such as `__attribute__`. These are tricky to parse (something admitted by the relevant `gcc` documentation): users are given almost unlimited liberty to put their `__attribute__` strings anywhere within a declaration.

For example, these three

```
int f(int) __attribute__((__const__));
__attribute__((__const__)) int f(int);
int __attribute__((__const__)) f(int);
```

are all supposed to parse successfully (and have the same meaning). Making this work is rather involved. Most attributes are ignored, but the standalone analysis tool does check that `const` and `pure` attributes are reasonable, given what it knows of how functions may modify and read the global state.

Bibliography

[1] D. Cock. Bitfields and tagged unions in C: Verification through automatic generation. In B. Beckert and G. Klein, editors, *Proc, 5th VERIFY*, volume 372 of *CEUR Workshop Proceedings*, pages 44–55, Sydney, Australia, Aug. 2008.

[2] D. Greenaway, J. Andronick, and G. Klein. Bridging the gap: Automatic verified abstraction of C. 7406:99–115, Aug. 2012.

[3] D. Greenaway, J. Lim, J. Andronick, and G. Klein. Don't sweat the small stuff: Formal verification of C code without the pain. pages 429–439, June 2014.

[4] J. Harrison. A HOL theory of Euclidean space. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference*, volume 3603 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2005.

[5] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. http://www.in.tum.de/~nipkow/LNCS2283/.

[6] M. Norrish. *C Formalised in HOL*. PhD thesis, Computer Laboratory, University of Cambridge, 1998. Also published as Technical Report 453, available from http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-453.pdf.

[7] M. Norrish. C-to-Isabelle parser, version 1.13.0, May 2013. Accessed May 2016.

[8] N. Schirmer. A sequential imperative programming language syntax, semantics, hoare logics and verification environment. *Archive of Formal Proofs*, Feb. 2008. Formal proof development.

[9] H. Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, Aug 2008.