The Imperative Refinement Framework

Peter Lammich

August 16, 2018
Abstract

We present the Imperative Refinement Framework (IRF), a tool that supports a stepwise refinement based approach to imperative programs. This entry is based on the material we presented in [ITP-2015, CPP-2016].

It uses the Monadic Refinement Framework as a frontend for the specification of the abstract programs, and Imperative/HOL as a backend to generate executable imperative programs.

The IRF comes with tool support to synthesize imperative programs from more abstract, functional ones, using efficient imperative implementations for the abstract data structures.

This entry also includes the Imperative Isabelle Collection Framework (IICF), which provides a library of re-usable imperative collection data structures.

Moreover, this entry contains a quickstart guide and a reference manual, which provide an introduction to using the IRF for Isabelle/HOL experts. It also provides a collection of (partly commented) practical examples, some highlights being Dijkstra’s Algorithm, Nested-DFS, and a generic worklist algorithm with subsumption.

Finally, this entry contains benchmark scripts that compare the runtime of some examples against reference implementations of the algorithms in Java and C++.

[CPP-2016] Peter Lammich: Refinement based verification of imperative data structures. CPP 2016: 27–36
## Contents

1 The Sepref Tool ........................................... 6
  1.1 Operation Identification Phase .................. 6
    1.1.1 Proper Protection of Term ....................... 6
    1.1.2 Operation Identification ....................... 7
    1.1.3 ML-Level code .................................. 8
    1.1.4 Default Setup .................................. 13
  1.2 Basic Definitions ................................... 13
    1.2.1 Values on Heap ................................ 13
    1.2.2 Constraints in Refinement Relations ............ 16
    1.2.3 Heap-Nres Refinement Calculus ................ 16
    1.2.4 Product Types .................................. 19
    1.2.5 Convenience Lemmas .............................. 19
    1.2.6 ML-Level Utilities ............................... 26
  1.3 Monadify .............................................. 37
  1.4 Frame Inference ...................................... 52
  1.5 Refinement Rule Management ....................... 63
    1.5.1 Assertion Interface Binding ..................... 63
    1.5.2 Function Refinement with Precondition ........... 64
    1.5.3 Heap-Function Refinement ....................... 66
    1.5.4 Automation ...................................... 77
  1.6 Setup for Combinators ............................... 100
    1.6.1 Interface Types .................................. 101
    1.6.2 Rewriting Inferred Interface Types ............... 103
    1.6.3 ML-Code ......................................... 104
    1.6.4 Obsolete Manual Setup Rules ..................... 112
  1.7 Translation .......................................... 113
    1.7.1 Import of Parametricity Theorems ............... 128
    1.7.2 Purity ......................................... 131
  1.8 Sepref-Definition Command ......................... 132
    1.8.1 Setup of Extraction-Tools ....................... 132
    1.8.2 Synthesis setup for sepref-definition goals ...... 132
  1.9 Utilities for Interface Specifications and Implementations 136
    1.9.1 Relation Interface Binding ..................... 136
3.4 Priority Bag Interface ........................................ 237
  3.4.1 Operations ........................................... 237
  3.4.2 Patterns ............................................. 240
3.5 Multisets by Lists ........................................... 240
  3.5.1 Abstract Operations ................................... 240
  3.5.2 Declaration of Implementations ......................... 241
  3.5.3 Swap two elements of a list, by index .................. 244
  3.5.4 Operations ........................................... 245
  3.5.5 Patterns ............................................. 246
3.6 Heap Implementation On Lists ................................ 247
  3.6.1 Basic Definitions .................................... 248
  3.6.2 Basic Operations .................................... 251
  3.6.3 Auxiliary operations .................................. 254
  3.6.4 Operations ........................................... 262
  3.6.5 Operations as Relator-Style Refinement ................. 265
3.7 Implementation of Heaps with Arrays ......................... 275
  3.7.1 Setup of the Sepref-Tool ................................ 275
  3.7.2 Synthesis of operations ................................ 277
  3.7.3 Regression Test ....................................... 278
3.8 Map Interface ............................................... 279
  3.8.1 Parametricity for Maps ................................ 279
  3.8.2 Interface Type ........................................ 280
  3.8.3 Operations ........................................... 281
  3.8.4 Patterns ............................................. 282
  3.8.5 Parametricity ......................................... 282
3.9 Priority Maps ................................................ 283
  3.9.1 Additional Operations .................................. 284
3.10 Priority Maps implemented with List and Map ................. 286
  3.10.1 Basic Setup ........................................... 286
  3.10.2 Basic Operations .................................... 287
  3.10.3 Auxiliary Operations .................................. 293
  3.10.4 Operations ........................................... 297
3.11 Plain Arrays Implementing List Interface ..................... 312
  3.11.1 Empty ................................................ 320
  3.11.2 Swap ................................................ 321
  3.11.3 Length ............................................... 322
  3.11.4 Index ................................................ 322
  3.11.5 Butlast ............................................... 323
  3.11.6 Append ............................................... 324
  3.11.7 Get .................................................. 326
  3.11.8 Contains ............................................. 326
3.12 Implementation of Heaps by Arrays .......................... 327
  3.12.1 Implement Basic Operations ............................ 333
  3.12.2 Auxiliary Operations .................................. 335
4 User Guides

4.1 Quickstart Guide

4.1.1 Introduction

4.1.2 First Example

4.1.3 Binary Search Example

4.1.4 Basic Troubleshooting

4.1.5 The Isabelle Imperative Collection Framework (IICF)

4.1.6 Specification of Preconditions

4.1.7 Linearity and Copying

4.1.8 Nesting of Data Structures

4.1.9 Fixed-Size Data Structures

4.1.10 Type Classes

4.1.11 Higher-Order

4.1.12 A-Posteriori Optimizations

4.1.13 Short-Circuit Evaluation

4.2 Reference Guide

4.2.1 The Sepref Method

4.2.2 Refinement Rules

4.2.3 Composition

4.2.4 Registration of Interface Types

4.2.5 Registration of Abstract Operations

4.2.6 High-Level tools for Interface/Implementation Declaration

4.2.7 Defining synthesized Constants
4.3 General Purpose Utilities ............................................ 438
  4.3.1 Methods ...................................................... 438
  4.3.2 Structured Apply Scripts (experimental) ................. 439
  4.3.3 Extracting Definitions from Theorems ..................... 440

5 Examples ........................................................................ 441
  5.1 Imperative Graph Representation ................................. 441
  5.2 Simple DFS Algorithm ............................................. 443
    5.2.1 Definition .................................................. 443
    5.2.2 Refinement to Imperative/HOL ............................. 447
  5.3 Imperative Implementation of Dijkstra’s Shortest Paths Al-
      gorithm .................................................................. 448
  5.4 Imperative Implementation of Nested DFS (HPY-Improvement) ........................................... 455
  5.5 Generic Worklist Algorithm with Subsumption ................ 458
    5.5.1 Utilities ....................................................... 458
    5.5.2 Search Spaces ................................................ 459
    5.5.3 Worklist Algorithm ......................................... 460
    5.5.4 Towards an Implementation ................................. 465
  5.6 Non-Recursive Algebraic Datatype ................................. 468
    5.6.1 Refinement Assertion ....................................... 468
    5.6.2 Constructors ................................................ 469
    5.6.3 Destructor ..................................................... 470
    5.6.4 Regression Test ............................................. 473
  5.7 Snippet to Define Custom Combinators ......................... 474
    5.7.1 Definition of the Combinator ............................... 474
    5.7.2 Synthesis of Implementation ............................... 475
    5.7.3 Setup for Sepref ........................................... 476
    5.7.4 Example ....................................................... 476
    5.7.5 Limitations ................................................... 477

6 Benchmarks ................................................................. 478
Chapter 1

The Sepref Tool

This chapter contains the Sepref tool and related tools.

1.1 Operation Identification Phase

theory Sepref-Id-Op
imports
  Main
  Automatic-Refinement.Refine-Lib
  Automatic-Refinement.Autoref-Tagging
  Lib/Named-Theorems-Rev
begin

The operation identification phase is adapted from the Autoref tool. The basic idea is to have a type system, which works on so called interface types (also called conceptual types). Each conceptual type denotes an abstract data type, e.g., set, map, priority queue.

Each abstract operation, which must be a constant applied to its arguments, is assigned a conceptual type. Additionally, there is a set of pattern rewrite rules, which are applied to subterms before type inference takes place, and which may be backtracked over. This way, encodings of abstract operations in Isabelle/HOL, like \( \lambda \)-. \( \text{None} \) for the empty map, or \( \text{fun-upd} \ m \ k \ (\text{Some} \ v) \) for map update, can be rewritten to abstract operations, and get properly typed.

1.1.1 Proper Protection of Term

The following constants are meant to encode abstraction and application as proper HOL-constants, and thus avoid strange effects with HOL’s higher-order unification heuristics and automatic beta and eta-contraction.

The first step of operation identification is to protect the term by replacing all function applications and abstractions be the constants defined below.
definition \([\text{simp}]: \text{PROTECT}2 \ x \ (y::\text{prop}) \equiv x\)

consts DUMMY :: \text{prop}

abbreviation \text{PROTECT}2\text{-syn} \ ((\#-\#)) \ where \ \text{PROTECT}2\text{-syn} \ t \equiv \text{PROTECT}2 \ t \ \text{DUMMY}

abbreviation \((\text{input})\) \text{ABS}2 :: (\'a\Rightarrow \'b) \Rightarrow \'a\Rightarrow \'b \ (\text{binder} \ \lambda_2 \ 10)

where \text{ABS}2 \ f \equiv (\lambda x. \text{PROTECT}2 \ (f \ x) \ \text{DUMMY})

lemma beta: \((\lambda_2 x. f x)\$x \equiv f x\) by simp

Another version of \((\$)\). Treated like \((\$)\) by our tool. Required to avoid infinite pattern rewriting in some cases, e.g., map-lookup.

definition \text{APP}' \ (\text{infix} \$'' \ 900) \ where \ [\text{simp, autoref-tag-defs}]: \ f'\ a \equiv f \ a

Sometimes, whole terms should be protected from being processed by our tool. For example, our tool should not look into numerals. For this reason, the \text{PR-CONST} \ tag indicates terms that our tool shall handle as atomic constants, an never look into them.

The special form \text{UNPROTECT} \ can be used inside pattern rewrite rules. It has the effect to revert the protection from its argument, and then wrap it into a \text{PR-CONST}.

definition \ [\text{simp, autoref-tag-defs}]: \text{PR-CONST} \ x \equiv x — \text{Tag to protect constant}
definition \ [\text{simp, autoref-tag-defs}]: \text{UNPROTECT} \ x \equiv x — \text{Gets converted to PR-CONST, after unprotecting its content}

1.1.2 Operation Identification

Indicator predicate for conceptual typing of a constant

definition intf-type :: \text{'}a \Rightarrow \text{'}b \text{ itself} \Rightarrow \text{bool} \ (\text{infix} ::; \ \text{10}) \ where

\[ [\text{simp}]: c;::I \equiv \text{True} \]

lemma itypeI: c;::I by simp
lemma itypeI': intf-type \ c \ \text{TYPE('T)} \ by \ (\text{rule itypeI})
lemma itype-self: \ (c::'a) ::; \text{TYPE('a)} \ by \ simp

definition CTYPE-ANNOT :: \text{'}b \Rightarrow \text{'}a \text{ itself} \Rightarrow \text{'}b \ (\text{infix} ::; \ \text{10}) \ where

\[ [\text{simp}]: c::;I \equiv c \]

Wrapper predicate for an conceptual type inference
definition ID :: \text{'}a \Rightarrow \text{'}a \Rightarrow \text{'}c \text{ itself} \Rightarrow \text{bool} \n
\where \ [\text{simp}]: \ ID \ t \ t' \ \text{T} \equiv \ t=t'

Conceptual Typing Rules

lemma ID-unfold-vars: \text{ID} \ x \ y \ \text{T} \Longrightarrow \ x\equiv y \ by \ simp
lemma ID-PR-CONST-trigger: ID (PR-CONST x) y T \implies ID (PR-CONST x) y T.

lemma pat-rule:
\[
[p \equiv p'; ID p' t' T] \implies ID p t' T \text{ by simp}
\]

lemma app-rule:
\[
[ID f f' \text{ TYPE('a\Rightarrow'b)}; ID x x' \text{ TYPE('a)}] \implies ID (f\, x') (f'\, x') \text{ TYPE('b)}
\text{ by simp}
\]

lemma app′-rule:
\[
[ID f f' \text{ TYPE('a\Rightarrow'b)}; ID x x' \text{ TYPE('a)}] \implies ID (f\, x') (f'\, x') \text{ TYPE('b)}
\text{ by simp}
\]

lemma abs-rule:
\[
\exists x. ID x x' \text{ TYPE('a)} \implies ID (t\, x) (t'\, x') \text{ TYPE('b)} \implies ID (\lambda x. t) (\lambda x'. t'\, x') \text{ TYPE('a\Rightarrow'b)}
\text{ by simp}
\]

lemma id-rule: c::I \implies ID c c I \text{ by simp}

lemma annot-rule: ID t t' I \implies ID (t::I) t' I
\text{ by simp}

lemma fallback-rule:
ID (c::'a) c \text{ TYPE('c)}
\text{ by simp}

lemma unprotect-rl1: ID (PR-CONST x) t T \implies ID (UNPROTECT x) t T
\text{ by simp}

1.1.3 ML-Level code

ML ⟨⟨
infixed 0 THEN-ELSE-COMB'

signature ID-OP-TACTICAL = sig
val SOLVE-FWD: tactic' \Rightarrow tactic'
val DF-SOLVE-FWD: bool \Rightarrow tactic' \Rightarrow tactic'

end

structure Id-Op-Tactical :ID-OP-TACTICAL = struct

fun SOLVE-FWD tac i st = SOLVED' (tac
THEN-ALL-NEW-FWD (SOLVE-FWD tac)) i st

(* Search for solution with DFS-strategy. If dbg-flag is given,
return sequence of stuck states if no solution is found.

\texttt{fun DF-SOLVE-FWD
dbg \textup{tac} = let
val stuck-list-ref = Unsynchronized.ref []

\texttt{fun stuck-tac \textup{- st} = if dbg then (
  stuck-list-ref := st :: !stuck-list-ref;
  Seq.empty
) else Seq.empty}

\texttt{fun rec-tac \textup{i st} =
  (tac THEN-ALL-NEW-FWD (SOLVED' rec-tac))
  ORELSE' stuck-tac
) \textup{i st}

\texttt{fun fail-tac \textup{- -} = if dbg then
  Seq.of-list (rev (!stuck-list-ref))
else Seq.empty
in
  rec-tac ORELSE' fail-tac
end}

end
\texttt{⟩⟩}

\texttt{named-theorems-rev \textup{id-rules} Operation identification rules}
\texttt{named-theorems-rev \textup{pat-rules} Operation pattern rules}
\texttt{named-theorems-rev \textup{def-pat-rules} Definite operation pattern rules (not backtracked over)}

\texttt{ML ⟨⟨

\texttt{structure Id-Op = struct}

\texttt{fun id-a-conv \textup{env} \textup{ct} = case Thm.term-of \textup{ct} of
  @\{mpat ID - - \} => Conv.fun-conv (Conv.fun-conv (Conv.arg-conv \textup{env})) \textup{ct}
  - => raise CTERM(id-a-conv,[\textup{ct}])}

\texttt{fun
  protect \textup{env} (@\{mpat ?t::?I\}) = let
  val \textup{t} = protect \textup{env} \textup{t}
  in
  @\{mk-term \textup{env}: \textup{t::?I}\}
  end
| protect - (\textup{t as @\{mpat PR-CONST -\}}) = \textup{t}
| protect \textup{env} (t1$t2) = let

\texttt{⟩⟩

9
val t1 = protect env t1
val t2 = protect env t2

in
\{mk-term env: ?t1.0 $ ?t2.0\}
end

| protect env (Abs (x,T,t)) = let
val t = protect (T::env) t
in
\{mk-term env: λv-x::?v-T. PROTECT2 ?t DUMMY\}
end

| protect - t = t

fun protect-conv ctxt = Refine-Util.f-tac-conv ctxt
(simp-tac
(put-simpset HOL-basic-ss ctxt addsimps \{ths PROTECT2-def APP-def\})
1)

fun unprotect-conv ctxt
= Simplifier.rewrite (put-simpset HOL-basic-ss ctxt
addsimps \{ths PROTECT2-def APP-def\})

fun do-unprotect-tac ctxt =
  resolve-tac ctxt @\{ths unprotect-rl1\} THEN’
  CONVERSION (Refine-Util.HOL-concl-conv (fn ctxt => id-a-conv (unprotect-conv ctxt)) ctxt)

val cfg-id-debug =
  Attrib.setup-config-bool \{binding id-debug\} (K false)

val cfg-id-trace-fallback =
  Attrib.setup-config-bool \{binding id-trace-fallback\} (K false)

fun dest-id-rl thm = case Thm.concl-of thm of
  \{mpat (typs) Trueprop (?c::TYPE(?v-T))\} => (c,T)
| - => raise THM(dest-id-rl,-1,[thm])

val add-id-rule = snd oo Thm.proof-attributes [Named-Theorems-Rev.add \{named-theorems-rev id-rules\}]

datatype id-tac-mode = Init | Step | Normal | Solve

fun id-tac ss ctxt = let
  open Id-Op-Tactical
  val certT = Thm.ctyp-of ctxt
  val cert = Thm. cterm-of ctxt
val thy = Proof-Context. theory-of ctxt
val id-rules = Named-Theorems-Rev.get ctxt @{named-theorems-rev id-rules}
val pat-rules = Named-Theorems-Rev.get ctxt @{named-theorems-rev pat-rules}
val def-pat-rules = Named-Theorems-Rev.get ctxt @{named-theorems-rev def-pat-rules}

val rl-net = Tactic.build-net {
  (pat-rules |> map (fn thm => thm RS @{thm pat-rule}))
@ @ {thms annot-rule app-rule app'-rule abs-rule}
@ (id-rules |> map (fn thm => thm RS @{thm id-rule}))
}

val def-rl-net = Tactic.build-net {
  (def-pat-rules |> map (fn thm => thm RS @{thm pat-rule}))
}

val id-pr-const-rename-tac = resolve-tac ctxt @{thms ID-PR-CONST-trigger} THEN'
  Subgoal.FOCUS (fn { context=ctxt, prems, ... } =>
    let
      fun is-ID @{mpat Trueprop (ID - - -)} = true | is-ID - = false
      val prems = filter (Thm.prop_of #> is-ID) prems
      val eqs = map (fn thm => thm RS @{thm ID-unfold-vars}) prems
      val conv = Conv.rewrs-conv eqs
      val conv = fn ctxt => (Conv.top-sweep-conv (K conv) ctxt)
      val conv = fn ctxt => Conv.fun2-conv (Conv.arg-conv (conv ctxt))
      val conv = Refine-Util.HOL-concl-conv conv ctxt
    in CONVERSION conv 1 end
  ) ctxt THEN'
  resolve-tac ctxt @{thms id-rule} THEN'
  resolve-tac ctxt id-rules

val ityping = id-rules
|> map dest-id-rl
|> filter (is-Const o #1)
|> map (apfst (#1 o dest-Const))
|> Symtab.make-list

val has-type = Symtab.defined ityping

fun mk-fallback name cT =
  case try (Sign.the-const-constraint thy) name of
    SOME T =>>
    try (Thm.instantiate' [SOME (certT cT), SOME (certT T)] [SOME (cert (Const (name, cT)))])
@{thm fallback-rule}
| NONE => NONE

fun trace-fallback thm =
```ocaml
Config.get ctxt cfg-id-trace-fallback
andalso let
  open Pretty
val p = block [str ID-OP: Applying fallback rule: , Thm.pretty-thm ctxt thm]
in
  string-of p |> tracing;
false
end

val fallback-tac = CONVERSION Thm.eta-conversion THEN' IF-EXGOAL
(fn i => fn st =>
  case Logic.concl-of-goal (Thm.prop-of st) i of
    @\{mpat Trueprop (ID (mpaq-STRUCT (mpaq-Const ?name ?cT)) - -)}
=> (if not (has-type name) then
    case mk-fallback name cT of
      SOME thm => (trace-fallback thm; resolve-tac ctxt [thm] i st)
        | NONE => Seq.empty
    else Seq.empty
  )
| - => Seq.empty)

val init-tac = CONVERSION (Refine-Util.HOL-concl-conv (fn ctxt => (id-a-conv (protect-conv ctxt))))
ctxt

val step-tac = (FIRST' [assume-tac ctxt,
eresolve-tac ctxt @\{thms id-rule\},
resolve-from-net-tac ctxt def-rl-net,
resolve-from-net-tac ctxt rl-net,
id-pr-const-rename-tac,
do-unprotect-tac ctxt,
fallback-tac])

val solve-tac = DF-SOLVE-FWD (Config.get ctxt cfg-id-debug) step-tac
in
  case ss of
    Init => init-tac
| Step => step-tac
| Normal => init-tac THEN' solve-tac
| Solve => solve-tac
end
end
```
1.1.4 Default Setup

Numerals

lemma pat-numeral[def-pat-rules]: numeral$x$ ≡ UNPROTECT (numeral$x$) by simp

lemma id-nat-const[id-rules]: (PR-CONST (a::nat)) :: TYPE(nat) by simp
lemma id-int-const[id-rules]: (PR-CONST (a::int)) :: TYPE(int) by simp

end

1.2 Basic Definitions

theory Sepref-Basic
imports HOL-Eisbach.Eisbach
Separation-Logic-Imperative-HOL.Sep-Main
Refine-Monadic.Refine-Monadic
Lib/Sepref-Misc
Lib/Structured-Apply
Sepref-Id-Op
begin
no-notation i-ANNOT (infixr :: 10)
no-notation CONST-INTF (infixr :: 10)

In this theory, we define the basic concept of refinement from a nonde-
terministic program specified in the Isabelle Refinement Framework to an
imperative deterministic one specified in Imperative/HOL.

1.2.1 Values on Heap

We tag every refinement assertion with the tag $hn-ctxt$, to avoid higher-order
unification problems when the refinement assertion is schematic.

definition $hn-ctxt$ :: ('a::'c⇒assn) ⇒ 'a ⇒ 'c ⇒ assn
— Tag for refinement assertion

where

$hn-ctxt$ $P$ $a$ $c$ ≡ $P$ $a$ $c$

definition pure :: ('b × 'a) set ⇒ 'a ⇒ 'b ⇒ assn
— Pure binding, not involving the heap

where pure $R$ ≡ ($\lambda$ $a$. $\uparrow$(($c$, $a$)$\in$R))
lemma pure-app-eq: pure R a c = ↑((c,a)∈R) by (auto simp: pure-def)

lemma pure-eq-conv[simp]: pure R = pure R' ↔ R=R'
  unfolding pure-def
  apply (rule iffI)
  apply safe
  apply (meson pure-assn-eq-conv)
  apply (meson pure-assn-eq-conv)
  done

lemma pure-rel-eq-false-iff: pure R x y = false ↔ (y,x)∉R
  by (auto simp: pure-def)

definition is-pure P ≡ ∃P'. ∀x x'. P x x' = ↑(P' x x')

lemma is-pureI[intro?]: assumes ∀x x'. P x x' = ↑(P' x x')
  shows is-pure P
  using assms unfolding is-pure-def by blast

lemma is-pureE: assumes is-pure P
do
  obtains P' where ∀x x'. P x x' = ↑(P' x x')
  using assms unfolding is-pure-def by blast

lemma pure-pure[simp]: is-pure (pure P)
  unfolding pure-def by rule blast

lemma pure-hn-ctxt[intro!]: is-pure P =⇒ is-pure (hn-ctxt P)
  unfolding hn-ctxt-def[abs-def].

definition the-pure P ≡ THE P'. ∀x x'. P x x' = ↑((x',x)∈P')

lemma the-pure-pure[simp]: the-pure (pure R) = R
  unfolding pure-def the-pure-def
  by (rule theI2[where a=R]) auto

lemma is-pure-alt-def: is-pure R ↔ (∃R'. ∀y. R x y = ↑((y,x)∈R'))
  unfolding is-pure-def
  apply auto
  apply (rename_tac P')
  apply (rule-tac x = {(x,y). P' y x} in exI)
  apply auto
  done

lemma pure-the-pure[simp]: is-pure R =⇒ pure (the-pure R) = R
  unfolding is-pure-alt-def pure-def the-pure-def
  apply (intro ext)
  apply clarsimp
apply (rename-tac a c Ri)
apply (rule-tac a=Ri in theI2)
apply auto
done

lemma is-pure-conv: is-pure R ⟷ (∃ R'. R = pure R')
  unfolding pure-def is-pure-alt-def by force

lemma is-pure-the-pure-id-eq[simp]: is-pure R ⟷ the-pure R = Id ⟷ R=pure Id
  by (auto simp: is-pure-conv)

lemma is-pure-iff-pure-assn: is-pure P = (∀ x x'. is-pure-assn (P x x'))
  unfolding is-pure-def is-pure-assn-def by metis

abbreviation hn-val R ≡ hn-ctxt (pure R)
lemma hn-val-unfold: hn-val R a b = ↑((b,a)∈R)
  by (simp add: hn-ctxt-def pure-def)

definition invalid-assn R x y ≡ ↑(∃ h. h|={ R x y) * true

abbreviation hn-invalid R ≡ hn-ctxt (invalid-assn R)
lemma invalidate-clone: R x y ⟷ A invalid-assn R x y * R x y
  apply (rule entailsI)
  unfolding invalid-assn-def
  apply (auto simp: models-in-range mod-star-trueI)
done

lemma invalidate-clone’: R x y ⟷ A invalid-assn R x y * R x y * true
  apply (rule entailsI)
  unfolding invalid-assn-def
  apply (auto simp: models-in-range mod-star-trueI)
done

lemma invalidate: R x y ⟷ A invalid-assn R x y
  apply (rule entailsI)
  unfolding invalid-assn-def
  apply (auto simp: models-in-range mod-star-trueI)
done

lemma invalid-pure-recover: invalid-assn (pure R) x y = pure R x y * true
  apply (rule ent-iffI)
  subgoal
    apply (rule entailsI)
unfolding invalid-assn-def
by (auto simp: pure-def)
subgoal
  unfolding invalid-assn-def
  by (auto simp: pure-def)
done

lemma hn-invalidI: h|=hn-ctxt P x y ⇒ hn-invalid P x y = true
  apply (cases h)
  apply (rule ent-iffI)
  apply (auto simp: invalid-assn-def hn-ctxt-def)
done

lemma invalid-assn-cong[cong]:
  assumes x≡x'
  assumes y≡y'
  assumes R x' y' ≡ R' x' y'
  shows invalid-assn R x y = invalid-assn R' x' y'
  using assms unfolding invalid-assn-def
by simp

1.2.2 Constraints in Refinement Relations

lemma mod-pure-conv[simp]: (h,as)|=pure R a b ←→ (as={} ∧ (b,a)∈R)
  by (auto simp: pure-def)

definition rdomp :: ('a ⇒ 'c ⇒ assn) ⇒ 'a ⇒ bool where
  rdomp R a ≡ ∃h c. h |= R a c

abbreviation rdom R ≡ Collect (rdomp R)

lemma rdomp-ctxt[simp]: rdomp (hn-ctxt R) = rdomp R
  by (simp add: hn-ctxt-def[abs-def])

lemma rdomp-pure[simp]: rdomp (pure R) a ←→ a∈Range R
  unfolding rdomp-def pure-def by auto

lemma rdom-pure[simp]: rdom (pure R) = Range R
  unfolding rdomp-def[abs-def] pure-def by auto

lemma Range-of-constraint-conv[simp]: Range (A∩UNIV×C) = Range A ∩ C
  by auto

1.2.3 Heap-Nres Refinement Calculus

Predicate that expresses refinement. Given a heap Γ, program c produces
a heap Γ' and a concrete result that is related with predicate R to some
abstract result from m

definition hn-refine Γ c Γ' R m ≡ nofail m →
\[<\Gamma> \ c <\lambda r. \ \Gamma' \ast (\exists A x. \ R x r \ast \uparrow(R\text{RETURN} \ x \leq m)) >_t\]

**simproc-setup** \texttt{assn-simproc-hnr (hn-refine} \ \Gamma \ c \ \Gamma')

\texttt{= {}〈K Seplogic-Auto.assn-simproc-fun〉}

**lemma** \texttt{hn-refineI[]intro?]:}
\texttt{assumes nofail \ m}
\texttt{implies \ <\Gamma> \ c <\lambda r. \ \Gamma' \ast (\exists A x. \ R x r \ast \uparrow(RETURN \ x \leq m)) >_t}
\texttt{shows \ hn-refine} \ \Gamma \ c \ \Gamma' \ R \ m
\texttt{using \ \texttt{assms unfolding \ \texttt{hn-refine-def by blast}}}

**lemma** \texttt{hn-refineD:}
\texttt{assumes \ hn-refine} \ \Gamma \ c \ \Gamma' \ R \ m
\texttt{assumes \ nofail} \ m
\texttt{shows \ <\Gamma> \ c <\lambda r. \ \Gamma' \ast (\exists A x. \ R x r \ast \uparrow(RETURN \ x \leq m)) >_t}
\texttt{using \ \texttt{assms unfolding \ \texttt{hn-refine-def by blast}}}

**lemma** \texttt{hn-refine-preI:}
\texttt{assumes} \ \bigwedge h. h\models \Gamma \implies \hn-refine} \ \Gamma \ c \ \Gamma' \ R \ a
\texttt{shows \ hn-refine} \ \Gamma \ c \ \Gamma' \ R \ a
\texttt{using \ \texttt{assms unfolding \ \texttt{hn-refine-def by (auto intro: hoare-triple-preI)}}}

**lemma** \texttt{hn-refine-nofailI:}
\texttt{assumes \ nofail} \ a \implies \hn-refine} \ \Gamma \ c \ \Gamma' \ R \ a
\texttt{shows \ hn-refine} \ \Gamma \ c \ \Gamma' \ R \ a
\texttt{using \ \texttt{assms by (auto simp: hn-refine-def)}}

**lemma** \texttt{hn-refine-false[simp]: \ hn-refine false} \ c \ \Gamma' \ R \ m
\texttt{by \ rule auto}

**lemma** \texttt{hn-refine-fail[simp]: \ hn-refine} \ \Gamma \ c \ \Gamma' \ R \ FAIL
\texttt{by \ rule auto}

**lemma** \texttt{hn-refine-frame:}
\texttt{assumes} \ hn-refine} \ P' \ c \ Q' \ R \ m
\texttt{assumes} \ P \implies \ F \ast P'
\texttt{shows} \ hn-refine} \ P \ c \ (F \ast Q') \ R \ m
\texttt{using \ \texttt{assms unfolding \ \texttt{hn-refine-def entailst-def}}}
\texttt{apply \ clarsimp}
\texttt{apply \ (erule \ cons-pre-rule)}
\texttt{apply \ (rule \ cons-post-rule)}
\texttt{apply \ (erule \ fi-rule, \ frame-inference)}
\texttt{apply \ (simp \ only: \ star-act)}
\texttt{apply \ \texttt{simp}}
\texttt{done

17
lemma hn-refine-cons:
assumes \( I: P \Rightarrow t P' \)
assumes \( R: \text{hn-refine } P' c Q R m \)
assumes \( I': Q \Rightarrow t Q' \)
assumes \( R': \forall x y. R x y \Rightarrow t R' x y \)
shows \( \text{hn-refine } P c Q R' m \)
using \( R \) unfolding \( \text{hn-refine-def} \)
apply clarify
apply (rule cons-pre-rulet[of \( I \)])
apply (rule cons-post-rulet)
apply assumption
apply (sep-auto simp: entailst-def)
apply (rule enttD)
apply (intro entt-star-mono \( I' R' \))
done

lemma hn-refine-cons-pre:
assumes \( I: P \Rightarrow t P' \)
assumes \( R: \text{hn-refine } P' c Q R m \)
shows \( \text{hn-refine } P c Q R m \)
by (rule hn-refine-cons[of \( I R \)]) sep-auto+

lemma hn-refine-cons-post:
assumes \( R: \text{hn-refine } P c Q R m \)
assumes \( I: Q \Rightarrow t Q' \)
shows \( \text{hn-refine } P c Q' R m \)
using assms
by (rule hn-refine-cons[of \( \text{entt-refl } - - \text{entt-refl} \)])

lemma hn-refine-cons-res:
[ \( \text{hn-refine } \Gamma f \Gamma' R g; \forall a c. R a c \Rightarrow t R' a c \) ] \( \Rightarrow \) \( \text{hn-refine } \Gamma f \Gamma' R' g \)
by (erule hn-refine-cons[of \( \text{entt-refl} \)]) sep-auto+

lemma hn-refine-ref:
assumes \( LE: m \leq m' \)
assumes \( R: \text{hn-refine } P c Q R m \)
shows \( \text{hn-refine } P c Q R m' \)
apply rule
apply (rule cons-post-rule)
apply (rule hn-refineD[of \( R \)])
using \( LE \) apply (simp add: pw-le-iff)
apply (sep-auto intro: order-trans[of \( - LE \)])
done

lemma hn-refine-cons-complete:
assumes \( I: P \Rightarrow t P' \)
assumes \( R: \text{hn-refine } P' c Q R m \)
assumes \( I': Q \Rightarrow t Q' \)
assumes $R': \forall x y. R x y \Rightarrow R' x y$
assumes $LE: m \leq m'$
shows $hn-refine P c Q' R' m'$
apply (rule $hn-refine-ref[OF LE]$)
apply (rule $hn-refine-cons[OF I R I' R']$)
done

lemma $hn-refine-augment-res$:
assumes $A: hn-refine \Gamma f \Gamma' R g$
assumes $B: g \leq n SPEC \Phi$
shows $hn-refine \Gamma f \Gamma' (\lambda a c. R a c * \uparrow(\Phi a)) g$
apply (rule $hn-refineI$)
apply (rule cons-post-rule)
apply (erule $A[THEN hn-refineD]$)
using $B$
apply (sep-auto simp: pw-le-iff pw-leaf-iff)
done

1.2.4 Product Types

Some notion for product types is already defined here, as it is used for currying and uncurrying, which is fundamental for the sepref tool

definition $prod-assn :: (a1 \Rightarrow c1 \Rightarrow assn) \Rightarrow (a2 \Rightarrow c2 \Rightarrow assn)$
$\Rightarrow (a1 * a2 \Rightarrow c1 * c2 \Rightarrow assn)$
where
$prod-assn P1 P2 a c \equiv case (a,c) of ((a1,a2),(c1,c2)) \Rightarrow P1 a1 c1 * P2 a2 c2$

notation $prod-assn$ (infixr $\times$ 70)

lemma $prod-assn-pure-conv[simp]$: $prod-assn (pure R1) (pure R2) = pure (R1 \times R2)$
by (auto simp: pure-def intro: ext)

lemma $prod-assn-pair-conv[simp]$: $prod-assn A B (a1,b1) (a2,b2) = A a1 a2 * B b1 b2$
unfolding $prod-assn-def$ by auto

lemma $prod-assn-true[simp]$: $prod-assn (\lambda -. true) (\lambda -. true) = (\lambda -. true)$
by (auto intro!: ext simp: hn-ctxt-def prod-assn-def)

1.2.5 Convenience Lemmas

lemma $hn-refine-guessI$:
assumes $hn-refine P f P' R f'$
assumes $f = f-conc$
shows $hn-refine P f-conc P' R f'$
— To prove a refinement, first synthesize one, and then prove equality
using $assms$ by simp
lemma imp-correctI:
  assumes $R$: hn-refine $\Gamma$ $c$ $\Gamma'$ $R$ $a$
  assumes $C$: $a \leq$ SPEC $\Phi$
  shows $\langle \Gamma > \cdot c < \lambda r'. \exists A r, \Gamma' \cdot R$ $r$ $r' \cdot \Uparrow (\Phi \ r) >t$
  apply (rule cons-post-rule)
  apply (rule hn-refineD[OF $R$])
  apply (rule le-RES-nofailI[OF $C$])
  apply (sep-auto dest: order-trans[OF - $C$])
  done

lemma hnr-pre-ex-conv:
  shows hn-refine ($\exists A x. \Gamma \cdot x$) $c$ $\Gamma'$ $R$ $a$ $\iff$ ($\forall x. \text{hn-refine} (\Gamma \cdot x) c$ $\Gamma'$ $R$ $a$)
  unfolding hn-refine-def
  by auto

lemma hnr-pre-pure-conv:
  shows hn-refine ($\Gamma \cdot \Uparrow P$) $c$ $\Gamma'$ $R$ $a$ $\iff$ ($P \longrightarrow \text{hn-refine} \Gamma c$ $\Gamma'$ $R$ $a$)
  unfolding hn-refine-def
  by auto

lemma hn-refine-split-post:
  assumes hn-refine $\Gamma c$ $\Gamma'$ $R a$
  shows hn-refine $\Gamma \cdot (\Gamma' \lor A) \Gamma''$ $R a$
  apply (rule hn-refine-cons-post[OF assms])
  by (rule entt-disjI1-direct)

lemma hn-refine-post-other:
  assumes hn-refine $\Gamma c$ $\Gamma''' R a$
  shows hn-refine $\Gamma c$ ($\Gamma' \lor A$ $\Gamma''$) $R a$
  apply (rule hn-refine-cons-post[OF assms])
  by (rule entt-disjI2-direct)

Return

lemma hnr-RETURN-pass:
  hn-refine (hn-ctxt $R$ $x$ $p$) (return $p$) (hn-invalid $R$ $x$ $p$) $R$ (RETURN $x$)
  — Pass on a value from the heap as return value
  apply rule
  apply (sep-auto simp: hn-ctxt-def eintros: invalidate-clone')
  done

lemma hnr-RETURN-pure:
  assumes $(c, a) \in R$

20
shows \( \text{hn-refine emp (return } c \text{) emp (pure } R \text{) (RETURN } a) \)
- Return pure value

unfolding \( \text{hn-refine-def} \) using \( \text{assms} \)
by \( \text{(sep-auto simp; pure-def)} \)

Assertion

lemma \( \text{hnr-FAIL} \) \( \text{[simp, intro!]} \): \( \text{hn-refine } \Gamma c \Gamma' R \text{ FAIL} \)
unfolding \( \text{hn-refine-def} \)
by \( \text{simp} \)

lemma \( \text{hnr-ASSERT} \)
assumes \( \Phi \Rightarrow \text{hn-refine } \Gamma c \Gamma' R c' \)
shows \( \text{hn-refine } \Gamma c \Gamma' R \text{ (do } \{ \text{ASSERT } \Phi; c' \}) \)
using \( \text{assms} \)
apply \( \text{(cases } \Phi) \)
by \( \text{auto} \)

Bind

lemma \( \text{bind-det-aux}: [\text{RETURN } x \leq m; \text{RETURN } y \leq f x] \Rightarrow \text{RETURN } y \leq m \gg f \)
apply \( \text{(rule order-trans[rotated])} \)
apply \( \text{(rule Refine-Basic.bind-mono)} \)
apply \( \text{assumption} \)
apply \( \text{(rule order-refl)} \)
apply \( \text{simp} \)
done

lemma \( \text{hnr-bind} \)
assumes \( D1: \text{hn-refine } \Gamma m' \Gamma 1 Rh m \)
assumes \( D2: \)
\[ \forall x x'. \text{RETURN } x \leq m \Rightarrow \text{hn-refine } (\Gamma 1 * \text{hn-ctxt } Rh x x') (f' x') (\Gamma 2 x x') \]
\( R (f x) \)
assumes \( \text{IMP}: \forall x x'. \Gamma 2 x x' \Rightarrow, \Gamma' * \text{hn-ctxt } Rx x x' \)
shows \( \text{hn-refine } \Gamma (m' \gg f') \Gamma' R (m \gg f) \)
using \( \text{assms} \)
unfolding \( \text{hn-refine-def} \)
apply \( \text{(clarsimp simp add: pw-bind-nofail)} \)
apply \( \text{(rule Hoare-Triple.bind-rule)} \)
apply \( \text{assumption} \)
apply \( \text{(clarsimp intro!: normalize-rules simp: hn-ctxt-def)} \)
proof -
fix \( x' x \)
assume \( 1: \text{RETURN } x \leq m \)
and \( \text{nofail m } \forall x. \text{inres m } x \Rightarrow \text{nofail } (f x) \)
hence \( \text{nofail } (f x) \) by \( \text{(auto simp; pu-le-iff)} \)
moreover assume \( \forall x x'. \text{RETURN } x \leq m \Rightarrow \)
\( \text{nofail } (f x) \Rightarrow <\Gamma 1 * Rh x x'> f' x' \)
\( <\lambda r'. \exists r. \Gamma 2 x x' * R r r' * \text{true } * \uparrow (\text{RETURN } r \leq f x)> \)
ultimately have $\bigwedge x'. <\Gamma 1 * Rh x x'> f' x'$

$<\lambda r'. \exists x'. \Gamma 2 x x' * R r r' * true * \uparrow (RETURN r \leq f x)>$

using 1 by simp

also have $\bigwedge r'. \exists x'. \Gamma 2 x x' * R r r' * true * \uparrow (RETURN r \leq f x) \Longrightarrow A$

$\exists x'. \Gamma' * R r r' * true * \uparrow (RETURN r \leq f x)$

apply (sep-auto)

apply (rule ent-frame-fwd[OF IMP[THEN enttD]])

apply frame-inference

apply (solve-entails)

done

finally (cons-post-rule) have

$R: <\Gamma 1 * Rh x x'> f' x'$

$<\lambda r'. \exists x'. \Gamma' * R r r' * true * \uparrow (RETURN r \leq f x)>$

show $<\Gamma 1 * Rh x x' * true > f' x'$

$<\lambda r'. \exists x'. \Gamma' * R r r' * true * \uparrow (RETURN r \leq m \gg f)>$

by (sep-auto heap: R intro: bind-det-aux[OF 1])

qed

Recursion

definition $hn-rel\ P\ m \equiv \lambda r. \exists x. P x r \uparrow (RETURN x \leq m)$

lemma $hn-refine-alt\ hn-refine\ Fpre\ c\ Fpost\ P\ m \equiv nofail\ m \rightarrow$

$<Fpre> c \ <\lambda r. \ hn-rel\ P\ m\ r\ *\ Fpost>_t$

apply (rule eq-reflection)

unfolding $hn-refine-def\ hn-rel-def$

apply (simp add: $hn-ctxt-def$)

apply (simp only: star-aci)

done

lemma $wit-swap-forall$

assumes $W: <P> c \ <\lambda.\ true>$

assumes $T: (\forall x. A x \rightarrow <P> c <Q x>)$

shows $<P> c \ <\lambda r. \ \neg A (\bigexists x. \uparrow (A x) \ast \neg A Q x r)>$

unfolding $hoare-triple-def\ Let-def$

apply (intro conjI impI allI)

subgoal by (elim conjE) (rule $hoare-tripleD[OF\ W]$, assumption+) []

subgoal

apply (clar simp, intro conjI allI)

apply1 (rule models-in-range)

apply1 (rule $hoare-tripleD[OF\ W]$, assumption; fail)

apply1 (simp only: disj-not2, intro impI)

apply1 (drule spec[OF T, THEN mp])

apply1 (drule (2) $hoare-tripleD(2)$)

by assumption

subgoal by (elim conjE) (rule $hoare-tripleD[OF\ W]$, assumption+)
subgoal by (elim conjE) (rule hoare-tripleD[of W], assumption+)
done

lemma hn-admissible:
  assumes PREC: precise Ry
  assumes E: ∀f∈A. nofail (f x) → (P c <λ. hn-rel Ry (f x) r F>)
  assumes NF: nofail (INF f:A. f x)
  shows (P c <λ. hn-rel Ry (INF f:A. f x) r F>)
proof –
  from NF obtain f where f∈A and nofail (f x)
    by (simp only: refine-pw-simps) blast

with E have (P c <λ. hn-rel Ry (f x) r F>) by blast
hence W: (P c <λ. true>) by (rule cons-post-rule, simp)

from E have
  E′: ∀f. f∈A ∧ nofail (f x) → (P c <λ. hn-rel Ry (f x) r F>)
  by blast
from wit-swap-forall[of W E′] have
  E′′: (P c <λ. ¬A (∃A xa. ↑(xa ∈ A ∧ nofail (xa x)) * ¬A (hn-rel Ry (xa x) r F))) .

thus ?thesis
  apply (rule cons-post-rule)
  unfolding entails-def hn-rel-def
  apply clarsimp
proof –
  fix h as p
  assume A: ∀f. f∈A → (∃a.
    ((h, as) |= Ry a p * F ∧ RETURN a ≤ f x)) ∨ ¬ nofail (f x)
  with f∈A and (nofail (f x)) obtain a where
    1: (h, as) |= Ry a p * F and RETURN a ≤ f x
    by blast
  have
    ∀f∈A. nofail (f x) → (h, as) |= Ry a p * F ∧ RETURN a ≤ f x
  proof clarsimp
    fix f′
    assume f′∈A and nofail (f′ x)
    with A obtain a′ where
      2: (h, as) |= Ry a′ p * F and RETURN a′ ≤ f′ x
      by blast
    moreover note preciseD′[of PREC 1 2]
    ultimately show (h, as) |= Ry a p * F ∧ RETURN a ≤ f′ x by simp
  qed
hence RETURN a ≤ (INF f:A. f x)
  by (metis (mono-tags) le-INF-iff le-nofailI)
with \( i \) show \( \exists a. (h, as) \models \text{Ry} p \bullet F \land \text{RETURN} a \leq (\text{INF} f : A. f x) \)
by blast
qed

lemma \textit{hn-admissible'}:
assumes \( \text{PREC}: \text{precise Ry} \)
assumes \( E: \forall f \in A. \text{nofail} (f x) \rightarrow <P> c <\lambda r. \text{hn-rel} \text{Ry} (f x) \circ F > t \)
assumes \( \text{NF}: \text{nofail} (\text{INF} f : A. f x) \)
shows \( <P> c <\lambda r. \text{hn-rel} \text{Ry} (\text{INF} f : A. f x) \circ F > t \)
apply (rule \text{hn-admissible}[OF \text{PREC}, \text{where} F = F \ast \text{true}, \text{simplified}])
apply simp
by fact+

lemma \textit{hnr-RECT-old}:
assumes \( S: \forall cf af ax px. \]
\( \forall ax px. \text{hn-refine} (\text{hn-ctxt} Rx ax px \bullet F) (cf px) (F' ax px) \text{Ry} (af ax)] \)
\( \Rightarrow \text{hn-refine} (\text{hn-ctxt} Rx ax px \bullet F) (cB cf px) (F' ax px) \text{Ry} (aB af ax) \]
assumes \( M: (\forall x. \text{mono-Heap} (\lambda f. cB f x)) \)
assumes \( \text{PREC}: \text{precise Ry} \)
shows \( \text{hn-refine} (\text{hn-ctxt} Rx ax px \bullet F) (\text{heap-fixp-fun} cB px) (F' ax px) \text{Ry} (\text{RECT} aB ax) \)
unfolding \textit{RECT-gfp-def}
proof (simp, intro conjI impI)
assume \text{trimono aB}

hence \text{mono aB} by (simp add: \text{trimonoD})

have \( \forall ax px. \)
\( \text{hn-refine} (\text{hn-ctxt} Rx ax px \bullet F) (\text{heap-fixp-fun} cB px) (F' ax px) \text{Ry} (gfp aB ax) \)
apply (rule \text{gfp-cadm-induct}[OF - - \langle \text{mono aB} \rangle])
apply rule
apply (auto simp: \text{hn-refine-alt intro: \text{hn-admissible'}[OF \text{PREC}]}) []
apply (auto simp: \text{hn-refine-alt}) []
apply clarsimp
apply (subst \text{heap-mono-body-fixp[of cB, OF M]})
apply (rule \text{S})
apply blast
thus \( \text{hn-refine} (\text{hn-ctxt} Rx ax px \bullet F) (\text{ccpo-fixp} \langle \text{fun-lub Heap-lub} \rangle \langle \text{fun-ord Heap-ord} \rangle cB px) (F' ax px) \text{Ry} (gfp aB ax) \) by simp
qed

lemma \textit{hnr-RECT}:
assumes \( S: \forall cf af ax px. \]
\( \forall ax px. \text{hn-refine} (\text{hn-ctxt} Rx ax px \bullet F) (cf px) (F' ax px) \text{Ry} (af ax)] \)
\[
\begin{align*}
\implies & \text{hn-refine (hn-ctxt Rx ax px \ast F)} (cB \ cf px) (F' \ ax px) \ Ry (aB \ af ax) \\
\text{assumes} & M : (\forall x. \ mono-Heap (\lambda f. cB f x)) \\
\text{shows} & \text{hn-refine} \\
& ((hn-ctxt Rx ax px \ast F) (heap.fixp-fun cB px) (F' \ ax px) \ Ry (RECT aB ax)) \\
\text{unfolding} & RECT-def \\
\text{proof} & (simp, intro conjI impI) \\
\text{assume} & trimono aB \\
\text{hence} & \text{flatf-mono-ge aB by (simp add: trimonoD)} \\
\text{have} & \forall ax px. \\
& \text{hn-refine (hn-ctxt Rx ax px \ast F)} (heap.fixp-fun cB px) (F' \ ax px) \ Ry \\
& (flatf-gfp aB ax) \\
\text{apply} & (rule flatf-ord.fixp-induct[OF - (flatf-mono-ge aB)]) \\
\text{apply} & (rule flatf-admissible-pointwise) \\
\text{apply} & simp \\
\text{apply} & (auto simp: hn-refine-alt []) \\
\text{apply} & clarsimp simp \\
\text{apply} & (subst heap.mono-body-fixp[of cB, OF M]) \\
\text{apply} & (rule S) \\
\text{apply} & blast \\
\text{done} \\
\text{thus} & \text{hn-refine (hn-ctxt Rx ax px \ast F)} \\
& (ccpo.fixp (fun-lub Heap-lub) (fun-ord Heap-ord) cB px) (F' \ ax px) \ Ry \\
& (flatf-gfp aB ax) by simp \\
\text{qed} \\
\end{align*}
\]

\textbf{lemma} \texttt{hnr-If}:
\textbf{assumes} \texttt{P: \Gamma \implies t \Gamma _1 \ast \text{hn-val bool-rel a \ a'}}
\textbf{assumes} \texttt{RT: a \implies \text{hn-refine (\Gamma _1 \ast \text{hn-val bool-rel a a'}) b' \Gamma _2b \ R b}}
\textbf{assumes} \texttt{RE: \neg a \implies \text{hn-refine (\Gamma _1 \ast \text{hn-val bool-rel a a'}) c' \Gamma _2c \ R c}}
\textbf{assumes} \texttt{IMP: \Gamma _2b \ \forall A \ \Gamma _2c \ \implies \Gamma '}
\textbf{shows} \texttt{\text{hn-refine} \Gamma (if a' then b' else c') \Gamma ' R (if a then b else c)}
\textbf{apply} (rule \texttt{hn-refine-cons[OF P])}
\textbf{apply1} (rule \texttt{hn-refine-preI})
\textbf{applyF} (cases a; simp add: hn-ctxt-def pure-def)
\textbf{focus}
\textbf{apply1} (rule \texttt{hn-refine-split-post})
\textbf{applyF} (rule \texttt{hn-refine-cons-pre[OF - RT])}
\textbf{applyS} (simp add: hn-ctxt-def pure-def)
\textbf{applyS} simp
\textbf{solved}
\textbf{solved}
\textbf{apply1} (rule \texttt{hn-refine-post-other})
\textbf{applyF} (rule \texttt{hn-refine-cons-pre[OF - RE])}
\textbf{applyS} (simp add: hn-ctxt-def pure-def)
\textbf{applyS} simp

25
solved
applyS (rule IMP)
applyS (rule entt-refl)
done

1.2.6 ML-Level Utilities

ML \[\text{signature SEPREF-BASIC = sig}\]
\(\text{(* Destroy lambda term, return function to reconstruct. Bound var is replaced by free. *)}\)
val dest-lambda-rc: Proof.context \(\rightarrow\) term \(\rightarrow\) ((term \(\ast\) (term \(\rightarrow\) term)) \(\ast\) Proof.context)
\(\text{(* Apply function under lambda. Bound var is replaced by free. *)}\)
val apply-under-lambda: (Proof.context \(\rightarrow\) term \(\rightarrow\) term) \(\rightarrow\) Proof.context \(\rightarrow\) term

\(\text{(* 'a nres type *)}\)
val is-nresT: typ \(\rightarrow\) bool
val mk-nresT: typ \(\rightarrow\) typ
val dest-nresT: typ \(\rightarrow\) typ

\(\text{(* Make certified == *)}\)
val mk-cequals: cterm \(\ast\) cterm \(\rightarrow\) cterm
\(\text{(* Make \(\Rightarrow\)A *)}\)
val mk-entails: term \(\ast\) term \(\rightarrow\) term

\(\text{(* Operations on pre-terms *)}\)
val constrain-type-pre: typ \(\rightarrow\) term \(\rightarrow\) term \(\ast\) t::T \(\ast\)

val mk-pair-in-pre: term \(\rightarrow\) term \(\rightarrow\) term \(\rightarrow\) term \(\ast\) (c,a) \(\in\) R \(\ast\)

val mk-compN-pre: int \(\rightarrow\) term \(\rightarrow\) term \(\rightarrow\) term \(\ast\) f o...o g \(\ast\)

val mk-curry0-pre: term \(\rightarrow\) term \(\ast\) curry0 f \(\ast\)
val mk-curry-pre: term \(\rightarrow\) term \(\ast\) curry f \(\ast\)
val mk-curryN-pre: int \(\rightarrow\) term \(\rightarrow\) term \(\ast\) curry (...(curry f)... \(\ast\))

val mk-uncurry0-pre: term \(\rightarrow\) term \(\ast\) uncurry0 f \(\ast\)
val mk-uncurry-pre: term \(\rightarrow\) term \(\ast\) uncurry f \(\ast\)
val mk-uncurryN-pre: int \(\rightarrow\) term \(\rightarrow\) term \(\ast\) uncurry (...(uncurry f)... \(\ast\))

\(\text{(* Conversion for hn-refine \(\ast\) terms*)}\)
val hn-refine-conv: conv \(\rightarrow\) conv \(\rightarrow\) conv \(\rightarrow\) conv \(\rightarrow\) conv \(\rightarrow\) conv

26
(\* Conversion on abstract value (last argument) of \texttt{hn-refine \textemdash \texttt{term \*)})
val hn-refine-con-a: conv \rightarrow \texttt{conv}

(\* Conversion on abstract value of \texttt{hn-refine term in conclusion of theorem \*)}
val hn-refine-concl-con-a: (\texttt{Proof.context \rightarrow \texttt{conv}}) \rightarrow \texttt{Proof.context \rightarrow \texttt{conv}}

(\* Destruct \texttt{hn-refine term \*)}
val dest-hn-refine: term \rightarrow \texttt{term \ast term \ast term \ast term \ast term}

(\* Make \texttt{hn-refine term \*)}
val mk-hn-refine: \texttt{term \ast term \ast term \ast term \ast term \rightarrow \texttt{term}}

(\* Check if given term is \texttt{Trueprop (hn-refine ...). Use with CONCL-COND'. \*)}
val is-hn-refine-concl: \texttt{term \rightarrow bool}

(\* Destruct \texttt{abs-fun}, returns \texttt{RETURN\textemdash flag, (f, args \*)} \*)
val dest-hnr-absfun: \texttt{term \rightarrow bool \ast (term \ast term list)}

(\* Make \texttt{abs-fun. \*)}
val mk-hnr-absfun: bool \ast (term \ast term list) \rightarrow \texttt{term}

(\* Make \texttt{abs-fun. Guess \texttt{RETURN\textemdash flag from type. \*)}
val mk-hnr-absfun': (term \ast term list) \rightarrow \texttt{term}

(\* Prove permutation of \ast. To be used with \texttt{f-tac-conv. \*)}
val star-permute-tac: \texttt{Proof.context \rightarrow tactic}

(\* Make separation conjunction \*)
val mk-star: \texttt{term \ast term \rightarrow \texttt{term}}

(\* Make separation conjunction from list. \texttt{[]} yields emp. \*)
val list-star: \texttt{term list \rightarrow \texttt{term}}

(\* Decompose separation conjunction. emp yields \texttt{[]} \*)
val strip-star: \texttt{term \rightarrow \texttt{term list}}

(\* Check if true\textemdash assertion \*)
val is-true: \texttt{term \rightarrow bool}

(\* Check if term is \texttt{hn-ctxt\textemdash assertion \*)}
val is-hn-ctxt: \texttt{term \rightarrow bool}

(\* Decompose \texttt{hn-ctxt\textemdash assertion \*)}
val dest-hn-ctxt: \texttt{term \rightarrow \texttt{term \ast term \ast term}}

(\* Decompose \texttt{hn-ctxt\textemdash assertion. NONE if term has wrong format \*)}
val dest-hn-ctxt-opt: \texttt{term \rightarrow (term \ast term \ast term) option}

\texttt{type phases-ctrl = \{}
\hspace{1em} trace: bool, \hspace{1em} (* Trace phases *)
\hspace{1em} int-res: bool, \hspace{1em} (* Stop with intermediate result *)
\hspace{1em} start: string option, \hspace{1em} (* Start with this phase. NONE: First phase *)
\hspace{1em} stop: string option \hspace{1em} (* Stop after this phase. NONE: Last phase *)
\}

27
val dflt-phases-ctrl = phases-ctrl
val dbg-phases-ctrl = phases-ctrl
val flag-phases-ctrl : bool

val PHASES' : phase list -> phases-ctrl -> Proof.context -> tactic' = struct

fun is-nresT (Type (@{type-name nres},[])) = true | is-nresT = false
fun mk-nresT T = Type(@{type-name nres},[T])
fun dest-nresT (Type (@{type-name nres},[T])) = T | dest-nresT T = raise TYPE(dest-nresT,[T],[[]])

fun dest-lambda-rc ctxt (Abs(x,T,t)) = let
  val (u,ctxt) = yield-singleton Variable.variant-fixes x ctxt
  val u = Free(u,T)
  val t = subst-bound(u,t)
  val reconstruct = Term.lambda-name (x,u)
  in
    [(t,reconstruct),ctxt]
  end
| dest-lambda-rc - t = raise TERM(dest-lambda-rc,[t])

fun apply-under-lambda f ctxt t = let
  val ((t,rc),ctxt) = dest-lambda-rc ctxt t
  val t = f ctxt t
  in
    rc t
  end

fun mk-pair-in-pre x y r = Const (@{const-name Set.member}, dummyT) $ (Const (@{const-name Product-Type.Pair}, dummyT) $ x $ y) $ r
fun mk-uncurry-pre \( t \) = Const(\[@\{\text{const-name uncurry}\}, \text{dummyT}\]$t
fun mk-uncurry0-pre \( t \) = Const(\[@\{\text{const-name uncurry0}\}, \text{dummyT}\]$t
fun mk-uncurryN-pre \( n \) = mk-uncurry0-pre 0
| mk-uncurryN-pre \( n \) = mk-uncurry-pre \( n \) $\Rightarrow$ mk-uncurryN-pre \( n-1 \)

fun mk-curry-pre \( t \) = Const(\[@\{\text{const-name curry}\}, \text{dummyT}\]$t
fun mk-curry0-pre \( t \) = Const(\[@\{\text{const-name curry0}\}, \text{dummyT}\]$t
fun mk-curryN-pre \( n \) = mk-curry0-pre 0
| mk-curryN-pre \( n \) = mk-curry-pre \( n \) $\Rightarrow$ mk-curryN-pre \( n-1 \)

fun mk-compN-pre \( n \) \( f \) \( g \) = \( f \) $\Rightarrow$ \( g \)
| mk-compN-pre \( n \) \( f \) \( g \) = let
val \( g \) = fold \((\text{fn } i \Rightarrow \text{fn } t \Rightarrow t\text{-Bound } i)\) \((n-2)\text{-downto } 0\) \( g \)
val \( t \) = Const(\[@\{\text{const-name Fun}.\,\text{comp}\}, \text{dummyT}\]$ f $\Rightarrow$ g
val \( t \) = fold \((\text{fn } i \Rightarrow \text{fn } t \Rightarrow \text{Abs } (x\text{-string-of-int } i, \text{dummyT}, t))\) \((n-1)\text{-downto } 1\) \( t \)
in
\( t \)
end

fun constrain-type-pre \( T \) \( t \) = Const(\[@\{\text{syntax-const-type-constraint-}\}, T\Rightarrow T\]$ $\Rightarrow$ t

local open Conv in
fun hn-refine-conv c1 c2 c3 c4 c5 ct = case Thm.\text{term-of } ct of
\(@\{\text{mpat hn-refine - - - - -}\} \Rightarrow \text{let}
val cc = combination-conv
in
cc (cc (cc (cc (cc all-conv c1) c2) c3) c4) c5 ct
end
| - \Rightarrow \text{raise CTERM (hn-refine-conv,}[ct])
val hn-refine-conv-a = hn-refine-conv all-conv all-conv all-conv all-conv all-conv

fun hn-refine-concl-conv-a conv ctxt = Refine-Util.HOL-concl-conv (fn ctxt => hn-refine-conv-a (conv ctxt)) ctxt
end

(* \text{FIXME}: Strange dependency! *)
val mk-cequals = uncurry SMT-Util.mk-cequals

29
val mk-entails = HOLogic.mk_binrel @{const-name entails}

val mk-star = HOLogic.mk_binop @{const-name Groups.times-class.times}

fun list-star [] = @{term emp::assn}
  | list-star [a] = a
  | list-star (a::l) = mk-star (list-star l, a)

fun strip-star @{mpat ?a*?b} = strip-star a @ strip-star b
  | strip-star @{mpat emp} = []
  | strip-star t = [t]

fun is-true @{mpat true} = true
  | is-true = false

fun is-hn ctxt @{mpat hn-ctxt - - -} = true
  | is-hn ctxt = false

fun dest-hn ctxt @{mpat hn-ctxt ?R ?a ?p} = (R, a, p)
  | dest-hn ctxt = raise TERM (dest-hn ctxt, [t])

fun dest-hn ctxt opt @{mpat hn-ctxt ?R ?a ?p} = SOME (R, a, p)
  | dest-hn ctxt opt = NONE

fun strip-abs-args (t as @{mpat PR-CONST -}) = (t, [])
  | strip-abs-args @{mpat ?f$?a} = (case strip-abs-args f of (f, args) => (f, args @{mpat ?a}))
  | strip-abs-args t = (t, [])

fun dest-hn-absfun @{mpat RETURN$?a} = (true, strip-abs-args a)
  | dest-hn-absfun f = (false, strip-abs-args f)

fun mk-hn-absfun (true, fa) = Autoref-Tagging.list-APP fa @{mk-term RETURN$?a}
  | mk-hn-absfun (false, fa) = Autoref-Tagging.list-APP fa

fun mk-hn-absfun' fa = let
  val t = Autoref-Tagging.list-APP fa
  val T = fastype-of t
  in
  case T of
  Type (@{type-name nres}, -) => t
  | _ => @{mk-term RETURN$?t}
  end

  | dest-hn-refine t = raise TERM (dest-hn-refine, [t])


val is-hn-refine-concl = can (HOLogic.dest_Trueprop @> dest-hn-refine)
fun star-permute-tac ctxt = ALLGOALS (simp-tac (put-simpset HOL-basic-ss
cxt addssimps @{ths star-aci}))

type phases-ctrl = {
    trace: bool,
    int-res: bool,
    start: string option,
    stop: string option
}

val dflt-phases-ctrl = {
    trace = false,
    int-res = false,
    start = NONE,
    stop = NONE
}
val dbg-phases-ctrl = {
    trace = true,
    int-res = true,
    start = NONE,
    stop = NONE
}

fun flag-phases-ctrl dbg = if dbg then dbg-phases-ctrl else dflt-phases-ctrl

fun ph-range phases start stop = let
    fun find-phase name = let
        val i = find-index (fn (n, _, _) => n=name) phases
    in
        if i < 0 then error (No such phase: "name") else ()
        i
    end
    val i = case start of NONE => 0 | SOME n => find-phase n
    val j = case stop of NONE => length phases - 1 | SOME n => find-phase n
    in
        phases = take (j+1) phases |> drop i
        val - = case phases of [] => error No phases selected, range is empty |
        => ()
        in
            phases
        end
    in

fun PHASES' phases ctrl ctxt = let
    val phases = ph-range phases (#start ctrl) (#stop ctrl)
    val phases = map (fn (n, tac, d) => (n, tac ctxt, d)) phases
    fun r [] - st = Seq.single st
    | r ((name, tac, d)::tacs) i st = let
        val n = Thm.nprems_of st
        val bailout-tac = if #int-res ctrl then all-tac else no-tac
        fun trace-tac msg st = (if #trace ctrl then tracing msg else ()); Seq.single

    in
        Seq.fold (fn (n, tac, d) => r (tacs) i (tac ctxt (tac ctxt n st))) st
    end
end
val trace-start-tac = trace-tac (Phase name) in
  K trace-start-tac THEN' IF-EXGOAL (tac)
  THEN-ELSE' (fn i => fn st =>
    (* Bail out if a phase does not solve/create exactly the expected subgoals *)
    if Thm.nprems-of st = n+d then
      ((trace-tac Done THEN r tacs i) st)
    else
      (trace-tac *** Wrong number of produced goals THEN bailout-tac)
  st)
  ,
  K (trace-tac *** Phase tactic failed THEN bailout-tac))
end i st
in
  r phases
end

(* Perform sequence of tactics (tac,n), each expected to create n new goals, or solve goals if n is negative. Debug-flag: Stop with intermediate state after tactic fails or produces less/more goals as expected. *)
val PHASES': phase list -> phases-ctrl -> Proof.context -> tactic'

(*
fun xPHASES' dbg tacs ctxt = let
  val tacs = map (fn (tac,d) => (tac ctxt,d)) tacs
  fun r [] - st = Seq.single st
    | r ((tac,d)::tacs) i st = let
      val n = Thm.nprems-of st
      val bailout-tac = if dbg then all-tac else no-tac
      in
        IF-EXGOAL (tac)
        THEN-ELSE' (fn i => fn st =>
          (* Bail out if a phase does not solve/create exactly the expected subgoals *)
          if Thm.nprems-of st = n+d then
            st)
      end
    end
  in
    r
  end

(*
  Seq.fold (fn tac => tac ctxt) tacs st
(r tacs i st)
  else
  bailout-tac st

  $K$ bailout-tac)
end i st

in
  r tacs
end

*)
end

signature SEPREF-DEBUGGING = sig
(**************************************************************************)
(* Debugging *)
(* Centralized debugging mode flag *)
val cfg-debug-all: bool Config.

val is-debug: bool Config.T -> Proof.context -> bool
val is-debug': Proof.context -> bool

(* Conversion, trace errors if custom or central debugging flag is activated *)
val DBG-CONVERSION: bool Config.T -> Proof.context -> conv -> tactic'

(* Conversion, trace errors if central debugging flag is activated *)
val DBG-CONVERSION': Proof.context -> conv -> tactic'

(* Conversion, trace errors if central debugging flag is activated *)
val tracing-tac': string -> Proof.context -> tactic'
(* Warning message and current subgoal *)
val warning-tac': string -> Proof.context -> tactic'
(* Error message and current subgoal *)
val error-tac': string -> Proof.context -> tactic'

(* Trace debug message *)
val dbg-trace-msg: bool Config.T -> Proof.context -> string -> unit
val dbg-trace-msg': Proof.context -> string -> unit

val dbg-msg-tac: bool Config.T -> (Proof.context -> int -> thm -> string)
  -> Proof.context -> tactic'
val dbg-msg-tac': (Proof.context -> int -> thm -> string) -> Proof.context
  -> tactic'

val msg-text: string -> Proof.context -> int -> thm -> string
val msg-subgoal: string -> Proof.context -> int -> thm -> string
val msg-from-subgoal: string -> (term -> Proof.context -> string) ->
  Proof.context -> int -> thm -> string
val msg-allgoals: string -> Proof.context -> int -> thm -> string
end

structure Sepref-Debugging: SEPREF-DEBUGGING = struct

val cfg-debug-all = Attrb.setup-config-bool @{binding sepref-debug-all} (K false)

fun is-debug cfg ctxt = Config.get ctxt cfg orelse Config.get ctxt cfg-debug-all
fun is-debug' ctxt = Config.get ctxt cfg-debug-all

fun dbg-trace cfg ctxt obj = if is-debug cfg ctxt then tracing (@{make-string} obj) else ()
fun dbg-trace' ctxt obj = if is-debug' ctxt then tracing (@{make-string} obj) else ()

fun dbg-trace-msg cfg ctxt msg = if is-debug cfg ctxt then tracing msg else ()
fun dbg-trace-msg' ctxt msg = if is-debug' ctxt then tracing msg else ()

fun DBG-CONVERSION cfg ctxt cv i st = Seq.single (Conv.gconv-rule cv i st)
  handle e as THM =>> (dbg-trace cfg ctxt e; Seq.empty)
  | e as CTERM =>> (dbg-trace cfg ctxt e; Seq.empty)
  | e as TERM =>> (dbg-trace cfg ctxt e; Seq.empty)
  | e as TYPE =>> (dbg-trace cfg ctxt e; Seq.empty);

fun DBG-CONVERSION' ctxt cv i st = Seq.single (Conv.gconv-rule cv i st)
  handle e as THM =>> (dbg-trace' ctxt e; Seq.empty)
  | e as CTERM =>> (dbg-trace' ctxt e; Seq.empty)
  | e as TERM =>> (dbg-trace' ctxt e; Seq.empty)
  | e as TYPE =>> (dbg-trace' ctxt e; Seq.empty);

local
  fun gen-subgoal-msg-tac do-msg msg ctxt = IF-EXGOAL (fn i => fn st =>>
let
val t = nth (Thm.prems-of st) (i-1)
val - = Pretty.block [Pretty.str msg, Pretty.fbrk, Syntax.pretty-term ctxt t]
  |> Pretty.string-of |> do-msg

in
  Seq.single st
end)
in
val tracing-tac' = gen-subgoal-msg-tac tracing
val warning-tac' = gen-subgoal-msg-tac warning
val error-tac' = gen-subgoal-msg-tac error
end

fun dbg-msg-tac cfg msg ctxt =
  if is-debug cfg ctxt then
    (fn i => fn st => (tracing (msg ctxt i st); Seq.single st))
  else K all-tac
fun dbg-msg-tac' msg ctxt =
  if is-debug' ctxt then
    (fn i => fn st => (tracing (msg ctxt i st); Seq.single st))
  else K all-tac
fun msg-text msg = msg
fun msg-from-subgoal msg sgmsg ctxt i st =
  case try (nth (Thm.prems-of st) (i-1)) of
    NONE => msg ^ \n ^ Subgoal out of range
    | SOME t => msg ^ \n ^ sgmsg t ctxt
fun msg-subgoal msg = msg-from-subgoal msg (fn t => fn ctxt =>
  Syntax.pretty-term ctxt t |> Pretty.string-of)

fun msg-allgoals msg ctxt st =
  msg ^ \n ^ Pretty.string-of (Pretty.chunks (Goal-Display.pretty-goals ctxt st))
end

ML |

(* Tactics for produced subgoals *)
infix 1 THEN-NEXT THEN-ALL-NEW-LIST THEN-ALL-NEW-LIST'
signature STACTICAL = sig
  (* Apply first tactic on this subgoal, and then second tactic on next subgoal *)
  val THEN-NEXT: tactic' * tactic' => tactic'
  (* Apply tactics to the current and following subgoals *)
  val APPLY-LIST: tactic' list => tactic'
(* Apply list of tactics on subgoals emerging from tactic. Requires exactly one tactic per emerging subgoal.*)

val THEN-ALL-NEW-LIST : tactic' * tactic' list -> tactic'

(* Apply list of tactics to subgoals emerging from tactic, use fallback for additional subgoals.*)

val THEN-ALL-NEW-LIST' : tactic' * (tactic' list * tactic') -> tactic'

end

structure STactical : STACTICAL = struct

infix 1 THEN-WITH-GOALDIFF

fun (tac1 THEN-WITH-GOALDIFF tac2) st = let
    val n1 = Thm.nprems-of st
  in
    st |> (tac1 THEN (fn st => tac2 (Thm.nprems-of st - n1) st ))
  end

fun (tac1 THEN-NEXT tac2) i =
  tac1 i THEN-WITH-GOALDIFF (fn d =>
    if d < ~1 then
      (error THEN-NEXT: Tactic solved more than one goal; no-tac)
    else
      tac2 (i+1+d)
  )

fun APPLY-LIST [] = K all-tac
    | APPLY-LIST (tac::tacs) = tac THEN-NEXT APPLY-LIST tacs

fun (tac1 THEN-ALL-NEW-LIST tacs) i =
  tac1 i THEN-WITH-GOALDIFF (fn d =>
    if d+1 < length tacs then (error THEN-ALL-NEW-LIST: Tactic produced wrong number of goals; no-tac)
    else
      APPLY-LIST tacs i
  )

fun (tac1 THEN-ALL-NEW-LIST' (tacs,rtac)) i =
  tac1 i THEN-WITH-GOALDIFF (fn d => let
    val - = if d+1 < length tacs then error THEN-ALL-NEW-LIST': Tactic produced too few goals else ()
    in
      tacs' = tacs @ replicate (d + 1 - length tacs) rtac
      APPLY-LIST tacs' i
    end)

end
1.3 Monadify

theory Sepref-Monadify
imports Sepref-Basic Sepref-Id-Op

begin

In this phase, a monadic program is converted to complete monadic form, that is, computation of compound expressions are made visible as top-level operations in the monad.

The monadify process is separated into 2 steps.

1. In a first step, eta-expansion is used to add missing operands to operations and combinators. This way, operators and combinators always occur with the same arity, which simplifies further processing.

2. In a second step, computation of compound operands is flattened, introducing new bindings for the intermediate values.

definition SP — Tag to protect content from further application of arity and combinator equations
where [simp]: SP x ≡ x

lemma SP-cong [cong]: SP x ≡ SP x by simp

lemma PR-CONST-cong [cong]: PR-CONST x ≡ PR-CONST x by simp

definition RCALL — Tag that marks recursive call
where [simp]: RCALL D ≡ D

definition EVAL — Tag that marks evaluation of plain expression for monadify phase
where [simp]: EVAL x ≡ RETURN x

Internally, the package first applies rewriting rules from sepref-monadify-arity, which use eta-expansion to ensure that every combinator has enough actual parameters. Moreover, this phase will mark recursive calls by the tag RCALL.

Next, rewriting rules from sepref-monadify-comb are used to add EVAL-tags to plain expressions that should be evaluated in the monad. The EVAL tags are flattened using a default simproc that generates left-to-right argument order.

lemma monadify-simps:
  Refine-Basic.bind$(\text{return}$x)$\lambda x. f x) = f x
\[ \text{EVAL}x \equiv \text{RETURN}x \]
by \text{simp-all} 

\textbf{definition} [\text{simp}]: \text{PASS} \equiv \text{RETURN}
— Pass on value, invalidating old one

\textbf{lemma} \text{remove-pass-simps}:
\begin{align*}
\text{Refine-Basic.bind}$(\text{PASS}$x)$(\lambda_2 x. f x) & \equiv f x \\
\text{Refine-Basic.bind}$(m$)(\lambda_2 x. \text{PASS}$x) & \equiv m
\end{align*}
by \text{simp-all}

\textbf{definition} \text{COPY} : '\text{a} \Rightarrow \text{'a}'
— Marks required copying of parameter

\textbf{where} [\text{simp}]: \text{COPY}x \equiv x

\textbf{lemma} \text{RET-COPY-PASS-eq}: \text{RETURN}$\text{(COPY}p) = \text{PASS}$p by \text{simp}

\textbf{named-theorems-rev} \text{sepref-monadify-arity Sepref.Monadify: Arity alignment equations}
\textbf{named-theorems-rev} \text{sepref-monadify-comb Sepref.Monadify: Combinator equations}

\textbf{ML} (\textbackslash)
\begin{verbatim}
structure Sepref-Monadify = struct
  local
  fun cr-var (i, T) = (vˆstring-of-int i, Free (\--vˆstring-of-int i, T))

  fun lambda2-name n t = let
    val t = @\{mk-term PROTECT2 ?t DUMMY\}
  in
    Term.lambda-name n t
  end

  fun monadify t = let
    val (f, args) = Autoref-Tagging.strip-app t
    val - = not (is-Abs f) orelse raise TERM (monadify: higher-order,[t])
    val argTs = map fastype-of args
    (*val args = map monadify args*)
    val args = map (fn a => @\{mk-term EVAL$a\}) args
end
\end{verbatim}

38
(*val fT = fastype-of f
val argTs = binder-types fT*)

val argVs = tag-list 0 argTs
|> map cr-var

val res0 = let
  val x = Autoref-Tagging.list-APP (f, map #2 argVs)
in
  @{mk-term SP (RETURN $?x)}
end

val res = bind-args res0 (argVs ~> args)
in
res
end

fun monadify-conv-aux ctxt ct = case Thm.term-of ct of
  @{mpat EVAL$-} => let
    val ss = put-simpset HOL-basic-ss ctxt
    val ss = (ss addsimps @{thms monadify-simps SP-def})
    val tac = (simp-tac ss 1)
in (*Refine-Util.monitor-conv monadify*) (Refine-Util.f-tac-conv ctxt (dest-comb #> #2 #> monadify) tac) ct
end
| t => raise TERM (monadify-conv, [t])

(*fun extract-comb-conv ctxt = Conv.rewrs-conv
 (Named-Theorems-Rev.get ctxt @{named-theorems-rev sepref-monadify-evalcomb})
*)
in
(*
val monadify-conv = Conv.top-conv
(fn ctxt =>
  Conv.try-conv (extract-comb-conv ctxt else-conv monadify-conv-aux ctxt)
)
*)

val monadify-simproc = Simplifier.make-simproc @{context} monadify-simproc
  {lhss = [Logic.varify-global @{term EVAL$a}],
     proc = K (try o monadify-conv-aux)};

end
local
open Sepref-Basic
fun mark-params t = let
  val (P,c,Q,R,a) = dest-hn-refine t
  val pps = strip-star P |> map-filter (dest-hn-ctxt-opt #> map-option #2)
  val a = tr [] a
in
  mk-hn-refine (P,c,Q,R,a)
end

fun tr env (t as @{mpat RETURN$?x}) = 
  if is-Bound x orelse member (aconv) pps x then
    @{mk-term env: PASS$?x}
  else t
  | tr env (t1$t2) = tr env t1 $ tr env t2
  | tr env (Abs (x,T,t)) = Abs (x,T,tr (T::env) t)
  | tr - t = t

val a = tr [] a
in
  mk-hn-refine (P,c,Q,R,a)
end

fun mark-params-conv ctxt = Refine-Util.f-tac-conv ctxt (mark-params)
  (simp-tac (put-simpset HOL-basic-ss ctxt addsimps @{thms PASS-def}) 1)
end

local
open Sepref-Basic

fun dp ctxt (@{mpat Refine-Basic.bind$(PASS$?p)$($t'$ AS_p ($\lambda$. PROTECT2 - DUMMY))}) = 
  let
    val (t',ps) = let
      val ((t',rc),ctxt) = dest-lambda-rc ctxt t'
      val f = case t' of @{mpat PROTECT2 ?f -} => f | _ => raise Match
      val (f,ps) = dp ctxt f
      val t' = @{mk-term PROTECT2 ?f DUMMY}
      val t' = rc t'
    in
      (t',ps)
    end
    val dup = member (aconv) ps p
    val t = if dup then
      @{mk-term Refine-Basic.bind$(RETURN$(COPY$?p))$?$t'}
    else
      @{mk-term Refine-Basic.bind$(PASS$?p)$?$t'}
  in
  end
fun dp-conv ctxt = Refine-Util.f-tac-conv ctxt
  (#1 o dp ctxt)
  (ALLGOALS (simp-tac (put-simpset HOL-basic-ss ctxt addsimps @{thms
  RET-COPY-PASS-eq})))

fun dup-tac ctxt = CONVERSION (Sepref-Basic.hn-refine-concl-conv-a dp-conv
  ctxt)
end

fun arity-tac ctxt = let
  val arity1-ss = put-simpset HOL-basic-ss ctxt
    addsimps ((Named-Theorems-Rev.get ctxt @{named-theorems-rev
    sepref-monadify-arity}))
    |> Simplifier.add-cong @{thm SP-cong}
    |> Simplifier.add-cong @{thm PR-CONST-cong}

  val arity2-ss = put-simpset HOL-basic-ss ctxt
    addsimps @{thms beta SP-def}
in
  simp-tac arity1-ss THEN' simp-tac arity2-ss
end

fun comb-tac ctxt = let
  val comb1-ss = put-simpset HOL-basic-ss ctxt
    addsimps (Named-Theorems-Rev.get ctxt @{named-theorems-rev
    sepref-monadify-comb})
    (*addsimps (Named-Theorems-Rev.get ctxt @{named-theorems-rev
    sepref-monadify-evalcomb})*)
    addsimprocs [monadify-simproc]
    |> Simplifier.add-cong @{thm SP-cong}
    |> Simplifier.add-cong @{thm PR-CONST-cong}

  val comb2-ss = put-simpset HOL-basic-ss ctxt
    addsimps @{thms SP-def}
  in
  simp-tac comb1-ss THEN' simp-tac comb2-ss
end

(fun ops-tac ctxt = CONVERSION (Sepref-Basic.hn-refine-concl-conv-a
  monadify-conv ctxt)*)

fun mark-params-tac ctxt = CONVERSION (}
Refine-Util.HOL-concl-conv (K (mark-params-conv ctxt)) ctxt

fan contains-eval @@\{mpat Trueprop (hn-refine - - - ?a)\} =
Term.exists-subterm (fn \a\{mpat EVAL\} => true | - => false) a
| contains-eval t = raise TERM{contains-eval,[t]};

fan remove-pass-tac ctxt =
simp-tac {put-simpset HOL-basic-ss ctxt addsimps @\{thms remove-pass-simps\}}

fan monadify-tac dbg ctxt = let
open Sepref-Basic
in
PHASES' [
  (arity, arity-tac, 0),
  (comb, comb-tac, 0),
  (*{ops, ops-tac, 0},*)
  (check-EVAL, K (CONCL-COND' (not o contains-eval)), 0),
  (mark-params, mark-params-tac, 0),
  (dup, dup-tac, 0),
  (remove-pass, remove-pass-tac, 0)
] (flag-phases-ctrl dbg) ctxt
end

end

lemma dflt-arity[sepref-monadify-arity]:
RETURN \equiv \lambda_2 x. SP RETURN\$x
RECT \equiv \lambda_2 B x. SP RECT\$\b{\lambda_2 D x. B\$(\lambda_2 x. RCALL\$D\$x)\$x}
case-list \equiv \lambda_2 fn fc l. SP case-list\$fn\$(\lambda_2 x x. fc\$x\$xs)\$l
case-prod \equiv \lambda_2 fp p. SP case-prod\$(\lambda_2 a b. fp\$a\$b)\$p
case-option \equiv \lambda_2 fn fs ov. SP case-option\$fn\$(\lambda_2 x. fs\$x)\$ov
If \equiv \lambda_2 b t e. SP If\$b\$t\$e
Let \equiv \lambda_2 x f. SP Let\$x\$(\lambda_2 x. f\$x)
by (simp-all only: SP-def APP-def PROTECT2-def RCALL-def)

lemma dflt-comb[sepref-monadify-comb]:
\forall B x. RECT\$B\$x \equiv Refine-Basic.bind\$(EVAL\$x)\$\b{\lambda_2 x. SP (RECT\$B\$x)}
\forall D x. RCALL\$D\$x \equiv Refine-Basic.bind\$(EVAL\$x)\$\b{\lambda_2 x. SP (RCALL\$D\$x)}
\forall fn fc l. case-list\$fn\$fc\$l \equiv Refine-Basic.bind\$(EVAL\$l)\$\b{\lambda_2 l. (SP case-list\$fn\$fc\$l)}
\forall p. case-prod\$fp\$p \equiv Refine-Basic.bind\$(EVAL\$p)\$\b{\lambda_2 p. (SP case-prod\$fp\$p)}
\forall fn fs ov. case-option\$fn\$fs\$ov
\equiv Refine-Basic.bind\$(EVAL\$ov)\$\b{\lambda_2 ov. (SP case-option\$fn\$fs\$ov)}
\forall b t e. If\$b\$t\$e \equiv Refine-Basic.bind\$(EVAL\$b)\$\b{\lambda_2 b. (SP If\$b\$t\$e)}
\forall x. RETURN\$x \equiv Refine-Basic.bind\$(EVAL\$x)\$\b{\lambda_2 x. SP (RETURN\$x)}
\forall x f. Let\$x\$f \equiv Refine-Basic.bind\$(EVAL\$x)\$\b{\lambda_2 x. (SP Let\$x\$f)}
by (simp-all)
lemma dflt-plain-comb[sepref-monadify-comb]:
EVAL($\text{If}\,b\,t\,e\equiv \text{Refine-Basic}$.bind($EVAL(b)$($\lambda_2 b. \text{If}\,b\,(EVAL(t))$($EVAL(e)$))
EVAL($\text{case-list} fn(l x s. fe x xs)$l) ≡
Refine-Basic.bind($EVAL(l)$($\lambda_2 l. \text{case-list} fn(l x s. EVAL(fe x xs)$)
EVAL($\text{case-prod} fn(l x s. fc x xs)$p) ≡
Refine-Basic.bind($EVAL(p)$($\lambda_2 p. \text{case-prod} fn(l x s. EVAL(fc x xs)$)
EVAL($\text{case-option} fn(l x s. fs x)$ov) ≡
Refine-Basic.bind($EVAL(ov)$($\lambda_2 ov. \text{case-option} fn(l x s. EVAL(fs x)$)
EVAL ($\text{Let}\,v\,(\lambda \,x. f x))\equiv \Rightarrow (\Rightarrow)\Rightarrow (\Rightarrow)\Rightarrow (\Rightarrow)\Rightarrow (\Rightarrow)\Rightarrow (\Rightarrow)\Rightarrow (\Rightarrow)
apply (\text{rule eq-reflection, simp split: list.split prod.split option.split})+
done

lemma evalcomb-PR-CONST[sepref-monadify-comb]:
EVAL($\text{PR-CONST} x\equiv \text{SP} (\text{RETURN} (\text{PR-CONST} x))$
by simp

end

theory Sepref-Constraints
imports Main Automatic-Refinement.Refine-Lib Sepref-Basic
begin

definition CONSTRAINT-SLOT (x::prop) ≡ x

lemma insert-slot-rl1:
assumes PROP P ⇒ PROP (CONSTRAINT-SLOT (Trueprop True)) ⇒ PROP Q
shows PROP (CONSTRAINT-SLOT (PROP P)) ⇒ PROP Q
using assms unfolding CONSTRAINT-SLOT-def by simp

lemma insert-slot-rl2:
assumes PROP P ⇒ PROP (CONSTRAINT-SLOT S) ⇒ PROP Q
shows PROP (CONSTRAINT-SLOT (PROP S k&& PROP P)) ⇒ PROP Q
using assms unfolding CONSTRAINT-SLOT-def conjunction-def .

lemma remove-slot: PROP (CONSTRAINT-SLOT (Trueprop True))
unfolding CONSTRAINT-SLOT-def by (rule TrueI)

definition CONSTRAINT where [simp]: CONSTRAINT P x ≡ P x

lemma CONSTRAINT-D:
assumes CONSTRAINT (P::'a => bool) x
shows P x
using assms unfolding CONSTRAINT-def by simp

43
lemma CONSTRAINT-I:
  assumes P x
  shows CONSTRAINT (P::'a => bool) x
  using assms unfolding CONSTRAINT-def by simp

Special predicate to indicate unsolvable constraint. The constraint solver refuses to put those into slot. Thus, adding safe rules introducing this can be used to indicate unsolvable constraints early.

definition CN-FALSE :: ('a=>bool) => 'a => bool where [simp]: CN-FALSE P x
  ≡ False
lemma CN-FALSEI: CN-FALSE P x => P x by simp

named-theorems constraint-simps (Simplification of constraints)

named-theorems constraint-abbrevs (Constraint Solver: Abbreviations)
lemmas split-constraint-rls =
  atomize-conj[symmetric] imp-conjunction all-conjunction conjunction-imp

ML

signature SEPREF-CONSTRAINTS = sig
  (****** Constraint Slot *)
  (Tactic with slot subgoal *)
  val WITH-SLOT: tactic' => tactic
  (Process all goals in slot *)
  val ON-SLOT: tactic => tactic
  (Create slot as last subgoal. Fail if slot already present. *)
  val create-slot-tac: tactic
  (Create slot if there isn’t one already *)
  val ensure-slot-tac: tactic
  (Remove empty slot *)
  val remove-slot-tac: tactic
  (Move slot to first subgoal *)
  val prefer-slot-tac: tactic
  (Destruct slot *)
  val dest-slot-tac: tactic'
  (Check if goal state has slot *)
  val has-slot: thm => bool
  (Defer subgoal to slot *)
  val to-slot-tac: tactic'
  (Print slot constraints *)
  val print-slot-tac: Proof.context => tactic

  (Focus on goals in slot *)
  val focus: tactic
  (Unfocus goals in slot *)
  val unfocus: tactic
  (Unfocus goals, and insert them as first subgoals *)
  val unfocus-ins:tactic

44
(* Focus on some goals in slot *)
val cond-focus: (term -> bool) -> tactic

(* Move some goals to slot *)
val some-to-slot-tac: (term -> bool) -> tactic

(******** Constraints *)
(* Check if subgoal is a constraint. To be used with COND' *)
val is-constraint-goal: term -> bool
(* Identity on constraint subgoal, no-tac otherwise *)
val is-constraint-tac: tactic'
(* Defer constraint to slot *)
val slot-constraint-tac: int -> tactic

(******** Constraint solving *)
val add-constraint-rule: thm -> Context.generic -> Context.generic
val del-constraint-rule: thm -> Context.generic -> Context.generic
val get-constraint-rules: Proof.context -> thm list
val del-safe-constraint-rule: thm -> Context.generic -> Context.generic
val get-safe-constraint-rules: Proof.context -> thm list

(* Solve constraint subgoal *)
val solve-constraint-tac: Proof.context -> tactic'
(* Solve constraint subgoal if solvable, fail if definitely unsolvable, apply simplification and unique rules otherwise. *)
val safe-constraint-tac: Proof.context -> tactic'

(* CONSTRAINT tag on goal is optional *)
val solve-constraint'-tac: Proof.context -> tactic'
(* CONSTRAINT tag on goal is optional *)
val safe-constraint'-tac: Proof.context -> tactic'

(* Solve, or apply safe-rules and defer to constraint slot *)
val constraint-tac: Proof.context -> tactic'

(* Apply safe rules to all constraint goals in slot *)
val process-constraint-slot: Proof.context -> tactic

(* Solve all constraint goals in slot, insert unsolved ones as first subgoals *)
val solve-constraint-slot: Proof.context -> tactic

val setup: theory -> theory

end
structure Sepref-Constraints: SEPREF-CONSTRAINTS = struct
  fun is-slot-goal @{mpat CONSTRAINT-SLOT -} = true | is-slot-goal - = false

  fun slot-goal-num st = let
    val i = find-index is-slot-goal (Thm.prems-of st) + 1
  in
    i
  end

  fun has-slot st = slot-goal-num st > 0

  fun WITH-SLOT tac st = let
    val si = slot-goal-num st
  in
    if si > 0 then tac si st else (warning Constraints: No slot; Seq.empty)
  end

  val to-slot-tac = IF-EXGOAL (fn i => WITH-SLOT (fn si =>
    if i < si then
      prefer-tac si THEN prefer-tac (i + 1)
      THEN (PRIMITIVE (fn st => Drule.comp-no-flatten (st, 0) 1 @{thm insert-slot-rl1})
        ORELSE PRIMITIVE (fn st => Drule.comp-no-flatten (st, 0) 1 @{thm insert-slot-rl2}))
    )
      THEN defer-tac 1
    else no-tac))

  val create-slot-tac =
    COND (has-slot) no-tac
    (PRIMITIVE (Thm.implies-intr @{cterms CONSTRAINT-SLOT (Trueprop True)})
      THEN defer-tac 1)

  val ensure-slot-tac = TRY create-slot-tac

  val prefer-slot-tac = WITH-SLOT prefer-tac

  val dest-slot-tac = SELECT-GOAL (ALLGOALS (CONVERSION (Conv.rewr-conv @{thm CONSTRAINT-SLOT-def})
    THEN' Goal.conjunction-tac
    THEN' TRY o resolve0-tac @{thms TrueI})
      THEN distinct-subgoals-tac)
val remove-slot-tac = WITH-SLOT (resolve0-tac @\{thms remove-slot\})

val focus = WITH-SLOT (fn i =>
  PRIMITIVE (Goal.restrict i 1)
  THEN ALLGOALS dest-slot-tac
  THEN create-slot-tac)

val unfocus-ins =
  PRIMITIVE (Goal.unrestrict 1)
  THEN WITH-SLOT defer-tac

fun some-to-slot-tac cond = (ALLGOALS (COND' (fn t => is-slot-goal t orelse not (cond t)) ORELSE' to-slot-tac))

val unfocus =
  some-to-slot-tac (K true)
  THEN unfocus-ins

fun cond-focus cond =
  focus
  THEN some-to-slot-tac (not o cond)

fun ON-SLOT tac = focus THEN tac THEN unfocus

fun print-slot-tac ctxt = ON-SLOT (print-tac ctxt SLOT:)

local
  (*fun prepare-constraint-conv ctxt = let
   open Conv
   fun CONSTRAINT-conv ct = case Thm.term-of ct of
     @\{mpat Trueprop (- -)} =>
       HOLogic.Trueprop-conv
        (rewr-conv @\{thm CONSTRAINT-def\[symmetric\]} ct
          | - => raise CTERM (CONSTRAINT-conv, [ct])
   in
     rec-conv ctxt ct
   end*)

  fun rec-conv ctxt ct = (CONSTRAINT-conv
    else-conv
    implies-conv (rec-conv ctxt) (rec-conv ctxt)
    else-conv
    forall-conv (rec-conv o #2) ctxt
  ) ct
in
  rec-conv ctxt
end*)

fun unfold-abbrevs ctxt =
Local-Defs.unfold0 ctxt {
@{thms split-constraint-rls CONSTRAINT-def}
@ Named-Theorems.get ctxt @{named-theorems constraint-abbrevs}
@ Named-Theorems.get ctxt @{named-theorems constraint-simps})
#> Conjunction.elim-conjunctions

fun check-constraint-rl thm = let
  fun ck (t as @{mpat Trueprop (?C -)}) = in
    ck (Thm.prop-of thm); thm end

  if is_Var (Term.head_of C) then raise TERM (Schematic head in constraint rule,[t,Thm.prop-of thm])
  else ()
  | ck @{mpat \_. PROP ?t} = ck t
  | ck @{mpat PROP ?s = PROP ?t} = (ck s; ck t)
  | ck t = raise TERM (Invalid part of constraint rule,[t,Thm.prop-of thm])
  in
   unfold-abbrevs ctxt #> map (check-constraint-rl o check-unsafe-constraint-rl)
  end

  in
structure constraint-rules = Named-Sorted-Thms (val name = @{binding constraint-rules}
val description = Constraint rules
val sort = K I
fun transform context = let
  open Conv
  val ctxt = Context.proof_of context
  in
    unfold-abbrevs ctxt #> map (check-constraint-rl o check-unsafe-constraint-rl)
  end)

structure safe-constraint-rules = Named-Sorted-Thms (val name = @{binding safe-constraint-rules}
val description = Safe Constraint rules
val sort = K I
fun transform context = let
  open Conv
  val ctxt = Context.proof_of context
  in
    unfold-abbrevs ctxt #> map check-constraint-rl
val add-constraint-rule = constraint-rules.add-thm
val del-constraint-rule = constraint-rules.del-thm
val get-constraint-rules = constraint-rules.get

val add-safe-constraint-rule = safe-constraint-rules.add-thm
val del-safe-constraint-rule = safe-constraint-rules.del-thm
val get-safe-constraint-rules = safe-constraint-rules.get

fun is-constraint-goal t = case Logic.strip-assums-concl t of
  @{mpat Trueprop (CONSTRAINT - -)} => true
| - => false

val is-constraint-tac = COND' is-constraint-goal

fun is-slottable-constraint-goal t = case Logic.strip-assums-concl t of
  @{mpat Trueprop (CONSTRAINT (CN-FALSE - -))} => false
| @{mpat Trueprop (CONSTRAINT - -)} => true
| - => false

val slot-constraint-tac = COND' is-slottable-constraint-goal THEN to-slot-tac

datatype 'a seq-cases = SC-NONE | SC-SINGLE of 'a Seq.seq | SC-MULTIPLE
  of 'a Seq.seq

fan seq-cases seq =
  case Seq.pull seq of
    NONE => SC-NONE
  | SOME (st1,seq) =>
      case Seq.pull seq of
        NONE => SC-SINGLE (Seq.single st1)
      | SOME (st2,seq) => SC-MULTIPLE (Seq.cons st1 (Seq.cons st2 seq))

fun SEQ-CASES tac (single-tac, multiple-tac) st = let
  val res = tac st
in
  case seq-cases res of
    SC-NONE => Seq.empty
  | SC-SINGLE res => Seq.maps single-tac res
  | SC-MULTIPLE res => Seq.maps multiple-tac res
end

fan SAFE tac = SEQ-CASES tac (all-tac, no-tac)
fan SAFE' tac = SAFE o tac

local
fun simp-constraints-tac ctxt = let
  val ctxt = put-simpset HOL-basic-ss ctxt
  addsimps (Named-Theorems.get ctxt @ {named-theorems constraint-simps})
in
  simp-tac ctxt
end

fun unfold-abbrevs-tac ctxt = let
  val ctxt = put-simpset HOL-basic-ss ctxt
  addsimps (Named-Theorems.get ctxt @ {named-theorems constraint-abbrevs})
  val ethms = @ {thms conjE}
  val ithms = @ {thms conjI}
in
  full-simp-tac ctxt
  THEN-ALL-NEW TRY o REPEAT-ALL-NEW (ematch-tac ctxt ethms)
  THEN-ALL-NEW TRY o REPEAT-ALL-NEW (match-tac ctxt ithms)
end

fun WITH-RULE-NETS tac ctxt = let
  val scn-net = safe-constraint-rules.get ctxt |> Tactic.build-net
  val cn-net = constraint-rules.get ctxt |> Tactic.build-net
in
  tac (scn-net, cn-net) ctxt
end

fun wrap-tac step-tac ctxt = REPEAT-ALL-NEW (simp-constraints-tac ctxt
  THEN-ALL-NEW unfold-abbrevs-tac ctxt
  THEN-ALL-NEW step-tac ctxt)
)

fun solve-step-tac (scn-net, cn-net) ctxt = REPEAT-ALL-NEW (DETERM o resolve-from-net-tac ctxt scn-net
  ORELSE' resolve-from-net-tac ctxt cn-net)
)

fun safe-step-tac (scn-net, cn-net) ctxt = REPEAT-ALL-NEW (DETERM o resolve-from-net-tac ctxt scn-net
  ORELSE' SAFE' (resolve-from-net-tac ctxt cn-net)
)

fun solve-tac cn-nets ctxt = SOLVED' (wrap-tac (solve-step-tac cn-nets) ctxt)
fun safe-tac cn-nets ctxt = simp-constraints-tac ctxt
  THEN-ALL-NEW unfold-abbrevs-tac ctxt
  THEN-ALL-NEW (safe-step-tac cn-nets ORELSE' TRY o wrap-tac (safe-step-tac cn-nets) ctxt)
in
val solve-constraint-tac = TRADE (fn ctxt =>
  is-constraint-tac
  THEN' resolve-tac ctxt @\{thms CONSTRAINT-I\}
  THEN' WITH-RULE-NETS solve-tac ctxt)

val safe-constraint-tac = TRADE (fn ctxt =>
  is-constraint-tac
  THEN' resolve-tac ctxt @\{thms CONSTRAINT-I\}
  THEN' WITH-RULE-NETS safe-tac ctxt
  THEN-ALL-NEW fo-resolve-tac @\{thms CONSTRAINT-D\} ctxt) (* TODO/FIXME: fo-resolve-tac has non-canonical parameter order *)

val solve-constraint'-tac = TRADE (fn ctxt =>
  TRY o resolve-tac ctxt @\{thms CONSTRAINT-I\}
  THEN' WITH-RULE-NETS solve-tac ctxt)

val safe-constraint'-tac = TRADE (fn ctxt =>
  TRY o resolve-tac ctxt @\{thms CONSTRAINT-I\}
  THEN' WITH-RULE-NETS safe-tac ctxt)

end

fun constraint-tac ctxt =
  safe-constraint-tac ctxt THEN-ALL-NEW slot-constraint-tac

fun process-constraint-slot ctxt = ON-SLOT (ALLGOALS (TRY o safe-constraint-tac ctxt))

fun solve-constraint-slot ctxt =
  cond-focus is-constraint-goal
  THEN ALLGOALS (
    COND' is-slot-goal
    ORELSE' (solve-constraint-tac ctxt
      ORELSE' TRY o safe-constraint-tac ctxt
    )
  )
  THEN unfocus-ins

val setup = I
  #> constraint-rules.setup
  #> safe-constraint-rules.setup

end

setup Sepref-Constraints.setup
method-setup print-slot = ⟨Scan.succeed (fn ctxt => SIMPLE-METHOD (Sepref-Constraints.print-slot-tac ctxt))⟩;

method-setup solve-constraint = ⟨Scan.succeed (fn ctxt => SIMPLE-METHOD′
(Sepref-Constraints.solve-constraint′-tac ctxt))⟩
method-setup safe-constraint = ⟨Scan.succeed (fn ctxt => SIMPLE-METHOD′
(Sepref-Constraints.safe-constraint′-tac ctxt))⟩

end

1.4 Frame Inference

theory Sepref-Frame
imports Sepref-Basic Sepref-Constraints
begin

In this theory, we provide a specific frame inference tactic for Sepref. The first tactic, frame-tac, is a standard frame inference tactic, based on the assumption that only hn-ctxt-assertions need to be matched. The second tactic, merge-tac, resolves entailments of the form \( F1 \lor A \ F2 \Rightarrow t \) that occur during translation of if and case statements. It synthesizes a new frame \(?F\), where refinements of variables with equal refinements in \( F1 \) and \( F2 \) are preserved, and the others are set to hn-invalid.

definition mismatch-assn :: (\'?a \Rightarrow \'?c \Rightarrow assn) \Rightarrow (\'?a \Rightarrow \'?c \Rightarrow assn) \Rightarrow \'?a \Rightarrow \'?c \Rightarrow assn
  where mismatch-assn R1 R2 x y ≡ R1 x y \lor A R2 x y

abbreviation hn-mismatch R1 R2 ≡ hn-ctxt (mismatch-assn R1 R2)

lemma recover-pure-aux: CONSTRAINT is-pure R \Rightarrow hn-invalid R x y \Rightarrow t
  hn-ctxt R x y
  by (auto simp: is-pure-conv invalid-pure-recover hn-ctxt-def)

lemma frame-thms:
  \( P \Rightarrow t \ P \)
  \( P \Rightarrow t \ P' \Rightarrow F \Rightarrow t \ F* \Rightarrow t \ F'*P \Rightarrow t \ F'*P' \)
  \( hn-ctxt R x y \Rightarrow t \ hn-invalid R x y \)
  \( hn-ctxt R x y \Rightarrow t \ hn-ctxt (\lambda- \cdot true) x y \)
  CONSTRAINT is-pure R \Rightarrow hn-invalid R x y \Rightarrow t \ hn-ctxt R x y
  apply −
  applyS simp
  applyS (rule entt-star-mono; assumption)
  subgoal
apply (simp add: hn-ctxt-def)
apply (rule entI)
apply (rule ent-trans[OF invalidate[of R]])
by solve-entails
applyS (sep-auto simp: hn-ctxt-def)
applyS (erule recover-pure-aux)
done

named-theorems-rev sepref-frame-match-rules (Sepref: Additional frame rules)

Rules to discharge unmatched stuff

lemma frame-rem1: \( P \implies P \) by simp

lemma frame-rem2: \( F \implies F \implies F \ast \operatorname{hn-ctx} t \quad A \times y \implies F' \ast \operatorname{hn-ctx} A \ast y \)
apply (rule entt-star-mono) by auto

lemma frame-rem3: \( F \implies F' \implies F \ast \operatorname{hn-ctx} t \quad A \times y \implies F' \)
using frame-thms(2) by fastforce

lemma frame-rem4: \( P \implies P \implies \top \)

lemma frame-rem-thms = frame-rem1 frame-rem2 frame-rem3 frame-rem4

named-theorems-rev sepref-frame-rem-rules
(Sepref: Additional rules to resolve remainder of frame - pairing)

lemma ent-disj-star-mono:
\[ \begin{array}{c}
A \lor A \implies A \lor E ; B \lor A \implies B \lor D \implies A \ast B \lor A \ast D \implies A \ast E \lor F \\
\end{array} \]
by (metis ent-disjI1 ent-disjI2 ent-disjE ent-star-mono)

lemma ent-disj-star-mono:
\[ \begin{array}{c}
A \lor A \implies A \lor E ; B \lor A \implies B \lor D \implies A \ast B \lor A \ast D \implies A \ast E \lor F \\
\end{array} \]

proof -
assume a1: \( A \lor A \implies A \lor E \)
assume B \( \lor A \implies B \lor F \)
then have A \( \ast B \lor A \ast D \implies A \ast E \ast (\text{true} \ast F) \)
using a1 by (simp add: ent-disj-star-mono enttD)
then show ?thesis
by (metis (no-types) assn-times-comm entI merge-true-star-ctx star-aci(3))
qed

lemma hn-merge1:
\[ F \lor A \implies F \lor A \]
\[ \begin{array}{c}
\operatorname{hn-ctx} R1 \times x' \lor A \implies \operatorname{hn-ctx} R2 \times x' \implies A \lor \operatorname{hn-ctx} R \times x' ; F \lor A \implies F \lor A \lor F' \implies \operatorname{hn-ctx} R \times x' \implies F \ast \operatorname{hn-ctx} R \times x' \\
\end{array} \]
apply simp
by (rule entt-disj-star-mono; simp)

lemma hn-merge2:
  hn-invalid R x x' \lor_A hn-ctxt R x x' \Longrightarrow t \ hn-invalid R x x'
  hn-ctxt R x x' \lor_A hn-invalid R x x' \Longrightarrow t \ hn-invalid R x x'
by (sep-auto eintr: invalidate ent-disjE intro!: ent-imp-ent simp: hn-ctxt-def)+

lemma invalid-assn-mono: hn-ctxt A x y \Longrightarrow t \ hn-invalid A x y
  \Longrightarrow hn-invalid A x y \Longrightarrow t \ hn-invalid B x y
by (clarsimp simp: invalid-assn-def entailst-def entails-def hn-ctxt-def)
  (force simp: mod-star-conv)

lemma hn-merge3:
  \[ \[ NO-MATCH (hn-invalid XX) \ R2; \ hn-ctxt R1 x x' \lor_A \ hn-ctxt R2 x x' \Longrightarrow t \\
  \ hntxt Rm x x' \Longrightarrow \ hn-invalid R1 x x' \lor_A \ hn-ctxt R2 x x' \Longrightarrow t \ hn-invalid Rm x x' \] \] 
  \[ \[ NO-MATCH (hn-invalid XX) \ R1; \ hn-ctxt R1 x x' \lor_A \ hn-ctxt R2 x x' \Longrightarrow t \\
  \ hntxt Rm x x' \Longrightarrow \ hn-ctxt R1 x x' \lor_A \ hn-invalid R2 x x' \Longrightarrow t \ hn-invalid Rm x x' \] \]
  apply (meson entt-disjD1 entt-disjD2 entt-disjE entt-trans frame-thms(3) invalid-assn-mono)
apply (meson entt-disjD1 entt-disjD2 entt-disjE entt-trans frame-thms(3) invalid-assn-mono)
done

lemmas merge-thms = hn-merge1 hn-merge2

named-theorems sepref-frame-merge-rules (Sepref: Additional merge rules)

lemma hn-merge-mismatch: hn-ctxt R1 x x' \lor_A hn-ctxt R2 x x' \Longrightarrow t \ hn-mismatch
R1 R2 x x'
  by (sep-auto simp: hn-ctxt-def mismatch-assn-def)

lemma is-merge: P1 \lor_A P2 \Longrightarrow t \ P \Longrightarrow t \ P1 \lor_A P2 \Longrightarrow t \ P.

lemma merge-mono: [A \Longrightarrow t A'; B \Longrightarrow t B'; A \lor_A B \Longrightarrow t C] \Longrightarrow A \lor_A B \Longrightarrow t C
by (meson entt-disjE entt-disj1-direct entt-disj2-direct entt-trans)

Apply forward rule on left or right side of merge

lemma gen-merge-cons1: [A \Longrightarrow t A'; A \lor_A B \Longrightarrow t C] \Longrightarrow A \lor_A B \Longrightarrow t C
by (meson merge-mono entt-refl)

lemma gen-merge-cons2: [B \Longrightarrow t B'; A \lor_A B' \Longrightarrow t C] \Longrightarrow A \lor_A B \Longrightarrow t C
by (meson merge-mono entt-refl)

lemmas gen-merge-cons = gen-merge-cons1 gen-merge-cons2

These rules are applied to recover pure values that have been destroyed by
rule application

**definition** \( \text{RECOVER-PURE} \) \( P \ Q \equiv P \rightarrow_\tau Q \)

**lemma** \( \text{recover-pure} \):
\[
\text{RECOVER-PURE} \ emp \ emp
\]
\[
\equiv \text{RECOVER-PURE} \ (P1 \ast P2) \ (Q1 \ast Q2)
\]
\[
\text{CONSTRAINT} \ is-pure \ R \implies \text{RECOVER-PURE} \ (\text{hn-invalid} \ R \ x \ y) \ (\text{hn-ctxt} \ R \ x \ y)
\]
\[
\text{unfolding} \ \text{RECOVER-PURE-def}
\]
\[
\text{subgoal by sep-auto}
\]
\[
\text{subgoal by (drule 1) entt-star-mono}
\]
\[
\text{subgoal by (rule recover-pure-aux)}
\]
\[
\text{subgoal by sep-auto}
\]
\[
\text{done}
\]

**lemma** \( \text{recover-pure-triv} \):
\[
\text{RECOVER-PURE} \ P \ P
\]
\[
\text{unfolding} \ \text{RECOVER-PURE-def by sep-auto}
\]

Weakening the postcondition by converting invalid-assn to \( \lambda \)-
true

**definition** \( \text{WEAKEN-HNR-POST} \) \( \Gamma \ \Gamma' \ \Gamma'' \equiv (\exists h. h \models \Gamma) \implies (\Gamma'' \implies_\tau \Gamma') \)

**lemma** \( \text{weaken-hnr-postI} \):
\[
\text{assumes} \ \text{WEAKEN-HNR-POST} \ \Gamma \ \Gamma' \ \Gamma''
\]
\[
\text{assumes} \ \text{hn-refine} \ \Gamma \ c \ \Gamma' \ R \ a
\]
\[
\text{shows} \ \text{hn-refine} \ c \ \Gamma'' \ R \ a
\]
\[
\text{apply (rule hn-refine-preI)}
\]
\[
\text{apply (rule hn-refine-cons-post)}
\]
\[
\text{apply (rule assms)}
\]
\[
\text{using assms} (1) \ \text{unfolding} \ \text{WEAKEN-HNR-POST-def by blast}
\]

**lemma** \( \text{weaken-hnr-post-triv} \): \( \text{WEAKEN-HNR-POST} \ \Gamma \ P \ P \)
\[
\text{unfolding} \ \text{WEAKEN-HNR-POST-def by sep-auto}
\]

**lemma** \( \text{weaken-hnr-post} \):
\[
[\text{WEAKEN-HNR-POST} \ \Gamma \ P \ P'; \ \text{WEAKEN-HNR-POST} \ \Gamma' \ Q \ Q'] \implies \text{WEAKEN-HNR-POST} \ (\Gamma \ast \Gamma') \ (P \ast Q) \ (P' \ast Q')
\]
\[
\text{WEAKEN-HNR-POST} \ (\text{hn-ctxt} \ R \ x \ y) \ (\text{hn-ctxt} \ R \ x \ y)
\]
\[
\text{WEAKEN-HNR-POST} \ (\text{hn-ctxt} \ R \ x \ y) \ (\text{hn-invalid} \ R \ x \ y)
\]
\[
\text{proof (goal-cases)}
\]
\[
\text{case 1 thus ?case}
\]
\[
\text{unfolding} \ \text{WEAKEN-HNR-POST-def}
\]
\[
\text{apply clarsimp}
\]
\[
\text{apply (rule entt-star-mono)}
\]
\[
\text{by (auto simp: mod-star-conv)}
\]
next
case 2 thus ?case by (rule weaken-hnr-post-triv)
next
case 3 thus ?case
    unfolding WEAKEN-HNR-POST-def
    by (sep-auto simp: invalid-assn-def hn-ctxt-def)
qed

lemma reorder-enttI:
  assumes A*true = C*true
  assumes B*true = D*true
  shows (A⇒t B) ≡ (C⇒t D)
  apply (intro eq-reflection)
  unfolding entt-def-true
  by (simp add: assms)

lemma merge-sat1: (A∨A′ Am⇒t Am) ⇒ (A∨Am⇒t Am)
  using entt-disjD1 entt-disjE by blast
lemma merge-sat2: (A∨A′ Am⇒t Am) ⇒ (Am∨A′ Am⇒t Am)
  using entt-disjD2 entt-disjE by blast

ML ⟨⟨
signature SEPREFIX-FRAME = sig

(* Check if subgoal is a frame obligation *)
(val is-frame : term -> bool *)
(* Check if subgoal is a merge obligation *)
(val is-merge: term -> bool)
(* Perform frame inference *)
(val frame-tac: (Proof.context -> tactic') -> Proof.context -> tactic')
(* Perform merging *)
(val merge-tac: (Proof.context -> tactic') -> Proof.context -> tactic')

(val frame-step-tac: (Proof.context -> tactic') -> bool -> Proof.context -> tactic')

(* Reorder frame *)
(val prepare-frame-tac : Proof.context -> tactic')
(* Solve a RECOVER-PURE goal, inserting constraints as necessary *)
(val recover-pure-tac: Proof.context -> tactic')
(* Split precondition of hnr-goal into frame and arguments *)
val align-goal-tac: Proof.context -> tactic'

(* Normalize goal’s precondition *)
val norm-goal-pre-tac: Proof.context -> tactic'

(* Rearrange precondition of hnr-term according to parameter order, normalize all relations *)
val align-rl-conv: Proof.context -> conv

(* Convert hn-invalid to λ- true in postcondition of hnr-goal. Makes proving the goal easier.*)
val weaken-post-tac: Proof.context -> tactic'

val add-normrel-eq : thm -> Context.generic -> Context.generic
val del-normrel-eq : thm -> Context.generic -> Context.generic
val get-normrel-eqs : Proof.context -> thm list

val cfg-debug: bool Config.T
val setup: theory -> theory
end

structure Sepref-Frame : SEPREF-FRAME = struct
  val cfg-debug = 
    Attrib.setup-config-bool @{binding sepref-debug-frame} (K false)
  val DCONVERSION = Sepref-Debugging DBG-CONVERSION cfg-debug
  val dbg-msg-tac = Sepref-Debugging dbg-msg-tac cfg-debug

structure normrel-eqs = Named-Thms ( 
  val name = @{binding sepref-frame-normrel-eqs}
  val description = Equations to normalize relations for frame matching
 )

val add-normrel-eq = normrel-eqs.add-thm
val del-normrel-eq = normrel-eqs.del-thm
val get-normrel-eqs = normrel-eqs.get

val mk-entailst = HOLogic.mkbinrel @{const-name entailst}

local
  open Sepref-Basic Refine-Util Cone
  fun assn-ord p = case apply2 dest-hn-ctxt-opt p of
    (NONE,NONE) => EQUAL
in
fan reorder-ctxt-conv ctxt ct = let
  val cert = Thm.cterm-of ctxt

  val new-ct = Thm.term-of ct
    |> strip-star
    |> sort assn-ord
    |> list-star
    |> cert

  val thm = Goal.prove-internal ctxt [] (mk-cequals (ct, new-ct))
    (fn - => simp-tac
    (put-simpset HOL-basic-ss ctxt addsimps @{thms star-aci}) 1)
  in thm end

fun prepare-fi-conv ctxt ct = case Thm.term-of ct of
  @{mpat ?P \Rightarrow \ ?Q} => let
    val cert = Thm.cterm-of ctxt

    val (Qm, Qum) = strip-star Q
      |> filter-out is-true
      |> List.partition is-hn-ctxt

    val Qtab = (
      Qm
      |> map (fn x => (\#2 (dest-hn-ctxt x),(NONE,x)))
      |> Termtab.make
    ) handle e as (Termtab.DUP -) => (}
    tracing (Dupa:heap: ^{\make-string} ct); raise e)

  (* Go over entries in P and try to find a partner *)
  val (Qtab,Pum) = fold (fn a => fn (Qtab,Pum) =>
    case dest-hn-ctxt-opt a of
      NONE => (Qtab,a::Pum)
    | SOME (-,p,_) => ( case Termtab.lookup Qtab p of
      SOME (NONE,tg) => (Termtab.update (p,(SOME a,tg)) Qtab, Pum)
    | - => (Qtab,a::Pum))
  ) (strip-star P) (Qtab,[])

  val Pum = filter-out is-true Pum

  (* Read out information from Qtab *)
val (pairs, Qum2) = TermTab.dest Qtab |> map #2
|> List.partition (is-some o #1)
|> apfst (map (apfst the))
|> apsnd (map #2)

(* Build reordered terms: P' = fst pairs * Pum, Q' = snd pairs *
 (Qum2 * Qum) *)
val P' = mk-star (list-star (map fst pairs), list-star Pum)
val Q' = mk-star (list-star (map snd pairs), list-star (Qum2 @ Qum))

val new-ct = mk-entailst (P', Q') |> cert

val msg-tac = dbg-msg-tac (Sepref-Debugging.msg-allgoals Solving frame permutation) ctxt 1
val tac = msg-tac THEN ALLGOALS (resolve-tac ctxt @ {thms reorder-enttI})
THEN star-permute-tac ctxt

val thm = Goal.prove-internal ctxt [] (mk-cequals (ct, new-ct)) (fn - => tac)
in thm end
| - => no-conv ct end

fun is-merge @ {mpat Trueprop (- \ A - \ \ \ t \ \ \ t)} = true | is-merge - = false
fun is-gen-frame @ {mpat Trueprop (- \ \ \ \ t \ \ \ t)} = true | is-gen-frame - = false

fun prepare-frame-tac ctxt = let
  open Refine.Util Conv
  val frame-ss = put-simpset HOL-basic-ss ctxt addsimps
    @{thms mult-1-right \ where 'a = assn} mult-1-left \ where 'a = assn\}
in CONVERSION Thm.eta-conversion THEN' (*CONCL-COND' is-frame THEN'*)
simp-tac frame-ss THEN'
CONVERSION (HOL-concl-conv (fn - => prepare-fi-conv ctxt) ctxt)
end

local
  fun wrap-side-tac side-tac dbg tac = tac THEN-ALL-NEW-FWD (CONCL-COND' is-gen-frame
    ORELSE' (if dbg then TRY-SOLVED' else SOLVED') side-tac
  )
in
  fun frame-step-tac side-tac dbg ctxt = let

59
open Refine-Util Conv

(* Constraint solving is built-in *)
val side-tac = Sepref-Constraints.constraint-tac ctxt ORELSE' side-tac ctxt

val frame-thms = @{thms frame-thms} @
Named-Theorems-Rev.get ctxt @{named-theorems-rev sepref-frame-match-rules}

val merge-thms = @{thms merge-thms} @
Named-Theorems.get ctxt @{named-theorems sepref-frame-merge-rules}
val ss = put-simpset HOL-basic-ss ctxt add_simps normrel-eqs get ctxt
fun frame-thm-tac dbg = wrap-side-tac side-tac dbg (resolve-tac ctxt frame-thms)
fun merge-thm-tac dbg = wrap-side-tac side-tac dbg (resolve-tac ctxt merge-thms)

fun thm-tac dbg = CONCL-COND' is-merge THEN-ELSE' (merge-thm-tac
dbg, frame-thm-tac dbg)
  in
  full-simp-tac ss THEN' thm-tac dbg
end

fun frame-loop-tac side-tac ctxt = let
  in
   TRY o (  
     REPEAT-ALL-NEW (DETERM o frame-step-tac side-tac false ctxt)  
   )
  end

fun frame-tac side-tac ctxt = let
  open Refine-Util Conv
  val frame-rem-thms = @{thms frame-rem-thms}
  @
Named-Theorems-Rev.get ctxt @{named-theorems-rev sepref-frame-rem-rules}
val solve-remainder-tac = TRY o REPEAT-ALL-NEW (DETERM o resolve-tac
cxt frame-rem-thms)
  in
   (prepare-frame-tac ctxt
     THEN' resolve-tac ctxt @{thms ent-star-mono entt-star-mono})
THEN-ALL-NEW-LIST [  
  frame-loop-tac side-tac ctxt,
  solve-remainder-tac
 ]
end

fun merge-tac side-tac ctxt = let
  open Refine-Util Conv
  val merge-conv = arg1-conv (binop-conv (reorder-ctxt-conv ctxt))
  in
CONVERSION Thm.eta-conversion THEN'
CONCL-COND' is-merge THEN'
simp-tac (put-simpset HOL-basic-ss ctxt addsimps @{thms star-aci}) THEN'
CONVERSION (HOL-concl-conv (fn - => merge-conv) ctxt) THEN'
frame-loop-tac side-tac ctxt
end

val setup = normrel-eqs.setup

local
  open Sepref-Basic
  fun is-invalid @{mpat hn-invalid - - :: assn} = true | is-invalid - = false
  fun contains-invalid @{mpat Trueprop (RECOVER-PURE ?Q -)} = exists is-invalid (strip-star Q)
    | contains-invalid - = false
in
  fun recover-pure-tac ctxt =
    CONCL-COND' contains-invalid THEN-ELSE'
    (REPEAT-ALL-NEW (DETERM o (resolve-tac ctxt @{thms recover-pure} ORELSE' Sepref-Constraints.constraint-tac ctxt)),
     resolve-tac ctxt @{thms recover-pure-triv})
  )
end

local
  open Sepref-Basic Refine-Util
datatype cte = Other of term | Hn of term * term * term
  fun dest-ctxt-elem @{mpat hn-ctxt ?R ?a ?c} = Hn (R, a, c)
    | dest-ctxt-elem t = Other t
  fun mk-ctxt-elem (Other t) = t
    | mk-ctxt-elem (Hn (R, a, c)) = @{mk-term hn-ctxt ?R ?a ?c}
  fun match x (Hn (-, y, -)) = x aconv y
    | match - - = false
  fun dest-with-frame (*ctxt*) - t = let
    val (P, c, Q, R, a) = dest-hn-refine t
    val (args) = dest-hnr-absfun a
    val pre-ctes = strip-star P |> map dest-ctxt-elem
    val (pre-args, frame) = (case split-matching match args pre-ctes of
      NONE => raise TERM (align-conv: Could not match all arguments,[P, a])
      | SOME x => x)
in
fun align-goal-conv-aux ctxt t = let
val ((frame,pre-args),c,Q,R,a) = dest-with-frame ctxt t
val P' = apply2 (list-star o map mk-ctxt-elem) (frame,pre-args) |> mk-star
val t' = mk-hn-refine (P',c,Q,R,a)
in t' end

fun align-rl-conv-aux ctxt t = let
val ((frame,pre-args),c,Q,R,a) = dest-with-frame ctxt t
val frame = [] orelse raise TERM (align-rl-conv: Extra preconditions in rule,[l,list-star (map mk-ctxt-elem elem)]))
val P' = list-star (map mk-ctxt-elem elem pre-args)
val t' = mk-hn-refine (P',c,Q,R,a)
in t' end

fun normrel-conv ctxt = let
val ss = put-simpset HOL-basic-ss ctxt addsimps normrel-eqs.get ctxt
in
Simplifier.rewrite ss
end

fun align-goal-conv ctxt = f-tac-conv ctxt (align-goal-conv-aux ctxt) (star-permute-tac ctxt)

fun norm-goal-pre-conv ctxt = let
open Conv
val nr-conv = normrel-conv ctxt
in
HOL-concl-conv (fn - => hn-refine-conv nr-conv all-conv all-conv all-conv all-conv) ctxt
end

fun norm-goal-pre-tac ctxt = CONVERSION (norm-goal-pre-conv ctxt)

fun align-rl-conv ctxt = let
open Conv
val nr-conv = normrel-conv ctxt
in
HOL-concl-conv (fn ctxt => f-tac-conv ctxt (align-rl-conv-aux ctxt) (star-permute-tac ctxt)) ctxt
then-conv HOL-concl-conv (K (hn-refine-conv nr-conv all-conv nr-conv nr-conv all-conv)) ctxt
end
fun align-goal-tac ctxt =  
CONCL-COND' is-hn-refine-concl  
THEN' DCONVERSION ctxt (HOL-concl-conv align-goal-conv ctxt)  
end

fun weaken-post-tac ctxt =  
TRADE (fn ctxt =>  
resolve-tac ctxt @$\{\text{thms weaken-hnr-postI}\}$  
THEN' SOLVED' (REPEAT-ALL-NEW (DETERM o resolve-tac ctxt @$\{\text{thms weaken-hnr-post weaken-hnr-post-triv}\}$)))  
) ctxt  
end

setup Sepref-Frame.setup

method-setup weaken-hnr-post = (Scan.succeed (fn ctxt => SIMPLE-METHOD'  
(Sepref-Frame.weaken-post-tac ctxt))):  
(Convert hn-invalid to hn-ctxt ($\lambda$ -. true) in postcondition of hn-refine goal)

method extract-hnr-invalids = (  
rule hn-refine-preI,  
((drule mod-starD hn-invalidI | elim conjE exE)?)?  
) — Extract hn-invalid - - = true preconditions from hn-refine goal.

lemmas $[\text{sepref-frame-normrel-eqs} = \text{the-pure-pure pure-the-pure}$

end

1.5 Refinement Rule Management

theory Sepref-Rules  
imports Sepref-Basic Sepref-Constraints  
begin

This theory contains tools for managing the refinement rules used by Sepref

The theories are based on uncurried functions, i.e., every function has type 'a ⇒ 'b, where 'a is the tuple of parameters, or unit if there are none.

1.5.1 Assertion Interface Binding

Binding of interface types to refinement assertions
definition intf-of-assn :: ('a ⇒ - ⇒ assn) ⇒ 'b itself ⇒ bool where
[simp]: intf-of-assn a b = True

lemma intf-of-assnI: intf-of-assn R TYPE('a) by simp

named-theorems-rev intf-of-assn (Links between refinement assertions and interface types)

lemma intf-of-assn-fallback: intf-of-assn (R :: 'a ⇒ - ⇒ assn) TYPE('a) by simp

1.5.2 Function Refinement with Precondition

definition fref :: ('c ⇒ bool) ⇒ ('a × 'c) set ⇒ ('b × 'd) set
⇒ (('a ⇒ 'b) × ('c ⇒ 'd)) set
([|-] f - - [0,60,60] 60)
where |P|_f R → S ≡ \{(f,g). \forall x y. P y \land (x,y)\in R \rightarrow (f x, g y)\in S\}

abbreviation freft (- → f - [60,60] 60) where R → f S ≡ (\[λ. True\] f R → S)

lemma rel2p-fref[rel2p]: rel2p (fref P R S)
= (\lambda f g. (\forall x y. P y \rightarrow rel2p R x y \rightarrow rel2p S (f x) (g y)))
by (auto simp: fref-def rel2p-def[abs-def])

lemma fref-cons:
assumes (f,g) ∈ |P|_f R → S
assumes c a. (c,a)\in R’ ⇒ Q a ⇒ P a
assumes S’ ⊆ S
shows (f,g) ∈ |Q|_f R’ → S’
using assms
unfolding fref-def
by fastforce

lemmas fref-cons’ = fref-cons[OF - - order-refl order-refl]

lemma frefl[intro?]:
assumes \[\land x y. |P y; (x,y)\in R| \rightarrow (f x, g y)\in S\]
shows (f,g)\in fref P R S
using assms
unfolding fref-def
by auto

lemma fref-ncI: (f,g)\in R→S ⇒ (f,g)\in R→f S
apply (rule frefl)
apply parametricity
done

lemma frefD:
assumes \((f, g) \in \text{fref } P \ R \ S\)
shows \([P \ y; (x, y) \in R] \implies (f \ x, g \ y) \in S\)
using assms
unfolding \(\text{fref-def}\)
by auto

**lemma** \(\text{fref-ncD}: (f, g) \in R \to S \implies (f, g) \in R \to S\)
apply (rule fun-refI)
apply (drule \(\text{frefD}\))
apply simp
apply assumption+
done

**lemma** \(\text{fref-compI}:\)
\(\text{fref } P \ R_1 \ R_2 \ O \text{ fref } Q \ S_1 \ S_2 \subseteq\)
\(\text{fref} (\lambda x. \ Q \ x \land (\forall y. (y, x) \in S_1 \to P \ y)) (R_1 \ O \ S_1) (R_2 \ O \ S_2)\)
unfolding \(\text{fref-def}\)
apply (auto)
apply blast
done

**lemma** \(\text{fref-compI}':\)
\[ (f, g) \in \text{fref } P \ R_1 \ R_2; (g, h) \in \text{fref } Q \ S_1 \ S_2 \]
\(\implies (f, h) \in \text{fref} (\lambda x. \ Q \ x \land (\forall y. (y, x) \in S_1 \to P \ y)) (R_1 \ O \ S_1) (R_2 \ O \ S_2)\)
using \(\text{fref-compI[of } P \ R_1 \ R_2 \ Q \ S_1 \ S_2]\)
by auto

**lemma** \(\text{fref-unit-conv}:\)
\(\text{fref} (\lambda x. \ c, \lambda y. \ a) \in \text{fref } P \ unit-rel \ S \iff (P () \to (c, a) \in S)\)
by (auto simp: \(\text{fref-def}\))

**lemma** \(\text{fref-uncurry-conv}:\)
\(\text{fref} (\text{uncurry } c, \text{uncurry } a) \in \text{fref } (R_1 \times, R_2) S \)
\(\iff (\forall x_1 y_1 x_2 y_2. \ P (y_1, y_2) \to (x_1, y_1) \in R_1 \to (x_2, y_2) \in R_2 \to (c \ x_1 x_2, a \ y_1 y_2) \in S)\)
by (auto simp: \(\text{fref-def}\))

**lemma** \(\text{fref-mono}: \[ \forall x. P' x \implies P x; R' \subseteq R; S \subseteq S' \]\(\implies \text{fref} P \ R \ S \subseteq \text{fref} P' \ R' \ S'\)
unfolding \(\text{fref-def}\)
by auto blast

**lemma** \(\text{fref-composeI}:\)
assumes \(\text{FR1}: (f, g) \in \text{fref } P \ R_1 \ R_2\)
assumes \(\text{FR2}: (g, h) \in \text{fref } Q \ S_1 \ S_2\)
assumes \(\text{C1}: \forall x. P' x \implies Q x\)
assumes \(\text{C2}: \forall x \ y. [P' x; (y, x) \in S_1] \implies P \ y\)
assumes \(\text{R1}: R' \subseteq R_1 \ O \ S_1\)

65
assumes $R2 : R2 O S2 \subseteq S'$
assumes $FH : f' = f h' = h$
shows $(f', h') \in \text{ref} P' R' S'$

unfolding $FH$
apply (rule set-mp [\text{OF ref-mono ref-compI'[OF FR1 FR2]!]})
using $C1 C2$ apply blast
using $R1$ apply blast
using $R2$ apply blast
done

lemma $\text{fref-triv}: A \subseteq \text{Id} \implies (f, f) \in [P]_f A \rightarrow \text{Id}$
by (auto simp: $\text{fref-def}$)

1.5.3 Heap-Function Refinement

The following relates a heap-function with a pure function. It contains a precondition, a refinement assertion for the arguments before and after execution, and a refinement relation for the result.

definition $\text{hfref}$
::
\begin{align*}
  ('a \Rightarrow \text{bool}) \\
  \Rightarrow \ (('a \Rightarrow 'ai \Rightarrow \text{assn}) \times ('a \Rightarrow 'ai \Rightarrow \text{assn})) \\
  \Rightarrow \ (('b \Rightarrow 'bi \Rightarrow \text{assn}) \\
  \Rightarrow \ (('ai \Rightarrow 'bi \text{ Heap}) \times ('a \Rightarrow 'b \text{ nres})) \text{ set} \\
  ([c]_a - \rightarrow - [60, 60] 60)
\end{align*}

where

$[P]_a RS \rightarrow T \equiv \{ (f, g) . \forall c a . P a \rightarrow \text{hn-refine} (\text{fst RS} a c) (f c) (\text{snd RS} a c) T (g a)\}$

abbreviation $\text{hfref} (- \rightarrow_a - [60, 60] 60)$ where $RS \rightarrow_a T \equiv ([\lambda . \text{True}]_a RS \rightarrow T)$

lemma $\text{hfref}[\text{intro?}]$:
assumes $\forall c a . P a \implies \text{hn-refine} (\text{fst RS} a c) (f c) (\text{snd RS} a c) T (g a)$
shows $(f, g) \in \text{hfref} P RS T$
using assms unfolding $\text{hfref-def}$ by blast

lemma $\text{hfrefD}$:
assumes $(f, g) \in \text{hfref} P RS T$
shows $\forall c a . P a \implies \text{hn-refine} (\text{fst RS} a c) (f c) (\text{snd RS} a c) T (g a)$
using assms unfolding $\text{hfref-def}$ by blast

lemma $\text{hfref-to-ASSERT-conv}$:
$\text{NO-MATCH} (\lambda . \text{True}) P \longrightarrow (a, b) \in [P]_a R \rightarrow S \longleftrightarrow (a, \lambda x . \text{ASSERT} (P x))$

unfolding $\text{hfref-def}$
apply (clarsimp; safe; clarsimp?)
apply (rule hn-refine-nofailI)
apply (simp add: refine-pw-simps)
subgoal for $xc$ $xa$
apply (drule spec[of - $xc$])
apply (drule spec[of - $xa$])
by simp
done

A pair of argument refinement assertions can be created by the input assertion and the information whether the parameter is kept or destroyed by the function.

primrec hf-pres :: $(\alpha\Rightarrow\beta\Rightarrow\text{assn})\Rightarrow\text{bool}\Rightarrow(\alpha\Rightarrow\beta\Rightarrow\text{assn})\times(\alpha\Rightarrow\beta\Rightarrow\text{assn})$
where
$hf-pres\ \text{R\ True} = (\text{R},\text{R})$ | $hf-pres\ \text{R\ False} = (\text{R},\text{invalid-assn\ R})$

abbreviation $hfkeep$ :: $(\alpha\Rightarrow\beta\Rightarrow\text{assn})\Rightarrow(\alpha\Rightarrow\beta\Rightarrow\text{assn})\times(\alpha\Rightarrow\beta\Rightarrow\text{assn})$
where $R^k \equiv hf-pres\ \text{R\ True}$

abbreviation $hfdrop$ :: $(\alpha\Rightarrow\beta\Rightarrow\text{assn})\Rightarrow(\alpha\Rightarrow\beta\Rightarrow\text{assn})\times(\alpha\Rightarrow\beta\Rightarrow\text{assn})$
where $R^d \equiv hf-pres\ \text{R\ False}$

abbreviation $hn-kede\ R\ kd \equiv hn-ctxt\ (\text{snd}\ (hf-pres\ \text{R\ kd}))$
abbreviation $hn-keep\ R \equiv hn-kede\ \text{R\ True}$
abbreviation $hn-dest\ R \equiv hn-kede\ \text{R\ False}$

lemma $keep-drop-sels$[simp]:
\[
\text{fst}\ (R^k) = R
\]
\[
\text{snd}\ (R^k) = R
\]
\[
\text{fst}\ (R^d) = R
\]
\[
\text{snd}\ (R^d) = \text{invalid-assn\ R}
\]
by auto

lemma $hf-pres-fst$[simp]: $\text{fst}\ (hf-pres\ \text{R\ k}) = R$ by (cases $k$) auto

The following operator combines multiple argument assertion-pairs to argument assertion-pairs for the product. It is required to state argument assertion-pairs for uncurried functions.

definition $hfprod$ ::
\[
(\alpha\Rightarrow\beta\Rightarrow\text{assn})\times(\alpha\Rightarrow\beta\Rightarrow\text{assn})
\Rightarrow (\alpha\Rightarrow\beta\Rightarrow\text{assn})\times(\alpha\Rightarrow\beta\Rightarrow\text{assn})
\Rightarrow ((\alpha\times\beta)\Rightarrow(\alpha\times\beta)\Rightarrow\text{assn})
\times((\alpha\times\beta)\Rightarrow(\alpha\times\beta)\Rightarrow\text{assn})
\text{(infixl} \ast\text{ 65)})
\]
where $RR \ast_a SS \equiv (\prod\text{-assn}\ (\text{fst}\ RR)\ (\text{fst}\ SS),\ \prod\text{-assn}\ (\text{snd}\ RR)\ (\text{snd}\ SS))$

lemma $hfprod-fst-snd$[simp]: $\text{fst}\ (hf-pres\ \text{R\ k}) = R$ by (cases $k$) auto

67
\( \text{snd} (A \ast_a B) = \text{prod-assn} (\text{snd} A) (\text{snd} B) \)

unfolding hfprod-def by auto

Conversion from fref to hfref

**lemma** fref-to-pure-hfref:\*
  **assumes** \((f,g) \in [P]_f \rightarrow (S)_{nres-rel} \)
  **assumes** \(\forall x. x \in \text{Domain } R \land R^{-1}\)\(\text{Collect } P \implies f x = \text{RETURN } (f' x) \)
  **shows** \((\text{return } o f', g) \in [P]_a (\text{pure } R)^{k} \rightarrow \text{pure } S \)
  **apply** (rule hfrefI) **apply** (rule hn-refineI)
  **using** assms
  **apply** ((sep-auto simp: fref-def pure-def pw-le-iff pw-nres-rel-iff
               refine-pw-simps eintros del: exI))
  **apply** force
  done

Conversion from hfref to hnr

This section contains the lemmas. The ML code is further down.

**lemma** hf2hnr:
  **assumes** \((f,g) \in [P]_f R \rightarrow S \)
  **shows** \(\forall x \cdot \text{P } x \rightarrow \text{hn-refine } (\text{emp } \ast \text{hn-ctxt } (\text{fst } R) x \text{ xi}) (f \text{ xi}) (\text{emp } \ast \text{hn-ctxt } (\text{snd } R) x \text{ xi}) S (g \text{ xi}) \)
  **using** assms
  **unfolding** hfref-def
  **by** (auto simp: hn-ctxt-def)

**definition** [simp]: to-hnr-prod = prod-assn

**lemma** to-hnr-prod-fst_snd:
  \(\text{fst } (A \ast_a B) = \text{to-hnr-prod } (\text{fst } A) (\text{fst } B) \)
  \(\text{snd } (A \ast_a B) = \text{to-hnr-prod } (\text{snd } A) (\text{snd } B) \)
  **unfolding** hfprod-def **by** auto

**lemma** hnr-uncurry-unfold:
  \((\forall x \cdot \text{P } x \rightarrow \text{hn-refine}) \)
  \((\Gamma \ast \text{hn-ctxt } (\text{to-hnr-prod } A B) x \text{ xi}) \)
  \((f \text{ xi}) \)
  \((\Gamma' \ast \text{hn-ctxt } (\text{to-hnr-prod } A' B') x \text{ xi}) \)
  \(R \)
  \((f x)\)
  \(\iff (\forall b \cdot \text{P } (a,b) \rightarrow \text{hn-refine})\)
\[(\Gamma \ast \text{hn-ctxt } B \ b \ bi \ast \text{hn-ctxt } A \ a \ ai)\]
\[\text{hn-context } B \ bi \ast \text{hn-context } A \ a \ ai\]
\[R\]
\[(f \ (a,b))\]

by (auto simp: \text{hn-ctxt-def prod-assn-def star-aci})

lemma \text{hn-intro-dummy}:
\[
\forall \ x \ xi. \ P \ x \rightarrow \text{hn-refine} \ (\Gamma \ x \ xi) \ (c \ xi) \ (\Gamma' \ x \ xi) \ R \ (a \ x) \implies \forall \ x \ xi. \ P \ x \rightarrow \text{hn-refine} \ (\text{emp} \ast \Gamma \ x \ xi) \ (c \ xi) \ (\text{emp} \ast \Gamma' \ x \ xi) \ R \ (a \ x)
\]
by simp

lemma \text{hn-ctxt-ctxt-fix-conv}:
\[
\text{hn-ctxt} \ (\text{hn-ctxt } R) = \text{hn-ctxt } R
\]
by (simp add: \text{hn-ctxt-def abs-def})

lemma \text{uncurry-APP}:
\[
\text{uncurry } f \ (a,b) = f \ a \ b
\]
by auto

lemma \text{norm-RETURN-o}:
\[
\forall f. \ (\text{RETURN o f}) \ x = (\text{RETURN} \ (f \ x))
\]
\[
\forall f. \ (\text{RETURN oo f}) \ x \ y = (\text{RETURN} \ (f \ x \ y))
\]
\[
\forall f. \ (\text{RETURN ooo f}) \ x \ y \ z = (\text{RETURN} \ (f \ x \ y \ z))
\]
\[
\forall f. \ (\lambda x \ y. \ \text{RETURN ooo f } x \ y) \ x \ y \ z \ a = (\text{RETURN} \ (f \ x \ y \ z \ a))
\]
by auto

lemma \text{norm-return-o}:
\[
\forall f. \ (\text{return o f}) \ x = (\text{return} \ (f \ x))
\]
\[
\forall f. \ (\text{return oo f}) \ x \ y = (\text{return} \ (f \ x \ y))
\]
\[
\forall f. \ (\text{return ooo f}) \ x \ y \ z = (\text{return} \ (f \ x \ y \ z))
\]
\[
\forall f. \ (\lambda x \ y. \ \text{return ooo f } x \ y) \ x \ y \ z \ a = (\text{return} \ (f \ x \ y \ z \ a))
\]
by auto

lemma \text{hn-val-unit-conv-emp}[simp]: \text{hn-val} \ \text{unit-rel } x \ y = \text{emp}
by (auto simp: \text{hn-ctxt-def pure-def})

Conversion from \text{hn} to \text{hfref}

This section contains the lemmas. The ML code is further down.

abbreviation \text{id-assn} \equiv \text{pure Id}
abbreviation \text{unit-assn} \equiv \text{id-assn :: unit } \Rightarrow -

lemma \text{pure-unit-rel-eq-empty}: \text{unit-assn } x \ y = \text{emp}
by (auto simp: \text{pure-def})

lemma \text{uc-hfprod-sel}:
\[ \text{fst} \ (A \ast_a B) \ a \ c = \text{(case} \ (a,c) \ \text{of} \ ((a1,a2),(c1,c2)) \Rightarrow \text{fst} \ A \ a1 \ c1 \ast \text{fst} \ B \ a2 \ c2) \]
\[ \text{snd} \ (A \ast_a B) \ a \ c = \text{(case} \ (a,c) \ \text{of} \ ((a1,a2),(c1,c2)) \Rightarrow \text{snd} \ A \ a1 \ c1 \ast \text{snd} \ B \ a2 \ c2) \]

unfolding hpprod-def prod-assn-def[abs-def] by auto

Conversion from relation to fref

This section contains the lemmas. The ML code is further down.

definition CURRY R \equiv \{(f,g). (uncurry f, uncurry g) \in R\}

lemma fref-param1: R\rightarrow S = fref (\lambda -. True) R S
  by (auto simp: fref-def fun-relD)

lemma fref-nest: fref P1 R1 (fref P2 R2 S)
  \equiv CURRY (fref (\lambda (a,b). P1 a \& P2 b) (R1 \times, R2) S)
  apply (rule eq-reflection)
  by (auto simp: fref-def CURRY-def)

lemma in-CURRY-conv: (f, g) \in CURRY R \leftarrow\rightarrow (uncurry f, uncurry g) \in R
  unfolding CURRY-def by auto

lemma uncurry0-APP[simp]: uncurry0 c \$_x = c by auto

lemma fref-param0I: (c, a) \in R \Rightarrow (uncurry0 c, uncurry0 a) \in fref (\lambda -. True) unit-rel R
  by (auto simp: fref-def)

Composition

definition hr-comp ::('b \Rightarrow 'c \Rightarrow assn) \Rightarrow ('b \times 'a) set \Rightarrow 'a \Rightarrow 'c \Rightarrow assn
  -- Compose refinement assertion with refinement relation
  where hr-comp R1 R2 a c \equiv \exists A b. R1 b c \ast \gamma ((b,a)\in R2)

definition hrp-comp ::('d \Rightarrow 'b \Rightarrow assn) \times ('d \Rightarrow 'c \Rightarrow assn)
  \Rightarrow ('d \times 'a) set \Rightarrow ('a \Rightarrow 'b \Rightarrow assn) \times ('a \Rightarrow 'c \Rightarrow assn)
  -- Compose argument assertion-pair with refinement relation
  where hrp-comp RR' S \equiv (hr-comp (fst RR') S, hr-comp (snd RR') S)

lemma hr-compI: (b,a)\in R2 \Longrightarrow R1 b c \Longrightarrow_A hr-comp R1 R2 a c
  unfolding hr-comp-def
  by sep-auto

lemma hr-comp-Id1[simp]: hr-comp (pure Id) R = pure R
  unfolding hr-comp-def[abs-def] pure-def
  apply (intro ext ent-iffI)
  by sep-auto+
lemma hr-comp-Id2\[simp\]: hr-comp R Id = R
unfolding hr-comp-def[abs-def]
apply ( intro ext ent-iffI)
by sep-auto+

lemma hr-comp-emp\[simp\]: hr-comp (λa c. emp) R a c = ↑(∃ b. (b,a)∈R)
unfolding hr-comp-def[abs-def]
apply (intro ext ent-iffI)
apply sep-auto+
done

lemma hr-comp-prod-conv\[simp\]:
hr-comp (prod-assn Ra Rb) (Ra' × Rb')
= prod-assn (hr-comp Ra Ra') (hr-comp Rb Rb')
unfolding hr-comp-def[abs-def] prod-assn-def[abs-def]
apply (intro ext ent-iffI)
apply solve-entails apply clarsimp apply sep-auto
apply clarsimp apply (intro ent-ex-preI)
apply (rule ent-ex-postI) apply (sep-auto split: prod.splits)
done

lemma hr-comp-pure: hr-comp (pure R) S = pure (R O S)
apply (intro ext)
apply (rule ent-iffI)
unfolding hr-comp-def[abs-def]
apply (sep-auto simp: pure-def)+
done

lemma hr-comp-is-pure\[safe-constraint-rules\]: is-pure A ⇒ is-pure (hr-comp A B)
by (auto simp: hr-comp-pure is-pure-conv)

lemma hr-comp-the-pure: is-pure A ⇒ the-pure (hr-comp A B) = the-pure A O B
unfolding is-pure-conv
by (clarsimp simp: hr-comp-pure)

lemma rdomp-hrcomp-conv: rdomp (hr-comp A R) x \(\leftrightarrow\) (∃ y. rdomp A y ∧ (y,x)∈R)
by (auto simp: rdomp-def hr-comp-def)

lemma hn-rel-compI:
[notfail a; (b,a)∈(R2)\nres-rel] ⇒ hn-rel R1 b c ⇒ \(\exists\) A hn-rel (hr-comp R1 R2)
a c
unfolding hr-comp-def hn-rel-def \nres-rel-def
apply (clarsimp intro!: ent-ex-preI)
apply (drule (1) order-trans)
apply (simp add: ret-le-down-conv)
by sep-auto

lemma hr-comp-precise [constraint-rules]:
  assumes [safe-constraint-rules]: precise R
  assumes SV :: single-valued S
  shows precise (hr-comp R S)
  apply (rule preciseI)
  unfolding hr-comp-def
  apply clarsimp
  by (metis SV assms (1) preciseD single-valuedD)

lemma hr-comp-assoc: hr-comp (hr-comp R S) T = hr-comp R (S O T)
  apply (intro ext)
  unfolding hr-comp-def
  apply (rule ent-iffI ; clarsimp)
  apply sep-auto
  apply (rule ent-ex-preI ; clarsimp)
  apply sep-auto
  done

lemma hnr-comp:
  assumes R: \(\forall \, b_1 \, c_1. \, P \, b_1 \Longrightarrow \text{hn-refine} \, (R \, b_1 \, c_1 \, \Gamma \, (c \, c_1) \, (R_{1p} \, b_1 \, c_1 \, * \, \Gamma')) \, R \, (b \, b_1)\)
  assumes S: \(\forall \, a_1 \, b_1. \, [Q \, a_1; \, (b_1, \, a_1) \in R_{1}'\] \Longrightarrow (b \, b_1, \, a_1) \in \langle \, R' \rangle \text{nres-rel} \)
  assumes PQ: \(\forall \, a_1 \, b_1. \, [Q \, a_1; \, (b_1, \, a_1) \in R_{1}'\] \Longrightarrow P \, b_1\)
  assumes Q: \(\forall \, a_1\)
  shows hn-refine
  (hr-comp R1 R1' a1 c1 * Γ)
  (c c1)
  (hr-comp R1p R1' a1 c1 * Γ')
  (hr-comp R R')
  (a a1)
  unfolding hn-refine-alt
proof clarsimp
  assume NF: nofail (a a1)
  show <hr-comp R1 R1' a1 c1 * Γ>
    c c1
    <\\lambda r. \text{hn-rel} \, (hr-comp R \, R') \, (a \, a1) \, r \, * \, (hr-comp R_{1p} \, R_{1}' \, a1 \, c1 \, * \, \Gamma')>_t
    apply (subst hr-comp-def)
    apply (clarsimp intro!: norm-pre-ex-rule)
proof –
  fix b1
  assume R1: (b1, a1) ∈ R1'

  from S R1 Q have R': (b b1, a a1) ∈ (R')nres-rel by blast
  with NF have NFB: nofail (b b1)
by (simp add: nres-rel-def pw-le-iff refine-pw-simps)

from PQ R1 Q have P: P b1 by blast
with NFB R have \(<\lambda r. \text{hn-rel} (\text{hr-comp} R R') (a a1) r * (R1p b1 c1) c1 * \Gamma')>_t
  unfolding \text{hn-refine-alt} by auto
thus \(<\lambda r. \text{hn-rel} (\text{hr-comp} R R') (a a1) r * (R1p b1 c1) c1 * \Gamma')>_t
  apply (rule cons-post-rule)
  apply (solve-entails)
  by (intro ent-star-mono \text{hn-rel-compI}[\text{OF NF} R] \text{hr-compI}[\text{OF} R1] \text{ent-refl})
qed

lemma \text{hrn-comp1-aux}:
  assumes R: \(\forall b1 c1. P b1 = \implies \text{hn-refine} (\text{hn-ctxt} R1 b1 c1) (c c1) (\text{hn-ctxt} R1p b1 c1) R (b b1)\)
  assumes S: \(\forall a1 b1. [Q a1; (b1,a1)\in R1]' \implies (b1,a1)\in (\langle R' \rangle \text{nres-rel})\)
  assumes PQ: \(\forall a1 b1. [Q a1; (b1,a1)\in R1]' \implies P b1\)
  assumes Q: Q a1
  shows \(\text{hn-refine} (\langle a1, a1 \rangle) (\text{hr-comp} R1 R1' a1 c1) (\langle c, c1 \rangle) (\text{hr-comp} R R') (a a1)\)
  using assms \text{hrn-comp}[\text{where} \Gamma=\text{emp} and \Gamma'=\text{emp} and a=a and b=b and c=c and P=P and Q=Q]
  unfolding \text{hn-ctxt-def}
by auto

lemma \text{hfcomp}:
  assumes A: \((f,g) \in [P]_a RR' \rightarrow S\)
  assumes B: \((g,h) \in [Q]_f T \rightarrow (U)\text{nres-rel}\)
  shows \((f,h) \in [\lambda a. Q a \wedge (\forall a'. (a',a)\in T \rightarrow P a')]_a h r p - c o m p R R' T \rightarrow h r - c o m p S U\)
  using assms
  unfolding \text{fref-def hfref-def hrp-comp-def}
  apply clarsimp
  apply (rule \text{hrn-comp1-aux}[of \(P \text{\ fam} \text{\{\text{rr}\}} \text{\ fam} \text{\{\text{rr}\}} f s n d\ S g \lambda a. Q a \wedge (\forall a'. (a',a)\in T \rightarrow P a') T \text{\ fam} \text{\{\text{th}\}} U\))]
  apply (auto simp: \text{hn-ctxt-def})
  done

lemma \text{hfref-weaken-pre-nofail}:
  assumes \((f,g) \in [P]_a R \rightarrow S\)
  shows \((f,g) \in [\lambda x. \text{nofail} (g x) \rightarrow P x]_a R \rightarrow S\)
  using assms

73
unfolding hfref-def hn-refine-def
by auto

lemma hfref-cons:
assumes \((f, g) \in [P]_a R \rightarrow S\)
assumes \(\forall x. \; P' x \Longrightarrow P x\)
assumes \(\forall x y. \; \text{fst } R' x y \Longrightarrow \text{fst } R x y\)
assumes \(\forall x y. \; \text{snd } R x y \Longrightarrow \text{snd } R' x y\)
assumes \(\forall x y. \; S x y \Longrightarrow S' x y\)
shows \((f, g) \in [P']_a R' \rightarrow S'\)
unfolding hfref-def
apply clarsimp
apply (rule hn-refine-cons)
apply (rule assms\(\langle 3\rangle\))
defer
apply (rule entt-trans[OF assms\(\langle 4\rangle\)]; sep-auto)
apply (rule assms\(\langle 5\rangle\))
apply (frule assms\(\langle 2\rangle\))
using assms\(\langle 1\rangle\)
unfolding hfref-def
apply auto
done

Composition Automation

This section contains the lemmas. The ML code is further down.

lemma prod-hrp-comp:
hrp-comp \((A \times a B) \times r D) = hrp-comp A \times_a hrp-comp B D\)
unfolding hrp-comp-def hfprod-def by simp

lemma hrp-comp-keep: hrp-comp \((A^k) B) = (hr-comp A B)^k\)
by (auto simp: hrp-comp-def)

lemma hrp-comp-invalid: hrp-comp (invalid-assn R1) R2 = invalid-assn (hr-comp R1 R2)
apply (intro enth-iffI entailsI ext)
unfolding invalid-assn-def hr-comp-def
by auto

lemma hrp-comp-dest: hrp-comp \((A^d) B) = (hr-comp A B)^d\)
by (auto simp: hrp-comp-def hr-comp-invalid)

definition hrp-imp RR RR' ≡
\(\forall a b. \; (\text{fst } RR' a b \Longrightarrow \text{fst } RR a b) \wedge (\text{snd } RR a b \Longrightarrow \text{snd } RR' a b)\)

lemma hfref-imp: hrp-imp RR RR' \(\rightarrow [P]_a RR \rightarrow S \subseteq [P]_a RR' \rightarrow S\)
apply clarsimp
apply (erule hfref-cons)
apply (simp-all add: hrp-imp-def)
done

lemma hrp-imp-refl: hrp-imp RR RR
  unfolding hrp-imp-def by auto

lemma hrp-imp-reflI: RR = RR' ==\> hrp-imp RR RR'
  unfolding hrp-imp-def by auto

lemma hrp-comp-cong: hrp-imp A A' ==\> B=B' ==\> hrp-imp (hrp-comp A B)
  (hrp-comp A' B')
  by (sep-auto simp: hrp-imp-def hrp-comp-def hr-comp-def entailst-def)

lemma hrp-prod-cong: hrp-imp A A' ==\> hrp-imp B B' ==\> hrp-imp (A\*\_\_\_ B)
  (A'\*\_\_\_ B')
  by (sep-auto simp: hrp-imp-def prod-assn-def intro: entt-star-mono)

lemma hrp-imp-trans: hrp-imp A B ==\> hrp-imp B C ==\> hrp-imp A C
  unfolding hrp-imp-def
  by (fastforce intro: entt-trans)

lemma fcomp-norm-dflt-init: x \in [P]_a R \rightarrow T ==\> hrp-imp R S ==\> x \in [P]_a S \rightarrow T
  apply (erule set-rev-mp)
  by (rule hfref-imp)

definition comp-PRE R P Q S ≡ λx. S x \rightarrow (P x \land (\forall y. (y,x)\in R \rightarrow Q x y))

lemma comp-PRE-cong[cong]:
  assumes R≡R'
  assumes \land x. P x ≡ P' x
  assumes \land x. S x ≡ S' x
  assumes \land y. [P x; (y,x)\in R; y\in Domain R; S' x] ==\> Q x y ≡ Q' x y
  shows comp-PRE R P Q S ≡ comp-PRE R' P' Q' S'
  using assms
  by (fastforce simp: comp-PRE-def intro!: eq-reflection ext)

lemma cref-compI-PRE:
  [ (f,g)\in cref P R1 R2; (g,h)\in cref Q S1 S2 ]
  ==\> (f,h) \in cref (comp-PRE S1 Q (λ-. P) (λ-. True)) (R1 O S1) (R2 O S2)
  using cref-compI[of P R1 R2 Q S1 S2]
  unfolding comp-PRE-def
  by auto

lemma PRE-D1: (Q x \land P x) ==\> comp-PRE S1 Q (λx -. P x) S x
  by (auto simp: comp-PRE-def)
lemma PRE-D2: \((Q x \land (\forall y. (y,x) \in S1 \rightarrow S x \rightarrow P x y)) \rightarrow \text{comp-PRE } S1\)

\(Q P S x\)

by (auto simp: comp-PRE-def)

lemma fref-weaken-pre:
  assumes \(\forall x. P x \rightarrow P' x\)
  assumes \((f,h) \in \text{fref } P' R S\)
  shows \((f,h) \in \text{fref } P R S\)
  apply (rule set-rev-mp[OF assms(2) fref-mono])
  using assms(1) by auto

lemma fref-PRE-D1:
  assumes \((f,h) \in \text{fref } (\text{comp-PRE } S1 Q (\lambda x. P x)) X R S\)
  shows \((f,h) \in \text{fref } (\lambda x. Q x \land P x) R S\)
  by (rule fref-weaken-pre[OF PRE-D1 assms])

lemmas fref-PRE-D = fref-PRE-D1 fref-PRE-D2

lemma hfref-weaken-pre:
  assumes \(\forall x. P x \rightarrow P' x\)
  assumes \((f,h) \in \text{hfref } P' R S\)
  shows \((f,h) \in \text{hfref } P R S\)
  using assms
  by (auto simp: hfref-def)

lemma hfref-weaken-pre':
  assumes \(\forall x. [P x; \text{rdomp } (\text{fst } R) x] \Longrightarrow P' x\)
  assumes \((f,h) \in \text{hfref } P' R S\)
  shows \((f,h) \in \text{hfref } P R S\)
  apply (rule hfrefI)
  apply (rule hn-refine-preI)
  using assms
  by (auto simp: hfref-def rdomp-def)

lemma hfref-weaken-pre-nofail':
  assumes \((f,g) \in [P]_a R \rightarrow S\)
  assumes \(\forall x. [\text{nofail } (g x); Q x] \Longrightarrow P x\)
  shows \((f,g) \in [Q]_a R \rightarrow S\)
  apply (rule hfref-weaken-pre[OF - assms(1)[THEN hfref-weaken-pre-nofail]])
  using assms(2)
  by blast

lemma hfref-compI-PRE-aux:
assumes $A: (f, g) \in [P]_a \rightarrow S$
assumes $B: (g, h) \in [Q]_T \rightarrow (U)\text{res-rel}$
shows $(f, h) \in [\text{comp-PRE } T Q (\lambda x. P) (\lambda x. \text{nofail } (h x))]_a$
hrp-comp $RR' \rightarrow \text{hr-comp } S U$
apply (rule hfref-weaken-pre [OF - hfcomp [OF $A B$]])
by (auto simp: comp-PRE-def)

lemma hfref-compI-PRE:
assumes $A: (f, g) \in [P]_a \rightarrow S$
assumes $B: (g, h) \in [Q]_T \rightarrow (U)\text{res-rel}$
shows $(f, h) \in [\text{comp-PRE } T Q (\lambda x y. P y) (\lambda x. \text{nofail } (h x))]_a$
hrp-comp $RR' \rightarrow \text{hr-comp } S U$
using hfref-compI-PRE-aux [OF $A B$, THEN hfref-weaken-pre-nofail]
apply (rule hfref-weaken-pre [rotated])
apply (auto simp: comp-PRE-def)
done

lemma hfref-PRE-D1:
assumes $(f, h) \in \text{hfref } (\text{comp-PRE } S1 Q (\lambda x -. P x) X) \rightarrow R S$
shows $(f, h) \in \text{hfref } (\lambda x. Q x \land P x) \rightarrow R S$
by (rule hfref-weaken-pre [OF PRE-D1 assms])

lemma hfref-PRE-D2:
assumes $(f, h) \in \text{hfref } (\text{comp-PRE } S1 Q P X) \rightarrow R S$
shows $(f, h) \in \text{hfref } (\lambda x. Q x \land (\forall y. (y, x) \in S1 \rightarrow X x \rightarrow P x y)) \rightarrow R S$
by (rule hfref-weaken-pre [OF PRE-D2 assms])

lemma hfref-PRE-D3:
assumes $(f, h) \in \text{hfref } (\text{comp-PRE } S1 Q P X) \rightarrow R S$
shows $(f, h) \in \text{hfref } (\text{comp-PRE } S1 Q P X) \rightarrow R S$
using assms .

lemmas hfref-PRE-D = hfref-PRE-D1 hfref-PRE-D3

1.5.4 Automation
Purity configuration for constraint solver
lemmas [safe-constraint-rules] = pure-pure

Configuration for hfref to hn conversion
named-theorems to-hnr-post (to-hnr converter: Postprocessing unfold rules)

lemma uncurry0-add-app-tag: uncurry0 $\text{RETURN } c = \text{uncurry0 } (\text{RETURN } \$ c)$
by simp

lemmas [to-hnr-post] = norm-RETURN-o norm-return-o
uncurry0-add-app-tag uncurry0-add-app uncurry0-APP hn-val-unit-conv-emp
mult-1 [of $x::\text{assn } for x$] mult-1-right [of $x::\text{assn } for x$]

77
named-theorems to-hfref-post (to-hfref converter: Postprocessing unfold rules)
lemma prod-casesK[to-hfref-post]: case-prod (λ _ . k) = (λ _ . k) by auto
lemma uncurry0-hfref-post[to-hfref-post]: hfref (uncurry0 True) R S = hfref (λ . True) R S
  apply (fo-rule arg-cong fun-cong)+ by auto

Configuration for relation normalization after composition
named-theorems fcomp-norm-unfold (fcomp−normalizer: Unfold theorems)
named-theorems fcomp-norm-simps (fcomp−normalizer: Simplification theorems)
named-theorems fcomp-norm-init fcomp−normalizer: Initialization rules
named-theorems fcomp-norm-trans fcomp−normalizer: Transitivity rules
named-theorems fcomp-norm-cong fcomp−normalizer: Congruence rules
named-theorems fcomp-norm-norm fcomp−normalizer: Normalization rules
named-theorems fcomp-norm-refl fcomp−normalizer: Reflexivity rules

Default Setup
lemmas [fcomp-norm-unfold] = prod-rel-comp nres-rel-comp Id-O-R R-O-Id
lemmas [fcomp-norm-unfold] = hr-comp-Id1 hr-comp-Id2
lemmas [fcomp-norm-unfold] = hr-comp-prod-conv
lemmas [fcomp-norm-unfold] = prod-hrp-comp hrp-comp-keep hrp-comp-dest hr-comp-pure

lemma [fcomp-norm-simps]: CONSTRAINT is-pure P =⇒ pure (the-pure P) = P by simp
lemmas [fcomp-norm-simps] = True-implies-equals
lemmas [fcomp-norm-init] = fcomp-norm-dflt-init
lemmas [fcomp-norm-trans] = hrp-imp-trans
lemmas [fcomp-norm-cong] = hrp-comp-cong hrp-prod-cong
lemmas [fcomp-norm-refl] = refl hrp-imp-refl
lemma ensure-fref-nresI: (f,g)∈P R→S =⇒ (RETURN o f , RETURN o g)∈[P]f R→(S)nres-rel
  by (auto intro: nres-refl simp: fref-def)
lemma ensure-fref-nres-unfold:
  \( \forall f. \text{RETURN } o (\text{uncurry0 } f) = \text{uncurry0 } (\text{RETURN } f) \)
  \( \forall f. \text{RETURN } o (\text{uncurry } f) = \text{uncurry } (\text{RETURN } oo f) \)
  \( \forall f. (\text{RETURN } oo \text{uncurry } f) = \text{uncurry } (\text{RETURN } oo f) \)
  by auto

Composed precondition normalizer
named-theorems fcomp-prenorm-simps (fcomp precondition−normalizer: Simplification theorems)

Support for preconditions of the form \(-\in Domain R\), where R is the relation
of the next more abstract level.

\textbf{declare} DomainI [\texttt{fcomp-prenorm-simps}]

\textbf{lemma} auto-weaken-pre-init-hf:
\begin{itemize}
  \item assumes $\forall x. \text{PROTECT } P \ x \rightarrow P' x$
  \item assumes $(f,h) \in hfref P' R S$
  \item shows $(f,h) \in hfref P R S$
\end{itemize}
\textbf{using} \texttt{assms}
\textbf{by} (\texttt{auto simp: hfref-def})

\textbf{lemma} auto-weaken-pre-init-f:
\begin{itemize}
  \item assumes $\forall x. \text{PROTECT } P \ x \rightarrow P' x$
  \item assumes $(f,h) \in fref P' R S$
  \item shows $(f,h) \in fref P R S$
\end{itemize}
\textbf{using} \texttt{assms}
\textbf{by} (\texttt{auto simp: fref-def})

\textbf{lemmas} auto-weaken-pre-init = auto-weaken-pre-init-hf auto-weaken-pre-init-f

\textbf{lemma} auto-weaken-pre-uncurry-step:
\begin{itemize}
  \item assumes $\text{PROTECT } f \ a \equiv f'$
  \item shows $\text{PROTECT } (\lambda(x,y). f \ x \ y) \ (a,b) \equiv f' \ b$
\end{itemize}
\textbf{using} \texttt{assms}
\textbf{by} (\texttt{auto simp: curry-def dest: meta-eq-to-obj-eq intro: eq-reflection})

\textbf{lemma} auto-weaken-pre-uncurry-finish:
$\text{PROTECT } f \ x \equiv f \ x$ \textbf{by} (\texttt{auto})

\textbf{lemma} auto-weaken-pre-uncurry-start:
\begin{itemize}
  \item assumes $P \equiv P'$
  \item assumes $P' \rightarrow Q$
  \item shows $P \rightarrow Q$
\end{itemize}
\textbf{using} \texttt{assms by} (\texttt{auto})

\textbf{lemma} auto-weaken-pre-comp-PRE-I:
\begin{itemize}
  \item assumes $S \ x \Rightarrow P \ x$
  \item assumes $\forall y. [(y,x) \in R; P x; S x] \Rightarrow Q x y$
  \item shows $\text{comp-PRE } R P Q S x$
\end{itemize}
\textbf{using} \texttt{assms by} (\texttt{auto simp: comp-PRE-def})

\textbf{lemma} auto-weaken-pre-to-imp-nf:
\begin{align*}
  (A \rightarrow B \rightarrow C) &= (A \land B \rightarrow C) \\
  ((A \land B) \land C) &= (A \land B \land C)
\end{align*}
\textbf{by} auto

\textbf{lemma} auto-weaken-pre-add-dummy-imp:
\begin{itemize}
  \item $P \Rightarrow \text{True} \rightarrow P$ \textbf{by} \texttt{simp}
\end{itemize}

Synthesis for hfref statements
definition hfsynth-ID-R :: ('a ⇒ ⇒ assn) ⇒ 'a ⇒ bool where
  [simp]: hfsynth-ID-R - - ≡ True

lemma hfsynth-ID-R-D:
  fixes I :: 'a itself
  assumes hfsynth-ID-R R a
  assumes intf-of-assn R I
  shows a :: i I
  by simp

lemma hfsynth-hnr-from-hfI:
  assumes ∀ x xi. P x ∧ hfsynth-ID-R (fst R) x −→ hn-refine (emp * hn-ctxt (fst R) x xi) (emp * hn-ctxt (snd R) x xi) S (g$x)
  shows (f,g) ∈ [P]a R → S
  using asms
  unfolding href-def
  by (auto simp: hn-ctxt-def)

lemma hfsynth-ID-R-uncurry-unfold:
  hfsynth-ID-R (to-hnr-prod R S) (a,b) ≡ hfsynth-ID-R R a ∧ hfsynth-ID-R S b
  hfsynth-ID-R (fst (hf-pres R k)) ≡ hfsynth-ID-R R
  by (auto intro: eq-reflection)

ML (}

signature SEPREF-RULES = sig
  (* Analysis of relations, both cref and fun-rel *)
  (* R1→...→Rn→- [.]f ((R1×,R2)...×,Rn) → [R1,...,Rn] *)
  val binder-rels: term ⇒ term list
  (* →→→→S / [.]f - → S ⇒ S *)
  val body-rel: term ⇒ term
  (* Map →/ cref to (precond,args,res). NONE if no/trivial precond. *)
  val analyze-rel: term ⇒ term option * term list * term
  (* Make trivial (λ-. True) precond *)
  val mk-triv-precond: term list ⇒ term
  (* Make [P]f ((R1×,R2)...×,Rn) → S. Insert trivial precond if NONE. *)
  val mk-rel: term option * term list * term ⇒ term
  (* Map relation to (args,res) *)
  val strip-rel: term ⇒ term list * term

  (* Make hfprod (op *a) *)
  val mk-hfprod : term * term ⇒ term
  val mk-hfprods : term list ⇒ term

  (* Determine interface type of refinement assertion, using default fallback if necessary. Use named-thms intf-of-assn for configuration. *)
  val intf-of-assn : Proof.context ⇒ term ⇒ typ

  80
Convert a parametricity theorem in higher-order form to uncurried \texttt{fref} form. For functions without arguments, a unit-argument is added.

\textit{TODO/FIXME}: Currently this only works for higher-order theorems, \textit{i.e.}, theorems of the form \((f,g)\in R_1\rightarrow\ldots\rightarrow R_n\).

First-order theorems are silently treated as refinement theorems for functions with zero arguments, \textit{i.e.}, a unit-argument is added.

\begin{verbatim}
val to-fref : Proof.context -> thm -> thm

(* Convert a parametric or fref theorem to first order form *)
val to-foparam : Proof.context -> thm -> thm

(* Convert schematic hfref goal to hnr goal *)
val prepare-hfref-synth-tac : Proof.context -> tactic'

(* Convert theorem in hfref form to hnr form *)
val to-hnr : Proof.context -> thm -> thm

(* Convert theorem in hnr form to hfref form *)
val to-hfref : Proof.context -> thm -> thm

(* Convert theorem to given form, if not yet in this form *)
val ensure-fref : Proof.context -> thm -> thm
val ensure-fref-ars : Proof.context -> thm -> thm
val ensure-hfref : Proof.context -> thm -> thm
val ensure-hnr : Proof.context -> thm -> thm

type hnr-analysis = {
  thm: thm, (* Original theorem, may be normalized *)
  precond: term, (* Precondition, abstracted over abs-arguments *)
} (* Premises not depending on arguments *)
val analyze-hnr : Proof.context -> thm -> hnr-analysis
val pretty-hnr-analysis: Proof.context -> hnr-analysis -> Pretty.T
val mk-hfref-thm: Proof.context -> hnr-analysis -> thm
\end{verbatim}
(* Simplify precondition of \texttt{fref}/\texttt{hfref}–theorem *)
val simplify-precond: Proof.context -> thm -> thm

(* Normalize \texttt{hfref}–theorem after composition *)
val norm-fcomp-rule: Proof.context -> thm -> thm

(* Replace pure \texttt{?A} by \texttt{?A'} and is-pure constraint, then normalize *)
val add-pure-constraints-rule: Proof.context -> thm -> thm

(* Compose \texttt{fref}/\texttt{hfref} and \texttt{fref} theorem, to produce \texttt{hfref} theorem. The input theorems may also be in \texttt{ho-param} or \texttt{hnr} form, and are converted accordingly. *)
val gen-compose: Proof.context -> thm -> thm -> thm

(* \texttt{FCOMP}–attribute *)
val fcomp-attrib: attribute context-parser

end

structure Sepref-Rules: SEPFREF-RULES = struct

local open Refine-Util Relators in
fun binder-rels @\{mpat \texttt{?F} \to \texttt{?G}\} = F::binder-rels G
  | binder-rels @\{mpat \texttt{fref} \to \texttt{?F}\} = strip-prodrel-left F
  | binder-rels @\{} = []

local
  fun br-aux @\{mpat \to \?G\} = br-aux G
  | br-aux R = R
in
  fun body-rel @\{mpat \texttt{fref} \to \?G\} = G
  | body-rel R = br-aux R
end

fun strip-rel R = (binder-rels R, body-rel R)

fun analyze-rel @\{mpat \texttt{fref} (\lambda\_. \texttt{True}) \texttt{?R} \texttt{?S}\} = (NONE,strip-prodrel-left R,S)
  | analyze-rel @\{mpat \texttt{fref} \texttt{?P} \to \texttt{?R} \texttt{?S}\} = (SOME P,strip-prodrel-left R,S)
  | analyze-rel R = let
    val (args,res) = strip-rel R
  in
    (NONE,args,res)
  end

fun mk-triv-precond Rs = absdummy (map rel-absT Rs |> list-prodT-left)
@\{term \texttt{True}\}

fun mk-rel (P,Rs,S) = let
val $R = \text{list-prodrel-left } Rs$

val $P = \text{case } P \text{ of}$
  $SOME \quad P \implies P$
  $NONE \implies \text{mk-triv-precond } Rs$

fun $\text{mk-hfprod } (a, b) = \text{mk-term } ?a * ?b$

local
  fun $\text{mk-hfprods-rev } [] = \text{mk-term } \text{unit-assn}$
  $| \text{mk-hfprods-rev } [Rk] = Rk$
  $| \text{mk-hfprods-rev } (Rkn::Rks) = \text{mk-hfprod } (\text{mk-hfprods-rev } Rks, Rkn)$
  in
    val $\text{mk-hfprods } = \text{mk-hfprods-rev } \circ \text{rev}$
  end

fun $\text{intf-of-assn } ctxt \quad t = \text{let}$
  val $\text{orig-ctxt } = ctxt$
  val $(t,ctxt) = \text{yield-singleton } (\text{Variable.import-terms false}) \quad t \quad ctxt$
  val $v = \text{TVar } ((T,0),\text{Proof-Context.default-sort } ctxt \quad (T,0)) \implies \text{Logic.mk-type }$
  val $\text{goal } = \text{mk-term } \text{Trueprop } (\text{intf-of-assn } ?t \quad ?v)$
  val $\text{i-of-assn-rls } = \text{Named-Theorems-Rev.get } ctxt \quad @\{\text{named-theorems-rev intf-of-assn}\}$
  $\quad @\{\text{thms intf-of-assn-fallback}\}$
  fun $\text{tac } ctxt = \text{REPEAT-ALL-NEW } (\text{resolve-tac ctxt } \text{i-of-assn-rls})$
  val $\text{thm } = \text{Goal.prove } ctxt \quad [] \quad [] \quad \text{goal } (\text{fn } \{\text{context},...\} \implies \text{ALLGOALS } (\text{tac context}))$
  val $\text{inf } = \text{case } \text{Thm.concl-of } \text{thm } \text{of}$
  $\quad @\{(\text{mpat Trueprop } (\text{inf-of-assn } - (\text{?v } AS_p \quad \text{TYPE } (-)))) \implies v\}$
  $\quad | \cdot \implies \text{raise } \text{THM } (\text{Inf-of-assn: Proved a different theorem?}, \sim 1, [\text{thm}])$
  val $\text{intf } = \text{singleton } (\text{Variable.export-terms } ctxt \quad \text{orig-ctxt}) \quad \text{inf}$
  $\implies \text{Logic.dest-type }$
  in
    $\text{intf}$
  end
datatype rthm-type =
| RT-HOPARAM (* (\cdot, \cdot) \in \cdot \rightarrow \ldots \rightarrow \cdot *)
| RT-FREF (* (\cdot, \cdot) \in [\cdot]f \rightarrow \cdot *)
| RT-HNR (* hn-refine - - - - *)
| RT-HFREF (* (\cdot, \cdot) \in [\cdot]a \rightarrow \cdot *)
| RT-OTHER

fun rthm-type thm =
case Thm.concl_of thm |
HOLogic.dest_Trueprop of
  @{mpat (-,-) \in fref - - -} => RT-FREF
  @{mpat (\cdot,\cdot) \in hfref - - -} => RT-HFREF
| @{mpat hn-refine - - - - -} => RT-HNR
| @{mpat (\cdot,\cdot) \in -} => RT-HOPARAM (* TODO: Distinction between ho-param and fo-param *)
| - => RT-OTHER

fun to-fref ctxt thm =
  let
    open Conv
  in
case Thm.concl_of thm |
HOLogic.dest_Trueprop of
  @{mpat (-,-) \in ->} =>
    Local-Defs.unfold0 ctxt @{thms fref-param1} thm
    |> fconv-rule (repeat-conv (Refine-Util.ftop-conv (K (rewr-conv @{thm fref-nest}))))
  | @{mpat (\cdot,\cdot) \in -} => thm RS @{thm fref-param0I}
  | - => raise THM (to-fref: Expected theorem of form (\cdot,\cdot) \in \sim 1,\cdot [thm])
end

fun to-foparam ctxt thm =
  val unf-thms = @{thms}
    split-tupled-all prod-rel-simp uncurry-apply cnv-conj-to-meta Product-Type.split
  in
case Thm.concl_of thm of
  @{mpat Trueprop ((\cdot,\cdot) \in fref - - -)} =>
    @{thm frefD} OF [thm]
    |> forall-intr-vars
    |> Local-Defs.unfold0 ctxt unf-thms
    |> Variable.gen-all ctxt
  | @{mpat Trueprop ((\cdot,\cdot) \in -)} =>
    Parametricity,fo-rule thm
    |> raise THM (Expected parametricity or fref theorem, \sim 1,[thm])
end

fun to-hnr ctxt thm =
  (thm RS @{thm hf2hnr})
  |> Local-Defs.unfold0 ctxt @{thms to-hnr-prod-fst-snd keep-drop-sels} (* Resolve fst and snd over *,a and R^a, R^d * )
Local-Defs.unfold0 ctxt @\{thms hnr-uncurry-unfold\} (* Resolve products for uncurried parameters *)

Local-Defs.unfold0 ctxt @\{thms uncurry-apply uncurry-APP assn-one-left split\} (* Remove the uncurry modifiers, the emp−dummy, and unfold product cases *)

Local-Defs.unfold0 ctxt @\{thms hnr-ctxt-ctxt-fix-conv\} (* Remove duplicate hnr-ctxt tagging *)

Local-Defs.unfold0 ctxt @\{thms all-to-meta imp-to-meta HOL.True-implies-equals HOL.implies-True-equals Pure.triv-forall-equality cnv-conj-to-meta\} (* Convert to meta−level, remove vacuous condition *)

Local-Defs.unfold0 ctxt (Named-Theorems.get ctxt @\{named-theorems to-hnr-post\}) (* Post−Processing *)

Goal.norm-result ctxt

Conv.fconv-rule Thm.eta-conversion

(* Convert schematic hrref−goal to hn−refine goal *)

fun prepare-hrref-synth-tac ctxt = let
val i-of-assn-rls =
Named-Theorems-Rev.get ctxt @\{named-theorems-rev intf-of-assn\}
@ @
\{thms intf-of-assn-fallback\}
val to-hnr-post-rls =
Named-Theorems.get ctxt @\{named-theorems to-hnr-post\}

val i-of-assn-tac = (REPEAT' (DETERM o dresolve-tac ctxt @\{thms hfsynth-ID-R-D\})
THEN' DETERM o SOLVED' (REPEAT-ALL-NEW (resolve-tac ctxt
i-of-assn-rls)))
)

in
(* Note: To re−use the to-hnr infrastructure, we first work with
$−tags on the abstract function, which are finally removed. *)

resolve-tac ctxt @\{thms hfsynth-hnr-from-hfI\} THEN-ELSE' (SELECT-GOAL (unfold-tac ctxt @\{thms to-hnr-prod-fst-snd keep-drop-sels hf-pres-fst\} (* Distribute fst, snd over product and hf-pres *)
THEN unfold-tac ctxt @\{thms hnr-uncurry-unfold hfsynth-ID-R-uncurry-unfold\} (* Curry parameters *)
THEN unfold-tac ctxt @\{thms uncurry-apply uncurry-APP assn-one-left split\} (* Curry parameters (II) and remove emp assertion *)
(*THEN unfold-tac ctxt @\{thms hn-ctxt-ctxt-fix-conv\} (* Remove duplicate hn-ctxt (Should not be necessary) *)*)
THEN unfold-tac ctxt @\{thms all-to-meta imp-to-meta HOL.True-implies-equals HOL.implies-True-equals Pure.triv-forall-equality cnv-conj-to-meta\} (* Convert precondition to meta−level *)
THEN ALLGOALS i-of-assn-tac (* Generate ::;−premises *)
THEN unfold-tac ctxt to-hnr-post-rls (* Postprocessing *)
THEN unfold-tac ctxt @\{thms APP-def\} (* Get rid of $-$ tags *)
)
,
K all-tac
)
end

(*******************************
(* Analyze hnr *)
structure Termtab2 = Table(
  type key = term * term
  val ord = prod-ord Term-Ord.fast-term-ord Term-Ord.fast-term-ord);

type hnr-analysis = {
  thm: thm,
  precond: term,
  prems : term list,
  ahead: term * bool,
  chead: term * bool,
  argrels: (term * bool) list,
  result-rel: term
}

fun analyze-hnr (ctxt:Proof.context) thm = let

  (* Debug information: Stores string*term pairs, which are pretty-printed on error *)
  val dbg = Unsynchronized.ref []
  fun add-dbg msg ts = (dbg := (msg,ts) :: !dbg;())
  fun pretty-dbg (msg,ts) = Pretty.block [
    Pretty.str msg,
    Pretty.str :,
    Pretty.brk 1,
    Pretty.list [ ] (map (Syntax.pretty-term ctxt) ts)
  ]
  fun pretty-dbgs l = map pretty-dbg l |> Pretty.fbreaks |> Pretty.block

  fun trace-dbg msg = Pretty.block [Pretty.str msg, Pretty.brk, pretty-dbgs (rev (!dbg))] |> Pretty.string-of |> tracing

  fun fail msg = (trace-dbg msg; raise THM(msg,~1,[thm]))
  fun assert cond msg = cond orelse fail msg;

86
Heads may have a leading return/RETURN.
The following code strips off the leading return, unless it has the form
return x for an argument x

fun check-strip-leading args t f = (* Handle the case RETURN x, where x
is an argument *)
  if Termtab.defined args f then (t,false) else (f,true)

fun strip-leading-RETURN args (t as @{mpat RETURN$(?f)}) = check-strip-leading
args t f
  | strip-leading-RETURN args (t as @{mpat RETURN ?f}) = check-strip-leading
args t f
  | strip-leading-RETURN - t = (t,false)

fun strip-leading-return args (t as @{mpat return$(?f)}) = check-strip-leading
args t f
  | strip-leading-return args (t as @{mpat return ?f}) = check-strip-leading
args t f
  | strip-leading-return - t = (t,false)

(* The following code strips the arguments of the concrete or abstract
function. It knows how to handle APP−tags ($), and stops at PR-CONST−tags.

Moreover, it only strips actual arguments that occur in the
precondition−section of the hn-refine−statement. This ensures
that non−arguments, like maxsize, are treated correctly.

fun strip-fun - (t as @{mpat PR-CONST -}) = (t,[])
  | strip-fun s (t as @{mpat ?f?$?x}) = check-arg s t f x
  | strip-fun s (t as @{mpat ?f ?x}) = check-arg s t f x
  | strip-fun - f = (f,[])
and check-arg s t f x =
  if Termtab.defined s x then
    strip-fun s f |> apsnd (curry op :: x)
  else (t,[])

(* Arguments in the pre/postcondition are wrapped into hn-ctxt tags.
This function strips them off. *)
fun dest-hn-ctxt @{mpat hn-ctxt ?R ?a ?c} = ((a,c),R)
  | dest-hn-ctxt - = fail Invalid hn-ctxt parameter in pre or postcondition

  | dest-hn-refine - = fail Conclusion is not a hn-refine statement

(* Strip separation conjunctions. Special case for emp, which is ignored.

87
fun is-emp @ {mpat emp} = true | is-emp - = false

val strip-star' = Sepref-Basic.strip-star #> filter (not o is-emp)

(* Compare Termtab2s for equality of keys *)
fun pairs-eq pairs1 pairs2 =
  Termtab2.forall (Termtab2.defined pairs1 o fst) pairs2
  andalso Termtab2.forall (Termtab2.defined pairs2 o fst) pairs1

fun atomize-prem @ {mpat Trueprop ?p} = p
  | atomize-prem - = fail Non-atomic premises

(* Make HOL conjunction list *)
fun mk-conjs [] = @ {const True}
  | mk-conjs [p] = p
  | mk-conjs (p::ps) = HOLogic.mk_binop @ {const-name HOL.conj} (p, mk-conjs ps)

(* Start actual analysis *)
val - = add-dbg thm [Thm.prop-of thm]
val prems = Thm.prems-of thm
val concl = Thm.concl-of thm |> HOLogic.dest-Trueprop
val (G,c,G',R,a) = dest-hn-refine concl

val pre-pairs = G
  |> strip-star'
  |> tap (add-dbg precondition)
  |> map dest-hn-ctxt
  |> Termtab2.make

val post-pairs = G'
  |> strip-star'
  |> tap (add-dbg postcondition)
  |> map dest-hn-ctxt
  |> Termtab2.make

val - = assert (pairs-eq pre-pairs post-pairs)
Parameters in precondition do not match postcondition

val aa-set = pre-pairs |> Termtab2.keys |> map fst |> Termtab.make-set
val ca-set = pre-pairs |> Termtab2.keys |> map snd |> Termtab.make-set

val (a,leading-RETURN) = strip-leading-RETURN aa-set a
val (c,leading-return) = strip-leading-return ca-set c

88
val \( \text{add-dbg stripped abstract term} \ [a] \)
val \( \text{add-dbg stripped concrete term} \ [c] \)

val \((\text{ahead},\text{aargs}) = \text{strip-fun aa-set} \ a; \)
val \((\text{chead},\text{cargs}) = \text{strip-fun ca-set} \ c; \)

val \( \text{add-dbg abstract head} \ [\text{ahead}] \)
val \( \text{add-dbg abstract args} \ \text{aargs} \)
val \( \text{add-dbg concrete head} \ [\text{chead}] \)
val \( \text{add-dbg concrete args} \ \text{cargs} \)

val \( \text{assert} \ (\text{length} \ \text{cargs} = \text{length} \ \text{aargs}) \) Different number of abstract and concrete arguments;
val \( \text{assert} \ (\text{not} \ (\text{has-duplicates} \ \text{op} \ \text{aconv} \ \text{aargs})) \) Duplicate abstract arguments
val \( \text{assert} \ (\text{not} \ (\text{has-duplicates} \ \text{op} \ \text{aconv} \ \text{cargs})) \) Duplicate concrete arguments

val \( \text{argpairs} = \text{aargs} \sim \text{cargs} \)
val \( \text{ap-set} = \text{Termtab2.make-set} \ \text{argpairs} \)
val \( \text{assert} \ (\text{pairs-eq} \ \text{pre-pairs} \ \text{ap-set}) \) Arguments from pre/postcondition do not match operation’s arguments

val \( \text{pre-rels} = \text{map} \ (\text{the} \ o \ (\text{Termtab2.lookup} \ \text{pre-pairs})) \ \text{argpairs} \)
val \( \text{post-rels} = \text{map} \ (\text{the} \ o \ (\text{Termtab2.lookup} \ \text{post-pairs})) \ \text{argpairs} \)

val \( \text{assert} \ (\text{pre-rels} \sim \text{pre-rels}) \)
val \( \text{assert} \ (\text{post-rels} \sim \text{post-rels}) \)

fun \( \text{adjust-hf-pres} @\{?Rk\} = R \ | \ \text{adjust-hf-pres} \ t = t \)
val \( \text{post-rels} = \text{map} \ \text{adjust-hf-pres} \ \text{post-rels} \)

fun \( \text{is-invalid} \ R @\{?R'\} = R \ \text{aconv} \ R' \ | \ \text{is-invalid} - @\{?R'\} = \text{false} \ | \ \text{is-invalid} - - = \text{false} \)

fun \( \text{is-keep} \ (R,R') = \)
if \( R \ \text{aconv} \ R' \) then \text{true}
else if \( \text{is-invalid} \ R \ R' \) then \text{false}
else \text{fail} Mismatch between pre and post relation for argument

val \( \text{keep} = \text{map} \ \text{is-keep} \ (\text{pre-rels} \sim \text{post-rels}) \)
val \( \text{argrels} = \text{pre-rels} \sim \text{keep} \)
val aa-set = Termtab.make-set aargs
val ca-set = Termtab.make-set cargs

fun is-precond t = 
  (exists-subterm (Termtab.defined ca-set) t andalso fail Premise contains concrete argument)
  orelse exists-subterm (Termtab.defined aa-set) t

val (preconds, prems) = split is-precond prems

val precond = 
  map atomize-prem preconds
  |> mk-conjs
  |> fold lambda aargs

val - = add-dbg precond [precond]
val - = add-dbg prems prems

in
  
  { thm = thm,
    precond = precond,
    prems = prems,
    ahead = (ahead,leading-RETURN),
    chead = (chead,leading-return),
    argrels = argrels,
    result-rel = R
  }
end

fun pretty-hnr-analysis ctxt
  ({thm,precond,ahead,chead,argrels,result-rel,...}) 
  : Pretty.T =
let
  val - = thm (* Suppress unused warning for thm *)

fun pretty-argrel (R,k) = Pretty.block [
  Syntax.pretty-term ctxt R,
  if k then Pretty.str k else Pretty.str d
]

val pretty-chead = case chead of
  (t,false) => Syntax.pretty-term ctxt t
  | (t,true) => Pretty.block [Pretty.str return , Syntax.pretty-term ctxt t]

val pretty-ahead = case ahead of
  (t,false) => Syntax.pretty-term ctxt t
| (t,true) => Pretty.block [Pretty.str "RETURN", Syntax.pretty-term ctxt t]

in
Pretty.fbreaks [
(*Display.pretty-thm ctxt thm,*
Pretty.block [Pretty.enclose [[] [pretty-chead, pretty-ahead],
Pretty.enclose [[] [Syntax.pretty-term ctxt precond],
Pretty.brk 1,
Pretty.block (Pretty.separate \rightarrow (map pretty-argrel argrels @ [Syntax.pretty-term
ctxt result-rel]))
]
] |> Pretty.block
end

fun mk-hfref-thm
ctxt
({thm,precond,prems,ahead,chead,argrels,result-rel})
= let

  fun mk-keep (R,true) = @{mk-term ?Rk}
   | mk-keep (R,false) = @{mk-term ?Rk}

  (* TODO: Move, this is of general use! *)
  fun mk-uncurry f = @{mk-term uncurry ?f}

  (* Uncurry function for the given number of arguments.
   * For zero arguments, add a unit−parameter.
   *)
  fun rpt-uncurry n t =
    if n=0 then @{mk-term uncurry0 ?t}
    else if n=1 then t
    else funpow (n−1) mk-uncurry t

  (* Rewrite uncurried lambda’s to λ(−)−form. Use top−down rewriting
to correctly handle nesting to the left.

  TODO: Combine with abstraction and uncurry−procedure,
  and mark the deviation about uncurry as redundant
  intermediate step to be eliminated.
  *)
  fun rew-uncurry-lambda t = let
    val rr = map (Logic.dest-equals o Thm.prop-of) @{thms uncurry-def uncurry0-def}
    val thy = Proof-Context.theory-of ctxt
  in
    Pattern.rewrite-term-top thy rr [] t
  end
fun gsimp-only ctxt sec = let
  val ss = put-simpset HOL-basic-ss ctxt |> sec
in asm-full-simp-tac ss end

fun simp-only ctxt thms = gsimp-only ctxt (fn ctxt = |> ctxt addsimps thms)

(* Build theorem statement *)
(* [] ===> (chead, ahead) in [precond] rels \rightarrow R *)

(* Uncurry precondition *)
val num-args = length argrels
val precond = precond
 |> rpt-uncurry num-args
 |> rew-uncurry-lambda (* Convert to nicer \lambda((...),-) \rightarrow form*)

(* Re-attach leading RETURN/return *)
fun mk-RETURN (t, r) = if r then let
  val T = funpow num-args range-type (fastype-of (fst ahead))
  val tRETURN = Const (@{const-name RETURN}, T --> Type(@{type-name nres}, [T]))
in
  Refine-Util.mk-compN num-args tRETURN t
end
else t

fun mk-return (t, r) = if r then let
  val T = funpow num-args range-type (fastype-of (fst chead))
  val tRETURN = Const (@{const-name return}, T --> Type(@{type-name Heap}, [T]))
in
  Refine-Util.mk-compN num-args tRETURN t
end
else t

(* Hrmpf! : Gone for good from 2015→2016. Inserting ctxt-based substitute here. *)
fun certify-inst ctxt (instT, inst) = (map (apsnd (Thm.ctyp_of ctxt)) instT, map (apsnd (Thm.cterm_of ctxt)) inst);

(* fun mk-RETURN (t,r) = if r then @{mk-term RETURN o ?t} else t

fun mk-return (t, r) = if r then @\{mk-term return o ?t\} else t

(* Uncurry abstract and concrete function, append leading return *)
val ahead = ahead |\> mk-RETURN |\> rpt-uncurry num-args
val chead = chead |\> mk-return |\> rpt-uncurry num-args

(* Add keep−flags and summarize argument relations to product *)
val argrel = map mk-keep argrels |\> rev (* TODO: Why this rev? *) |\> mk-hfprods

(* Produce final result statement *)
val result = @\{mk-term Trueprop ((?chead,?ahead) ∈ [?precond]a \?argrel → ?result-rel)\}
val result = Logic.list-implies (prems,result)

(**************************************************************************
(* Prove theorem *)

(* Create context and import result statement and original theorem *)
val orig-ctxt = ctxt
(*val thy = Proof-Context.theory-of ctxt*)
val (insts, ctxt) = Variable.import-inst true [result] ctxt
val insts' = certify-inst ctxt insts
val result = Term-Subst.instantiate insts result
val thm = Thm.instantiate insts' thm

(* Unfold APP tags. This is required as some APP−tags have also been unfolded by analysis *)
val thm = Local-Defs.unfold0 ctxt @\{thms APP-def\} thm

(* Tactic to prove the theorem.
A first step uses hfrefI to get a hnr−goal.
This is then normalized in several consecutive steps, which
get rid of uncurrying. Finally, the original theorem is used for resolution,
where the pre− and postcondition, and result relation are connected with a consequence rule, to handle unfolded hn-ctxt−tags, re−ordered relations,
and introduced unit−parameters (TODO:
Mark artificially introduced unit−parameter specially, it may get confused
with intentional unit−parameter, e.g., functional empty-set (!))
)

fun tac ctxt =
    resolve-tac ctxt @\{thms hfrefI\}
    THEN' gsimp-only ctxt (fn c => c
    addsimps @\{thms uncurry-def hn-ctxt-def uncurry0-def
    keep-drop-sels uc-hfprod-sel o-apply
    APP-def\}

93
|> Splitter.add-split @\{thm prod.split\}

THEN' TRY o ( 
  REPEAT-ALL-NEW (match-tac ctxt @\{thms allI impI\})
  THEN' simp-only ctxt @\{thms Product-Type.split prod.inject\}
)

THEN' TRY o REPEAT-ALL-NEW (ematch-tac ctxt @\{thms conjE\})
THEN' TRY o hyp-subst-tac ctxt
THEN' simp-only ctxt @\{thms triv-forall-equality\}
THEN' (  
  resolve-tac ctxt @\{thms hn-refine-cons[rotated]\}
  THEN' (resolve-tac ctxt [thm] THEN-ALL-NEW assume-tac ctxt))
THEN-ALL-NEW simp-only ctxt
  @\{thms hn-ctxt-def enth-refl pure-unit-rel-eq-empty
       mult-ac mult-1 mult-1-right keep-drop-sels\}

(* Prove theorem *)
val result = Thm.cterm-of ctxt result
val rthm = Goal.prove-internal ctxt [] result (fn - => ALLGOALS (tac ctxt))

(* Export statement to original context *)
val rthm = singleton (Variable.export ctxt orig-ctxt) rthm

(* Post-processing *)
val rthm = Local-Defs.unfold0 ctxt (Named-Theorems.get ctxt @\{named-theorems to-hfref-post\}) rthm

in
  rthm
end

fun to-hfref ctxt = analyze-hnr ctxt => mk-hfref-thm ctxt


***************
(* Composition *)

local
  fun norm-set-of ctxt = 
    trans-rules = Named-Theorems.get ctxt @\{named-theorems fcomp-norm-trans\},
    cong-rules = Named-Theorems.get ctxt @\{named-theorems fcomp-norm-cong\},
    norm-rules = Named-Theorems.get ctxt @\{named-theorems fcomp-norm-norm\},
    refl-rules = Named-Theorems.get ctxt @\{named-theorems fcomp-norm-refl\}


fun init-rules-of ctxt = Named-Theorems.get ctxt @\{named-theorems fcomp-norm-init\}
fun unfold-rules-of ctxt = Named-Theorems.get ctxt @\{named-theorems fcomp-norm-unfold\}
fun simp-rules-of ctxt = Named-Theorems.get ctxt @\{named-theorems fcomp-norm-simps\}

in
fun norm-fcomp-rule ctxt = let
  open PO-Normalizer Refine-Util
  val norm1 = gen-norm-rule (init-rules-of ctxt) (norm-set-of ctxt) ctxt
  val norm2 = Local-Defs.unfold0 ctxt (unfold-rules-of ctxt)
  val norm3 = Conv.fconv-rule (Simplifier.asm-full-rewrite (put-simpset HOL-basic-ss ctxt addsimps simp-rules-of ctxt))
  val norm = changed-rule (try-rule norm1 o try-rule norm2 o try-rule norm3)
  in
  repeat-rule norm
  end end

fun add-pure-constraints-rule ctxt thm = let
  val orig-ctxt = ctxt
  val t = Thm.prop-of thm
  fun cnv (@\{mpat (typts) pure (mpaq-STRUCT (mpaq-Var ?x -) :: (?'v-c×?v-a) set\}) =
    let
      val T = a -- c -- @\{typ assn\}
      val t = Var (x, T)
      val t = @\{mk-term (the-pure ?t)\}
      in
        [(x, T, t)]
      end
    | cnv (t$u) = union op= (cnv t) (cnv u)
    | cnv (Abs (_, _, t)) = cnv t
    | cnv - = []
  val pvars = cnv t
  in
    val - = (pvars |> map #1 |> has-duplicates op=)
    andalso raise TERM (Duplicate indexname with different type,[t]) (* This should not happen *)
    val substs = map (fn (x,_,t) => (x,t)) pvars
  end
val \( t' = \text{subst-Vars} \ \text{substs} \ t \)

fun mk-asn \((x,T,\_)=\) let
val \( t = \text{Var} \ (x,T) \)
val \( t = @\{\text{mk-term Trueprop} \ (\text{CONSTRAINT} \ \text{is-pure} \ ?t)\} \) in
\( t \) end

val assms = \text{map} \ \text{mk-asn} \ \text{pvars} \)

fun add-prems \text{prems} \ t = let
val \( \text{prems'} = \text{Logic} \cdot \text{strip-imp-prems} \ t \)
val \( \text{concl} = \text{Logic} \cdot \text{strip-imp-concl} \ t \) in
\( \text{Logic} \cdot \text{list-implies} \ (\text{prems} @ \text{prems'}, \ \text{concl}) \) end

val \( t' = \text{add-prems} \ \text{assms} \ t \)

val \((t',\text{ctxt}) = \text{yield-singleton} \ (\text{Variable} \cdot \text{import-terms} \ \text{true}) \ t' \ \text{ctxt} \)

val thm' = \text{Goal} \cdot \text{prove-internal} \ \text{ctxt} [] \ (\text{Thm} \cdot \text{cterm-of} \ \text{ctxt} \ t') \ (\text{fn} \ - \to \ \text{ALLGOALS} \ (\text{resolve-tac} \ \text{ctxt} \ [\text{thm}] \ \text{THEN-ALL-NEW} \ \text{assume-tac} \ \text{ctxt})) \)

val thm' = \text{norm-fcomp-rule} \ \text{ctxt} \ \text{thm}'

val thm' = \text{singleton} \ (\text{Variable} \cdot \text{export} \ \text{ctxt} \ \text{orig-ctxt}) \ \text{thm}'\) in
\( \text{thm}' \) end

val \text{cfg-simp-precond} =
\text{Attrib} \cdot \text{setup-config-bool} @\{\text{binding fcomp-simp-precond} \} \ (\text{K} \ \text{true})

local
fun mk-simp-thm \ \text{ctxt} \ t = let
val \( st = t \)
\( | > \text{HOLogic.mk-Trueprop} \)
\( | > \text{Thm} \cdot \text{cterm-of} \ \text{ctxt} \)
\( | > \text{Goal} \cdot \text{init} \)
val \( \text{ctxt} = \text{Context-Position} \cdot \text{set-visible} \ \text{false} \ \text{ctxt} \)
val \( \text{ctxt} = \text{ctxt} \ \text{addsimps} \ (\) \text{refine-pw-simps} \cdot \text{get} \ \text{ctxt} \)
\( @ \text{Named-Theorems} \cdot \text{get} \ \text{ctxt} @\{\text{named-theorems} \ \text{fcomp-prenorm-simps} \}
\) @\{\text{thms} \ \text{split-tupled-all} \ \text{cnv-conj-to-meta} \}
\) end

96
val trace-incomplete-transfer-tac =
COND (Thm.prems-of #> exists (strip-all-body #> Logic.strip-imp-concl
#> Term.is-open))
(print-tac ctxt Failed transfer from intermediate level:) all-tac

val tac =
ALLGOALS (resolve-tac ctxt @\{thms auto-weaken-pre-comp-PRE-I\} )
THEN ALLGOALS (Simplifier.asm-full-simp-tac ctxt)
THEN trace-incomplete-transfer-tac
THEN ALLGOALS (TRY o filter-prems-tac ctxt (K false))
THEN Local-Defs.unfold0-tac ctxt [Drule.triv-forall-equality]

val st′ = tac st |> Seq.take 1 |> Seq.list-of
val thm = case st′ of [st′] => Goal.conclude st′ | => raise THM(Simp-Precond:
Simp-Tactic failed,~1,[st])

(* Check generated premises for leftover intermediate stuff *)
val - = exists (Logic.is-all) (Thm.prems-of thm)
andalso raise THM(Simp-Precond: Transfer from intermediate level
failed,~1,[thm])

val thm =
thm
(*=> map (Simplifier.asm-full-simplify ctxt)*)
|=> Conv.fconv-rule (Object-Logic.atomize ctxt)
|=> Local-Defs.unfold0 ctxt @\{thms auto-weaken-pre-to-imp-nf\}

val thm = case Thm.concl-of thm of
|@{mpat Trueprop (- → -)} => thm
|@{mpat Trueprop -} => thm RS @{thm auto-weaken-pre-add-dummy-imp}
| => raise THM(Simp-Precond: Generated odd theorem, expected form
'P→Q',~1,[thm])

in
thm
end
in

fun simplify-precond ctxt thm = let
val orig-ctxt = ctxt
val thm = Refine-Util.OF-fst @\{thms auto-weaken-pre-init\} [asm-rl,thm]
val thm =
Local-Defs.unfold0 ctxt @\{thms split-tupled-all\} thm
OF @\{thms auto-weaken-pre-uncurry-start\}

fun rec-uncurry thm =
case try (fn () => thm OF @\{thms auto-weaken-pre-uncurry-step\}) () of

97
NONE => thm OF @\{thms auto-weaken-pre-uncurry-finish\}
| SOME thm => rec-uncurry thm

val thm = rec-uncurry thm
|> Conv.fconv-rule Thm.eta-conversion

val t = case Thm.prems-of thm of
  t:=- => t | - => raise THM(Simp-Precond: Expected at least one premise,¬1,\[thm\])

val (t,ctxt) = yield-singleton (Variable.import-terms false) t ctxt
val ((-),ctxt) = Variable.focus NONE t ctxt
val t = case t of
  @\{mpat Trueprop (- → ?t)} => t | - => raise TERM(Simp-Precond:
  Expected implication,[t])

val simpthm = mk-simp-thm ctxt t
|> singleton (Variable.export ctxt orig-ctxt)

val thm = thm OF [simpthm]
val thm = Local-Defs.unfold0 ctxt @\{thms prod-casesK\} thm
  in
  thm
end

fun simplify-precond-if-cfg ctxt =
  if Config.get ctxt cfg-simp-precond then
    simplify-precond ctxt
  else I
end

(* fref O fref *)
fun compose-ff ctxt A B =
  (@\{thm fref-compI-PRE\} OF [A,B])
|> norm-fcomp-rule ctxt
|> simplify-precond-if-cfg ctxt
|> Conv.fconv-rule Thm.eta-conversion

(* href O fref *)
fun compose-hf ctxt A B =
  (@\{thm href-compI-PRE\} OF [A,B])
|> norm-fcomp-rule ctxt
|> simplify-precond-if-cfg ctxt
|> Conv.fconv-rule Thm.eta-conversion
|> add-pure-constraints-rule ctxt
|> Conv.fconv-rule Thm.eta-conversion

fun ensure-fref ctxt thm = case rthm-type thm of
fun ensure-fref-nres ctxt thm = let
  val thm = ensure-fref ctxt thm
  in
  case Thm.concl-of thm of
    @{mpat (typs) Trueprop (-∈fref - - (- nres×-)set))} => thm
  | @{mpat Trueprop ((-,-)∈fref - -)} =>
      (thm RS @{thm ensure-fref-nresI}) |> Local-Defs.unfold0 ctxt (thms ensure-fref-nres-unfold)
  | - => raise THM(Expected fref − theorem,~1,[thm])
  end

fun ensure-hfref ctxt thm = case rthm-type thm of
  RT-HNR => to-hfref ctxt thm
  | RT-HFREF => thm
  | - => raise THM(Expected hnr or hfref theorem,~1,[thm])

fun ensure-hnr ctxt thm = case rthm-type thm of
  RT-HNR => thm
  | RT-HFREF => to-hnr ctxt thm
  | - => raise THM(Expected hnr or hfref theorem,~1,[thm])

fun gen-compose ctxt A B = let
  val rtA = rthm-type A
  in
  if rtA = RT-HOPARAM orelse rtA = RT-FREF then
    compose-ff ctxt (ensure-fref ctxt A) (ensure-fref ctxt B)
  else
    compose-hf ctxt (ensure-hfref ctxt A) ((ensure-fref-nres ctxt B))
  end

val parse-fcomp-flags = Refine-Util.parse-paren-lists
  (Refine-Util.parse-bool-config prenorm cfg-simp-precond)

val fcomp-attrib = parse-fcomp-flags |> Attrib.thm |> (fn B => Thm.rule-attribute
  (fn context => fn A =>>
    let
      val ctxt = Context.proof-of context
      in
        gen-compose ctxt A B
      end))
1.6 Setup for Combinators

theory Sepref-Combinator-Setup
imports Sepref-Rules Sepref-Monadify
keywords sepref-register :: thy-decl
  and sepref-decl-intf :: thy-decl
begin
1.6.1 Interface Types

This tool allows the declaration of interface types. An interface type is a new type, and a rewriting rule to an existing (logic) type, which is used to encode objects of the interface type in the logic.

context begin

private definition $T :: \text{string} \Rightarrow \text{unit list} \Rightarrow \text{unit}$ where $T \cdot \cdot \equiv ()$

private lemma unit-eq: $(a::\text{unit}) \equiv b$ by simp

named-theorems --itype-rewrite

ML (signature SEPREF-INTF-TYPES = sig

(* Declare new interface type *)
val decl-intf-type-cmd: ((string list * binding) * mixfix) * string -> local-theory

(* Register interface type rewrite rule *)
val register-itype-rewrite: typ -> typ -> Proof.context -> local-theory

(* Convert interface type to logical type*)
val norm-intf-type: Proof.context -> typ -> typ

(* Check whether interface type matches operation's type *)
val check-intf-type: Proof.context -> typ -> typ -> bool

(* Invoke msg with (normalized) non-matching types in case of no-match *)
val check-intf-type-msg: (typ * typ -> unit) -> Proof.context -> typ -> typ

(* Trigger error message if no match *)
val check-intf-type-err: Proof.context -> typ -> typ -> unit

dv.

end

structure Sepref-Intf-Types: SEPREF-INTF-TYPES = struct

fun t2t (Type(name, args)) =
  @{term T}
  $\text{HOLogic.mk-string name}$
  $\text{HOLogic.mk-list @{typ unit}}$ (map t2t args)
| t2t (TFree (name, -)) = Var ((F name,0),HOLogic.unitT)
| t2t (TVar ((name,i),-)) = Var ((V name,i),HOLogic.unitT)

fun tt2 (t as (Var ((name,i),-))) =
  if match-string F* name then TFree (unprefix F name, dummyS)
  else if match-string V* name then TVar ((unprefix V name,i), dummyS)
  else raise TERM(tt2: Invalid var,[t])
| tt2 (@{(map T ?name ?args} = Type (HOLogic.dest-string name, HOLogic.dest-list args |> map tt2)
| tt2 t = raise TERM(tt2: Invalid,[t])

fun mk-t2t-rew ctxt T1 T2 = let
  fun chk-vars T = exists-subtype is-TVar T andalso raise TYPE(Type must
not contain schematics, [T], []

val - = chk-vars T1
val - = chk-vars T2

val free1 = Term.add-frees T1 []
val free2 = Term.add-frees T2 []

val = subset (=) (free2, free1) orelse raise TYPE (Free variables on RHS
must also occur on LHS, [T1, T2], [])

in
    Thm.instantiate' [] [
        t2t T1 |> Thm.cterm-of ctxt |> SOME,
        t2t T2 |> Thm.cterm-of ctxt |> SOME
    ]
    @{thm unit-eq}
end

fun register-itype-rewrite T1 T2 lthy =
    lthy
    |> Local-Theory.note ((Binding.empty, @{attributes [--itype-rewrite]}), [mk-t2t-rew
lthy T1 T2])
    |> #2

val decl-intf-type-parser =
    Parse.type-args -- Parse.binding -- Parse.opt-mixfix -- | @{keyword is}
    -- Parse.typ

fun decl-intf-type-cmd (((args, a), mx), T2-raw) lthy = let
    val (T1, lthy) = Typedecl.typedecl {final = true} (a, map (rpair dummyS)
args, mx) lthy
    val T2 = Syntax.read-typ lthy T2-raw
    in
        register-itype-rewrite T1 T2 lthy
    end

fun norm-intf-typet ctxt T = let
    val rew-rls = Named-Theorems.get ctxt @{named-theorems --itype-rewrite}
    in
        t2t T
        |> Thm.cterm-of ctxt
        |> Drule.mk-term
        |> Local-Defs.unfold0 ctxt rew-rls
        |> Drule.dest-term
        |> Thm.term-of
    end

fun norm-intf-type ctxt T = norm-intf-typet ctxt T |> tth
fun check-intf-type ctxt iT cT = let
  val it = norm-intf-typet ctxt iT
  val ct = t2t cT
  val thy = Proof-Context.theory-of ctxt
in
  Pattern.matches thy (it, ct)
end

fun check-intf-type-msg msg ctxt iT cT = let
  val it = norm-intf-typet ctxt iT
  val ct = t2t cT
  val thy = Proof-Context.theory-of ctxt
  in
  if Pattern.matches thy (it, ct) then ()
    else msg (tt2 it, tt2 ct)
  end
end

fun check-intf-type-err ctxt iT cT = let
  fun msg (iT′, cT′) = Pretty.block [Pretty.str Interface type and logical type do not match,
    Pretty.brk,
    Pretty.str Interface: Syntax.pretty-typ ctxt iT′, Pretty.brk,
    Pretty.str is , Syntax.pretty-typ ctxt iT′, Pretty.brk
    Pretty.str Logical: Syntax.pretty-typ ctxt cT′, Pretty.brk
    Pretty.str is , Syntax.pretty-typ ctxt cT′, Pretty.brk
    Pretty.string-of | > error]
  in
    check-intf-type-msg msg ctxt iT cT
  end
end

val - = Outer-Syntax.local-theory
  @{command-keyword sepref-decl-intf}
  (decl-intf-type-parser >> decl-intf-type-cmd); end

declaration map-type-eq :: 'a itself ⇒ 'b itself ⇒ bool
  (infixr ➔nt 60)
  where [simp]: map-type-eq - - ≡ True
lemma map-type-eql: map-type-eq L R by auto

named-theorems-rev map-type-eqs

1.6.2 Rewriting Inferred Interface Types
1.6.3 ML-Code

context begin

private lemma start-eval: x ≡ SP x by auto
private lemma add-eval: f x ≡ (⇒)$($EVAL$x)$($λ_2x. f x) by auto

private lemma init-mk-arity: f ≡ id (SP f) by simp
private lemma add-mk-arity: id f ≡ (λ_2x. id (f$x)) by auto
private lemma finish-mk-arity: id f ≡ f by simp

ML :
structure Sepref-Combinator-Setup = struct

(* Check whether this term is a valid abstract operation *)
fun is-valid-abs-op - (Const _) = true
| is-valid-abs-op ctxt (Free (name,)) = Variable.is-fixed ctxt name
| is-valid-abs-op - @{mpat PR-CONST -} = true
| is-valid-abs-op - _ = false

fun mk-itype ctxt t tyt =
let
  val cert = Thm.cterm-of ctxt
  val t = cert t
  val tyt = cert tyt
in
  Drule.infer-instantiate' ctxt [SOME t, SOME tyt] @{thm itypeI}
end

(* Generate mcomb-theorem, required for monadify transformation. t$x1$..$xn = x1←EVAL x1; ...; xn←EVAL xn; SP (t$x1$..$xn) *)
fun mk-mcomb ctxt t n =
let
  val T = fastype-of t
  val (argsT, _) = strip-type T
  val _ = length argsT >= n orelse raise TERM(Too few arguments,[t])
  val effT = take n argsT

  val orig-ctxt = ctxt
  val names = map (fn i => xˆstring-of-int i) (1 upto n)
  val (names,ctxt) = Variable.variant-fixes names ctxt
  val vars = map Free (names ~ effT)

  val lhs = Autoref-Tagging.list-APP (t,vars)
|> Thm.cterm-of ctxt

fun add-EVAL x thm =
  case Thm.prop-of thm of
    @{mpat _ ≡ ?rhs} => let
      val f = lambda x rhs |> Thm.cterm-of ctxt
      val x = Thm.cterm-of ctxt x

val eval-thm = Drule.infer-instantiate' ctxt
[SOME f, SOME x] @{thm add-eval}
val thm = @{thm transitive} OF [thm, eval-thm]
in thm end
| - => raise THM (mk-mcomb internal: Expected lhs==rhs, ~1,[thm])

val thm = Drule.infer-instantiate' ctxt [SOME lhs] @{thm start-eval}
val thm = fold add-EVAL (rev vars) thm
val thm = singleton (Proof-Context.export ctxt orig-ctxt) thm
in
thm
end;

(* Generate arity-theorem: t = lam x1...xn. SP t$1$...$n
*)
fun mk-arity ctxt t n = let
val t = Thm.cterm_of ctxt t
val thm = Drule.infer-instantiate' ctxt [SOME t] @{thm init-mk-arity}
val add-mk-arity = Conv.fconv-rule (
  Refine-Util.ftop-conv (K (Conv.rewr-conv @{thm add-mk-arity})) ctxt)
val thm = funpow n add-mk-arity thm
val thm = Conv.fconv-rule (
  Refine-Util.ftop-conv (K (Conv.rewr-conv @{thm finish-mk-arity})) ctxt)

thm
in
thm
end;

datatype opkind = PURE | COMB

fun analyze-decl c tyt = let
  fun add-tcons-of (Type (name, args)) l = fold add-tcons-of args (name::l)
      | add-tcons-of - l = l

  fun all-tcons-of P T = forall P (add-tcons-of T [])

val T = Logic.dest-type tyt
val (argsT,resT) = strip-type T

val = forall (all-tcons-of (fn tn => tn <> @{type-name nres})) argsT
  orelse raise TYPE (Arguments contain nres - type
    (currently not supported by this attribute),
    argsT,[e,tty])

val kind = case resT of
  Type (@{(type-name nres)},-) => COMB
fun analyze-itype-thm thm =
  case Thm.prop_of thm of
    @{mpat (typs) Trueprop (intf-type \?c (\$:\?'v-T itself))} => let
      val tyt = Logic.mk-type T
      val (kind,(argsT,resT)) = analyze-decl c tyt
      in
      PURE
    end
  in (kind,(argsT,resT)) end

(*fun register-combinator itype-thm context = let
  val ctxt = Context.proof_of context
  val (t,kind,(argsT,-)) = analyze-itype-thm itype-thm
  val n = length argsT
  in
  case kind of
    PURE => context
    | > Named-Theorems-Rev.add-thm @{named-theorems-rev id-rules} itype-thm
    | COMB => let
        val arity-thm = mk-arity ctxt t n
        (*val skel-thm = mk-skel ctxt t n*)
        val mcomb-thm = mk-mcomb ctxt t n
        in
        context
        | > Named-Theorems-Rev.add-thm @{named-theorems-rev id-rules} itype-thm
        | > Named-Theorems-Rev.add-thm @{named-theorems-rev sepref-monadify-arity} arity-thm
        | > Named-Theorems-Rev.add-thm @{named-theorems-rev sepref-monadify-comb} mcomb-thm
      (*| > Named-Theorems-Rev.add-thm @{named-theorems-rev sepref-la-skel} skel-thm*)
    end
  end
  end
  end

fun generate-basename ctxt t = let
  fun fail () = raise TERM (Basename generation heuristics failed. Specify a basename..\[t]\)
  val - = all-tcons-of (fn tn => tn <> @{type-name nres}) T 
  orelse raise TYPE ( 
    Result contains inner nres - type,
    argsT,[c,tyt])
  in
  PURE
end
fun gb (Const (n,-)) =
  (* TODO: There should be a cleaner way than handling this on string level! *)
  n |> space-explode . |> List.last
  | gb (@{mput PR-CONST ?t}) = gb t
  | gb (t as (-$-)) = let
    val h = head-of t
    val - = is-Const h orelse is-Free h orelse fail ()
  in
    gb h
  end
  | gb (Free (n,-)) =
    if Variable.is-fixed ctxt n then n
    else fail ()
  in
    gb t
  end

fun map-type-raw ctxt rls T =
  let
    val thy = Proof-Context.theory-of ctxt
  fun rewr-this (lhs,rhs) T =
    let
      val env = Sign.typ-match thy (lhs,T) Vartab.empty
    in
      Envir.norm-type env rhs
    end
  fun map-Targs f (Type (name,args)) = Type (name,map f args)
    | map-Targs - T = T
  fun rewr-thiss (r::rls) T =
    (SOME (rewr-this r T) handle Type.TYPE-MATCH => rewr-thiss rls T)
    | rewr-thiss [] - = NONE
  fun map-type-aux T =
    let
      val T = map-Targs map-type-aux T
    in
      case rewr-thiss rls T of
        SOME T => map-type-aux T
        NONE => T
    end
    in
      map-type-aux T
  end
  fun get-nt-rule thm = case Thm.prop-of thm of

107
let
val Lvars = Term.add-tvar-names T L []
val Rvars = Term.add-tvar-names T R []

val - = subset (\) (Rvars, Lvars) or else (
  let
    val frees = subtract (\) Lvars Rvars
    |\> map (Term.string-of-name)
    |\> Pretty.str-list []
    |\> Pretty.string-of
  in
    raise THM (Free variables on RHS: frees,\(1,\[thm\])
  end)

  in
  (L,R)
end

| - => raise THM (No map-type-eq theorem,\(1,\[thm\])

fun map-type ctxt T = let
  val rls =
  Named-Theorems-Rev.get ctxt @{\{named-theorems-rev map-type-eqs\}
  |\> map get-nt-rule
  in map-type-raw ctxt rls T end

fun read-term-type ts tys lthy = case tys of
SOME ty => let
  val ty = Syntax.read-typ lthy ty
  val ctxt = Variable.declare-typ ty lthy
  val t = Syntax.read-term ctxt ts
  val ctxt = Variable.declare-term t ctxt
  in
    ((t, ty), ctxt)
  end
| NONE => let
  val t = Syntax.read-term lthy ts
  val ctxt = Variable.declare-term t lthy

  val tgyt = fastype-of t |> map-type ctxt |> Logic.mk-type

  val tgyt = tgyt |> singleton (Variable.export-terms ctxt lthy)
  val (tgyt, ctxt) = yield-singleton (Variable.import-terms true) tgyt ctxt
  val ty = Logic.dest-type tgyt
  in
    ((t, ty), ctxt)
  end
fun check-type-intf ctxt Tc Ti = let

fun type2term (TFree (name,-)) = Var ((F{name},0),HOLogic.unitT)
| type2term (TVar ((name,i),-)) = Var ((V{name},i),HOLogic.unitT)
| type2term (Type (@{type-name fun}[T1,T2])) = 
  Free (F,HOLogic.unitT ---> HOLogic.unitT ---> HOLogic.unitT)
$\$type2term T1$\$type2term T2 
| type2term (Type (name,argsT)) = let
  val args = map type2term argsT
  val n = length args
  val T = replicate n HOLogic.unitT ---> HOLogic.unitT
  val v = Var ((T{name},0),T)
in list-comb (v, args) end

val c = type2term Tc
val i = type2term Ti
val thy = Proof-Context.theory-of ctxt
in Pattern.matches thy (i,c)
end

(* Import all terms into context, with disjoint free variables *)
fun import-terms-disj ts ctxt = let

fun exp ctxt t = let
  val new-ctxt = Variable.declare-term t ctxt
  val t = singleton (Variable.export-terms new-ctxt ctxt) t
  in t end

val ts = map (exp ctxt) ts

fun cons-fst f a (l,b) = let val (a,b) = f a b in (a::l,b) end

val (ts,ctxt) = fold-rev (cons-fst (yield-singleton (Variable.import-terms true)))
  ts ([],ctxt)
in (ts,ctxt)
end

type reg-thms = {
  type-thm: thm,
  arity-thm: thm option,
  mcomb-thm: thm option
}

fun cr-reg-thms t ty ctxt = let

val orig-ctxt = ctxt
val tyt = Logic.mk-type ty
val ([(t,tyt),ctxt]) = import-terms-disj [t,tyt] ctxt

val (kind,(argsT,-)) = analyze-decl t tyt

val c = type2term Tc
val i = type2term Ti
val thy = Proof-Context.theory-of ctxt
in Pattern.matches thy (i,c)
end

(* Import all terms into context, with disjoint free variables *)
fun import-terms-disj ts ctxt = let

fun exp ctxt t = let
  val new-ctxt = Variable.declare-term t ctxt
  val t = singleton (Variable.export-terms new-ctxt ctxt) t
  in t end

val ts = map (exp ctxt) ts

fun cons-fst f a (l,b) = let val (a,b) = f a b in (a::l,b) end

val (ts,ctxt) = fold-rev (cons-fst (yield-singleton (Variable.import-terms true)))
  ts ([],ctxt)
in (ts,ctxt)
end

type reg-thms = {
  type-thm: thm,
  arity-thm: thm option,
  mcomb-thm: thm option
}

fun cr-reg-thms t ty ctxt = let

val orig-ctxt = ctxt
val tyt = Logic.mk-type ty
val ([(t,tyt),ctxt]) = import-terms-disj [t,tyt] ctxt

val (kind,(argsT,-)) = analyze-decl t tyt

val c = type2term Tc
val i = type2term Ti
val thy = Proof-Context.theory-of ctxt
in Pattern.matches thy (i,c)
end
val n = length argsT

val - = Sepref-Intf-Types.check-intf-type-err ctxt ty (fastype-of t)

val - = is-valid-abs-op ctxt t
  orelse raise TERM(Malformed abstract operation. Use PR-CONST for
complex terms.[t])

val itype-thm = mk-itype ctxt t tyt
  |> singleton (Variable.export ctxt orig-ctxt)
in
  case kind of
    PURE => {itype-thm = itype-thm, arity-thm = NONE, mcomb-thm =
      NONE}
    | COMB => let
      val arity-thm = mk-arity ctxt t n
      |> singleton (Variable.export ctxt orig-ctxt)
      val mcomb-thm = mk-mcomb ctxt t n
      |> singleton (Variable.export ctxt orig-ctxt)
in
      {itype-thm = itype-thm, arity-thm = SOME arity-thm, mcomb-thm =
        SOME mcomb-thm}
    end
end

fun gen-pr-const-pat ctxt t =
if is-valid-abs-op ctxt t then (NONE, t)
else
  let
    val ct = Thm.cterm-of ctxt t
    val thm = Drule.infer-instantiate' ctxt [SOME ct] \{thm UNPROTECT-def[symmetric]\}
      |> Conv.fconv-rule (Conv.arg1-conv (Id-Op.protect-conv ctxt))
in
    (SOME thm, \{mk-term PR-CONST ?t\})
  end

fun sepref-register-single basename t ty lthy = let
  fun mk-qualified basename q = Binding.qualify true basename (Binding.name q);
  val pat-thm, t = gen-pr-const-pat lthy t
  val {itype-thm, arity-thm, mcomb-thm} = cr-reg-thms t ty lthy
val lthy = lthy
    |> do-note pat @{attributes [def-pat-rules]} pat-thm
    |> do-note itype @{attributes [id-rules]} (SOME itype-thm)
    |> do-note arity @{attributes [sepref-monadify-arity]} arity-thm
    |> do-note mcomb @{attributes [sepref-monadify-comb]} mcomb-thm

in
  (((arity-thm,mcomb-thm),itype-thm),lthy)
end

fan sepref-register-single-cmd ((basename,ts),tys) lthy = let
  val t = Syntax.read-term lthy ts
  val ty = map-option (Syntax.read-typ lthy) tys
  val ty = case ty of SOME ty => ty | NONE => fastype-of t | map-type lthy
  val basename = case basename of
    NONE =>> generate-basename lthy t
  | SOME n =>> n
  val ((-,itype-thm),lthy) = sepref-register-single basename t ty lthy
  val - = Thy-Output.pretty-thm lthy itype-thm |> Pretty.string-of |>.writeln

in
  lthy
end

val sepref-register-cmd = fold sepref-register-single-cmd

val sepref-register-parser = Scan.repeat1 (Scan.option (Parse.name -- | @{keyword :})
  -- Parse.term
  -- Scan.option (@{keyword :} | -- Parse.typ)
)

val - =
  Outer-Syntax.local-theory
  @{command-keyword sepref-register}
  Register operation for sepref
  ( sepref-register-parser
    >> sepref-register-cmd);

val sepref-register-adhoc-parser = Scan.repeat1 (Args.term -- Scan.option (Scan.lift (Args.$$ ::) | -- Args.typ)
)

fan sepref-register-adhoc-single (t,ty) context = let
  val ctxt = Context.proof-of context
(* TODO: Map-type probably not clean, as it draws info from (current) context, which may have changed if registered elsewhere ... *)

val ty = case ty of SOME ty => ty | NONE => fastype-of t | map-type ctxt

val {itype-thm, arity-thm, mcomb-thm} = cr-reg-thms t ty ctxt

fun app - NONE = I
| app attr (SOME thm) = Thm.apply-attribute attr thm #> snd

in
context
| > app (Named-Theorems-Rev.add @{named-theorems-rev def-pat-rules})
| > app (Named-Theorems-Rev.add @{named-theorems-rev id-rules}) (SOME itype-thm)
| > app (Named-Theorems-Rev.add @{named-theorems-rev sepref-monadify-arity})
| > app (Named-Theorems-Rev.add @{named-theorems-rev sepref-monadify-comb})

mcomb-thm
end

val sepref-register-adhoc = fold sepref-register-adhoc-single

fun sepref-register-adhoc-attr ttys = Thm.declaration-attribute (K (sepref-register-adhoc ttys))

val sepref-register-adhoc-attr-decl = sepref-register-adhoc-parser >> sepref-register-adhoc-attr

end

attribute-setup sepref-register-adhoc = Sepref-Combinator-Setup.sepref-register-adhoc-attr-decl

(*Register operations in ad-hoc manner. Improper if this gets exported!

1.6.4 Obsolete Manual Setup Rules

lemma

mk-mcomb1: \( \lambda c. c \times x1 \equiv (\Rightarrow)(EVAL\times x1)(\lambda x2. SP (c \times x1)) \)

and mk-mcomb2: \( \lambda c. c \times x1 \times x2 \equiv (\Rightarrow)(EVAL\times x1)(\lambda x2. (\Rightarrow)(EVAL\times x2)(\lambda x2. SP (c \times x1 \times x2))) \)

and mk-mcomb3: \( \lambda c. c \times x1 \times x2 \times x3 \equiv (\Rightarrow)(EVAL\times x1)(\lambda x2. (\Rightarrow)(EVAL\times x2)(\lambda x2. (\Rightarrow)(EVAL\times x3)(\lambda x3. SP (c \times x1 \times x2 \times x3)))) \)

by auto

112
This theory defines the translation phase.
The main functionality of the translation phase is to apply refinement rules.
Thereby, the linearity information is exploited to create copies of parameters
that are still required, but would be destroyed by a synthesized operation.
These frame-based rules are in the named theorem collection sepref-fr-rules,
and the collection sepref-copy-rules contains rules to handle copying of parameters.
Apart from the frame-based rules described above, there is also a set of rules
for combinators, in the collection sepref-comb-rules, where no automatic
copying of parameters is applied.
Moreover, this theory contains

- A setup for the basic monad combinators and recursion.
- A tool to import parametricity theorems.
- Some setup to identify pure refinement relations, i.e., those not involving the heap.
- A preprocessor that identifies parameters in refinement goals, and flags them with a special tag, that allows their correct handling.

Tag to keep track of abstract bindings. Required to recover information for
side-condition solving.

**definition** bind-ref-tag x m ≡ RETURN x ≤ m

Tag to keep track of preconditions in assertions

**definition** vassn-tag Γ ≡ ∃ h. h |= Γ

**lemma** vassn-tagI: h |= Γ ⇒ vassn-tag Γ
unfolding vassn-tag-def ..

lemma vassn-dest[dest!]:
vassn-tag (Γ₁ * Γ₂) ⇒ vassn-tag Γ₁ ∧ vassn-tag Γ₂
vassn-tag (hn-ctxt R a b) ⇒ a∈rdom R
unfolding vassn-tag-def rdomp-def[abs-def]
by (auto simp: mod-star-conv hn-ctxt-def)

lemma entails-preI:
  assumes vassn-tag A ⇒ A ⇒ₐ B
  shows A ⇒ₐ B
  using assms
by (auto simp: entails-def vassn-tag-def)

lemma invalid-assn-const:
  invalid-assn (λ_ -_. P) x y = ↑(vassn-tag P) * true
by (simp-all add: invalid-assn-def vassn-tag-def)

lemma vassn-tag-simps[simp]:
  vassn-tag emp
  vassn-tag true
by (sep-auto simp: vassn-tag-def mod-emp+)

definition GEN-ALGO f Φ ≡ Φ f
— Tag to synthesize f with property Φ.

lemma is-GEN-ALGO: GEN-ALGO f Φ ⇒ GEN-ALGO f Φ .
Tag for side-condition solver to discharge by assumption

definition RPREM :: bool ⇒ bool where [simp]: RPREM P = P
lemma RPREMI: P ⇒ RPREM P by simp

lemma trans-frame-rule:
  assumes RECOVER-PURE Γ Γ'
  assumes vassn-tag Γ' ⇒ hn-refine Γ' c Γ'' R a
  shows hn-refine (F*Γ) c (F*Γ'') R a
  apply (rule hn-refine-frame[OF - entt-refl])
applyF (rule hn-refine-cons-pre)
  focus using assms(1) unfolding RECOVER-PURE-def apply assumption
solved
  apply1 (rule hn-refine-preI)
  apply1 (rule assms)
  applyS (auto simp add: vassn-tag-def)
solved
done

lemma recover-pure-cons: — Used for debugging
  assumes RECOVER-PURE Γ Γ'

114
assumes \(hn\text{-}refine \Gamma' c \Gamma'' R a\)
shows \(hn\text{-}refine (\Gamma) c (\Gamma'') R a\)
using trans-frame-rule[where \(F=\mathit{emp}, OF\text{ assms}\)] by simp

— Tag to align structure of refinement assertions for consequence rule

definition CPR-TAG :: \(\text{assn} \Rightarrow \text{assn} \Rightarrow \text{bool}\) where \(\text{simp}:\ CPR-TAG y x \equiv \text{True}\)

lemma CPR-TAG-starI: 
  assumes CPR-TAG \(P1 Q1\)
  assumes CPR-TAG \(P2 Q2\)
  shows CPR-TAG \((P1*P2) (Q1*Q2)\)
  by simp

lemma CPR-tag-ctxtI: CPR-TAG \((\text{hn-ctxt } R x xi) (\text{hn-ctxt } R' x xi)\) by simp

lemma CPR-tag-fallbackI: CPR-TAG \(P Q\) by simp

lemmas CPR-TAG-rules = CPR-TAG-starI CPR-tag-ctxtI CPR-tag-fallbackI

lemma cons-pre-rule: — Consequence rule to be applied if no direct operation rule matches
  assumes CPR-TAG \(P P'\)
  assumes \(P \implies t P'\)
  assumes \(hn\text{-}refine P' c Q R m\)
  shows \(hn\text{-}refine P c Q R m\)
  using assms(2-)
  by (rule hn-refine-cons-pre)

named-theorems-rev sepref-gen-algo-rules ⟨Sepref: \text{Generic algorithm rules}⟩

ML :

structure Sepref-Translate = struct

val cfg-debug = 
Attrib.setup-config-bool @\{binding sepref-debug-translate\} (K false)

val dbg-msg-tac = Sepref-Debugging.dbg-msg-tac cfg-debug

fun gen-msg-analyze t ctxt = let
  val t = Logic.strip-assums-concl t
in 
case t of
  @\{mpat Trueprop ?t\} => (case t of
    @\{mpat - \(\forall A \longrightarrow t\)\} => t-merge
    | @\{mpat - \(\longrightarrow t\)\} => t-frame
    | @\{mpat INDEP - \} => t-indep
    | @\{mpat CONSTRAINT - \} => t-constraint
    | @\{mpat mono-Heap - \} => t-mono
    | @\{mpat PREFER-tag - \} => t-prefer
  

115
fun msg-analyze msg = Sepref-Debugging.msg-from-subgoal msg gen-msg-analyze

fun check-side-conds thm = let
  open Sepref-Basic
  (* Check that term is no binary operator on assertions *)
  fun is-atomic (Const (_, @{typ assn ⇒ assn ⇒ assn})$-$) = false
  | is-atomic - = true

  val is-atomic-star-list = (Expected atoms separated by star, forall is-atomic o strip-star)

  val is-trueprop = (Expected Trueprop conclusion, can HOLogic.dest-Trueprop)

  fun assert t' (msg, p) t = if p t then () else raise TERM (msg, [t', t])

  fun chk-prem t = let
    val assert = assert t
    val assert = assert t

    fun chk (l ∨ r ⇒ t) = (assert is-atomic-star-list l; assert is-atomic-star-list r; assert is-atomic-star-list m)

    fun chk (l ⇒ r) = assert is-atomic-star-list l; assert is-atomic-star-list r

    fun chk _ = ()

    val t = Logic.strip-assums-concl t
    in
      assert is-trueprop t;
      chk (HOLogic.dest-Trueprop t)
    end
  in
    map chk-prem (Thm.prems-of thm)
  end
structure sepref-comb-rules = Named-Sorted-Thms {
  val name = @{binding sepref-comb-rules}
  val description = Sepref: Combinator rules
  val sort = K I
  fun transform - thm = let
    val - = check-side-conds thm
    in
      [thm]
    end
  end
}

structure sepref-fr-rules = Named-Sorted-Thms {
  val name = @{binding sepref-fr-rules}
  val description = Sepref: Frame-based rules
  val sort = K I
  fun transform context thm = let
    val ctxt = Context.proof-of context
    val thm = Sepref-Rules.ensure-hnr ctxt thm
       |> Conv.fconv-rule (Sepref-Frame.align-rl-conv ctxt)
    val - = check-side-conds thm
    val - = case try (Sepref-Rules.analyze-hnr ctxt) thm of
      NONE =>
        (Pretty.block [
          Pretty.str hnr-analysis failed,
          Pretty.str :,
          Pretty.brk 1,
          Thm.pretty-thm ctxt thm])
       |> Pretty.string-of |
       |> error
      SOME ana =>
        let
          val - = Sepref-Combinator-Setup.is-valid-abs-op ctxt (fst (#ahead ana))
          in
            Pretty.block [
              Pretty.str Invalid abstract head:,
              Pretty.brk 1,
              Pretty.enclose ( ) [Syntax.pretty-term ctxt (fst (#ahead ana))],
              Pretty.brk 1,
              Pretty.str in thm,
              Pretty.brk 1,
              Thm.pretty-thm ctxt thm
            ]
        in
            Pretty.string-of |
            |> error
        in
          () end
        in
          [thm]
        end
  end
}

(***** Side Condition Solving *****)
local
  open Sepref-Basic
in

fun side-unfold-tac ctxt = let
  (*val ctxt = put-simpset HOL-basic-ss ctxt
    addsimps sepref-prep-side-simps.get ctxt*)
in
  CONVERSION (Id-Op.unprotect-conv ctxt)
  THEN' SELECT-GOAL (Local-Defs.unfold0-tac ctxt @{thms bind-ref-tag-def})
  (*THEN' asm-full-simp-tac ctxt*)
end

fun side-fallback-tac ctxt = side-unfold-tac ctxt THEN' TRADE (SELECT-GOAL o auto-tac) ctxt

val side-frame-tac = Sepref-Frame.frame-tac side-fallback-tac
val side-merge-tac = Sepref-Frame.merge-tac side-fallback-tac
fun side-constraint-tac ctxt = Sepref-Constraints.constraint-tac ctxt
fun side-mono-tac ctxt = side-unfold-tac ctxt THEN' TRADE Pf-Mono-Prover.mono-tac ctxt

fun side-gen-algo-tac ctxt =
  side-unfold-tac ctxt
  THEN' resolve-tac ctxt (Named-Theorems-Rev.get ctxt @{named-theorems-rev sepref-gen-algo-rules})

fun side-pref-def-tac ctxt =
  side-unfold-tac ctxt
  THEN' TRADE (fn ctxt =>
    resolve-tac ctxt @{thms PREFER-tagI DEFER-tagI}
    THEN' (Sepref-Debugging.warning-tac' Obsolete PREFER/DEFER side condition ctxt THEN' Tagged-Solver.solve-tac ctxt))

fun side-rprem-tac ctxt =
  resolve-tac ctxt @{thms RPREMI} THEN' Refine-Util.rprems-tac ctxt
  THEN' (K (smash-new-rule ctxt))

(* Solve side condition, or invoke hnr-tac on hn-refine goal. In debug mode, side-condition solvers are allowed to not completely solve the side condition, but must change the goal. *)

fun side-cond-dispatch-tac dbg hnr-tac ctxt = let
  fun MK tac = if dbg then CHANGED o tac ctxt else SOLVED' (tac ctxt)
val t-merge = MK side-merge-tac
val t-frame = MK side-frame-tac
val t-indep = MK Indep-Vars.indep-tac
val t-constraint = MK side-constraint-tac
val t-mono = MK side-mono-tac
val t-pref-def = MK side-pref-def-tac
val t-rprem = MK side-rprem-tac
val t-gen-algo = side-gen-algo-tac ctxt
val t-fallback = MK side-fallback-tac

in

WITH-concl (fn @\{mpat Trueprop ?t\} => (case t of
  @\{mpat - \∨ - \rightarrow\_\_\_\} => t-merge
| @\{mpat - \rightarrow\_\_\_\_\} => t-frame
| @\{mpat - \rightarrow\_\_\_\} => Sepref-Debugging.warning-tac' Old-style frame
side condition ctxt THEN' (K no-tac)
  @\{mpat INDEP -\} => t-indep    (* TODO: Get rid of this!? *)
| @\{mpat CONSTRAINT -\_\} => t-constraint
| @\{mpat mono-Heap -\} => t-mono
| @\{mpat PREFER-tag -\} => t-pref-def
| @\{mpat DEFER-tag -\} => t-pref-def
| @\{mpat RPREM -\} => t-rprem
| @\{mpat GEN-ALGO -\} => t-gen-algo
| @\{mpat hn-refine - - - - -\} => hnr-tac
| - => t-fallback
)
| - => K no-tac
)
end

(***** Main Translation Tactic *)
local
open Sepref-Basic STactical

(* ATTENTION: Beware of evaluation order, as some initialization operations
on context are expensive, and should not be repeated during proof search! *)
in

(* Translate combinator, yields new translation goals and side conditions
which must be processed in order. *)
fun trans-comb-tac ctxt = let
  val comb-rl-net = sepref-comb-rules.get ctxt
  |> Tactic.build-net

in
DETERM o (resolve-from-net-tac ctxt comb-rl-net
ORELSE' (Sepref-Frame.norm-goal-pre-tac ctxt
THEN' resolve-from-net-tac ctxt comb-rl-net
)
)
end

(* Translate operator. Only succeeds if it finds an operator rule such that all resulting side conditions can be solved. Takes the first such rule.

In debug mode, it returns a sequence of the unsolved side conditions of each applicable rule. *)
fun gen-trans-op-tac dbg ctxt = let
val fr-rl-net = sepref-fr-rules.get ctxt |> Tactic.build-net
val fr-rl-tac = resolve-from-net-tac ctxt fr-rl-net (* Try direct match *)
ORELSE' (Sepref-Frame.norm-goal-pre-tac ctxt (* Normalize precondition *)
THEN' (resolve-from-net-tac ctxt fr-rl-net
ORELSE' (resolve-tac ctxt @{thms cons-pre-rule} (* Finally, generate a frame condition *)
THEN-ALL-NEW-LIST [SOLVED' (REPEAT-ALL-NEW-FWD (DETERM o resolve-tac ctxt @{thms CPR-TAG-rules})),
K all-tac, (* Frame remains unchanged as first goal, even if fr-rl creates side-conditions *)
resolve-from-net-tac ctxt fr-rl-net
]
)
)

val side-tac = REPEAT-ALL-NEW-FWD (side-cond-dispatch-tac false (K no-tac) ctxt)

val fr-tac = if dbg then (* Present all possibilities with (partially resolved) side conditions *)
fr-rl-tac THEN-ALL-NEW-FWD (TRY o side-tac)
else (* Choose first rule that solves all side conditions *)
DETERM o SOLVED' (fr-rl-tac THEN-ALL-NEW-FWD (SOLVED' side-tac))

in
PHASES' [  (Align goal, Sepref-Frame.align-goal-tac, 0),  (Frame rule fn ctxt => resolve-tac ctxt @ {thms trans-frame-rule}, 1),  (* RECOVER-PURE goal *)  (Recover pure, Sepref-Frame.recover-pure-tac, ~1),  (* hn-refine goal with stripped precondition *)  (Apply rule, K fr-tac, ~1) ] (flag-phases-ctrl dbg) ctxt
end

(* Translate combinator, operator, or side condition. *)
fun gen-trans-step-tac dbg ctxt = side-cond-dispatch-tac dbg
(trans-comb-tac ctxt ORELSE' gen-trans-op-tac dbg ctxt)
ctxt
val trans-step-tac = gen-trans-step-tac false
val trans-step-keep-tac = gen-trans-step-tac true

fun gen-trans-tac dbg ctxt = PHASES' [  (Translation steps, REPEAT-DETERM' o trans-step-tac, ~1),  (Constraint solving, fn ctxt => fn - => Sepref-Constraints.process-constraint-slot ctxt, 0) ] (flag-phases-ctrl dbg) ctxt
val trans-tac = gen-trans-tac false
val trans-keep-tac = gen-trans-tac true

end

val setup = I
  #=> sepref-fr-rules.setup
  #=> sepref-comb-rules.setup

end

setup Sepref-Translate.setup

Basic Setup

lemma hn-pass[sepref-fr-rules];
sshows hn-refine (hn-ctxt P x x') (return x') (hn-invalid P x x') P (PASS$x)
apply rule apply (sep-auto simp: hn-ctxt-def invalidate-clone')
done
lemma $hn$-bind[sepref-comb-rules]:
assumes $D1$: $hn$-refine $\Gamma$ $m'$ $\Gamma$ $R$ $m$
assumes $D2$:
\[ \forall x'. \ binds-ref-tag x m \implies\]
$hn$-refine $(\Gamma \ast \hn$-ctxt $Rh$ $x$ $x')$ $(f' \ x')$ $(\Gamma 2 \ x \ x')$ $R$ $(f \ x)$
assumes $IMP$: $\forall x \ x'. \Gamma 2 \ x \ x' \implies \Gamma' \ast \hn$-ctxt $Rx$ $x'$
shows $hn$-refine $\Gamma$ $(m' \approx f')$ $\Gamma'$ $R$ (Refine-Basic.$\ binds\$$(\lambda_2 x. \ f \ x)$)
using $assms$
unfolding APP-def PROTECT2-def bind-ref-tag-def
by (rule $hn$-bind)

lemma $hn$-RECT'[sepref-comb-rules]:
assumes $INDEP$ $Ry$ $INDEP$ $Rx$ $INDEP$ $Rx'$
assumes $FR$: $P \implies \Gamma, \hn$-ctxt $Rx$ $ax$ $px$ $* \ F$
assumes $S$: $\forall cf \ of \ ax \ px.\ []$
$\forall ax \ px. \ binds-ref-tag (\hn$-ctxt $Rx$ $ax$ $px$ $* \ F$) $(cf \ px)$ (\hn$-ctxt $Rx'$ $ax$ $px$ $* \ F$) $Ry$
$\ (RCALL$sq$ax$)$
$\implies \ binds-ref-tag (\hn$-ctxt $Rx$ $ax$ $px$ $* \ F$) $(cB \ cf \ px)$ $(F' \ ax \ px)$ $Ry$
$\ (aB \ of \ ax)$
assumes $FR'$: $\forall ax \ px. \ F' \ ax \ px \implies \Gamma, \hn$-ctxt $Rx'$ $ax$ $px$ $* \ F$
assumes $M$: $\forall x. \ binds-ref-tag (\lambda f. \ cB \ f \ x))$
shows $hn$-refine
$(P)$ $(heap,fixp-fun \ cB \ px)$ $(\hn$-ctxt $Rx'$ $ax$ $px$ $* \ F)$ $Ry$
$\ (RECT$sq$D \ x \ aB \ D \ x)\ !ax$)
unfolding APP-def PROTECT2-def
apply (rule $hn$-refine-cons-pre$[\OF \ FR]$)
apply (rule $hn$-RECT)
apply (rule $hn$-refine-cons-post$[\OF \ FR']$)
apply (rule $S[unfolded \ RCALL-def \ APP-def]$)
apply assumption
apply fact+
done

lemma $hn$-RCALL[sepref-comb-rules]:
assumes $RPREM$ $(\hn$-refine $P' \ c \ Q' \ R$ $(RCALL \ \$a \ \$b))$
and $P \implies \Gamma, \ F \ast \ P'$
shows $hn$-refine $P \ c \ (F \ * \ Q') \ R$ $(RCALL \ \$a \ \$b)$
using $assms$ $hn$-refine-frame$[\where \ m=RCALL$sq$a$b$]
by simp

definition monadic-WHILEIT $I \ b \ f \ s \equiv \{ \$
$\ (\lambda D \ s. \ do \{ \$

122
```plaintext
ASSERT (I s);  
  bv ← b s;  
  if bv then do  
    s ← f s;  
    D s  
  } else do {RETURN s}  
})) s

definition heap-WHILE b f s ≡ do  
  heap.fixp-fun (λ D s. do  
    bv ← b s;  
    if bv then do  
      s ← f s;  
      D s  
    } else do {return s}  
  ) s

lemma heap-WHILE-unfold[code]: heap-WHILE b f s =  
  do  
    bv ← b s;  
    if bv then do  
      s ← f s;  
    heap-WHILE b f s  
  } else  
  return s

unfolding heap-WHILE-def  
apply (subst heap.mono-body-fixp)  
apply pf-mono  
apply simp  
done

lemma WHILEIT-to-monadic: WHILEIT I b f s = monadic-WHILEIT I (λs. RETURN (b s)) f s  
unfolding WHILEIT-def monadic-WHILEIT-def  
unfolding WHILEI-body-def bind-ASSERT-eq-if  
by (simp cong: if-cong)

lemma WHILEIT-pat[def-pat-rules]:  
  WHILEIT$I ≡ UNPROTECT (WHILEIT I)  
  WHILEIT ≡ PR-CONST (WHILEIT (λ-. True))  
by (simp-all add: WHILEIT-def)

lemma id-WHILEIT[id-rules]:  
  PR-CONST (WHILEIT I) :: TYPE(('a ⇒ bool) ⇒ ('a ⇒ 'a nres) ⇒ 'a ⇒ 'a
```

123
lemma WHILE-arities[sepref-monadify-arity]:

\[
PR-CONST \ (WHILEIT \ I) \equiv \lambda_2 b \ f \ s. \ SP \ (PR-CONST \ (WHILEIT \ I)) \$(\lambda_2s. \ b\$s)\$(\lambda_2s. \ f\$s)\$s
\]

by (simp-all add: WHILET-def)

lemma WHILEIT-comb[sepref-monadify-comb]:

\[
PR-CONST \ (WHILEIT \ I) \$(\lambda_2x. \ b \ x)\$f\$s \equiv
\]

Refine-Basic.bind$(EVAL$s)$$(\lambda_2s. \ SP \ (PR-CONST \ (monadic-WHILEIT \ I)))$(\lambda_2s. \ (EVAL(b \ x)))\$f\$s

by (simp-all add: WHILEIT-to-monadic)

lemma hn-monadic-WHILE-aux:

assumes FR: \[P = \Rightarrow \Gamma * \text{hn-ctxt} \ Rs \ s'] (s' \ s)

assumes b-ref: \[s' s. I s' \Rightarrow \text{hn-refine} \]

(\Gamma * \text{hn-ctxt} \ Rs \ s') (b \ s)

(\Gamma b s' s)

(pure bool-rel)

(b' s')

assumes b-fr: \[s' s. \Gamma b s' s \Rightarrow (\Gamma * \text{hn-ctxt} \ Rs \ s')

assumes f-ref: \[s' s. I s' \Rightarrow (\Gamma * \text{hn-ctxt} \ Rs \ s')

(\Gamma f s' s)

(Rs)

(f' s')

assumes f-fr: \[s' s. \Gamma f s' s \Rightarrow (\lambda - -. \text{true}) \ s' s

shows \text{hn-refine} (P) (heap-WHILET b f s) (\Gamma * \text{hn-invalid} \ Rs \ s') (\Gamma b' f' s')

unfolding monadic-WHILEIT-def heap-WHILET-def

apply1 (rule hn-refine-cons-pre[OF FR])

apply weaken-hnr-post

focus (rule hn-refine-cons-pre[OF hnr-RECT])

applyS (subst mult-ac(2)[of \Gamma]; rule entt-refl; fail)

apply1 (rule hnr-ASSERT)

focus (rule hnr-bind)

focus (rule hn-refine-cons-pre[OF hnr-REFE])

applyS (simp add: star-aci)

applyS assumption

solved
focus (rule hnr-If)
applyS (sep-auto; fail)
focus (rule hnr-bind)
  focus (rule hn-refine-cons[OF - f-ref f-fr entt-refl])
    apply (sep-auto simp: hn-ctxt-def pure-def intro!: enttI; fail)
    apply assumption
  solved

focus (rule hn-refine-frame)
applyS rprems
applyS (rule enttI; solve-entails)
solved

apply (sep-auto intro!: enttI; fail)
solved
applyF (sep-auto, rule hn-refine-frame)
applyS (rule hnr-RETURN-pass)
  apply (rule enttI)
  apply (fr-rot-rhs 1)
  apply (fr-rot 1, rule fr-refl)
  apply (rule fr-refl)
  apply solve-entails
  solved

apply (rule entt-refl)
solved

apply (rule enttI)
applyF (rule ent-disjE)
  apply1 (sep-auto simp: hn-ctxt-def pure-def)
  apply1 (rule ent-true-drop)
  apply1 (rule ent-true-drop)
  applyS (rule ent-refl)
  applyS (sep-auto simp: hn-ctxt-def pure-def)
solved
solved
apply pf-mono
solved
done

lemma hn-monadic-WHILE-lin[sepref-comb-rules]:
  assumes INDEP Rs
  assumes FR: P \implies \Gamma \ast hn-ctxt Rs s' s
  assumes b-ref: \\\forall s' s. I s' \implies hn-refine
  (\Gamma \ast hn-ctxt Rs s' s)
  (b s)
  (\Gamma b s' s)
unfolding WHILET-def
by (refine-vec; assumption?)

lemma monadic-WHILET-pat[def-pat-rules]:
monadic-WHILET$I \equiv$ UNPROTECT (monadic-WHILEIT I)
by auto

lemma id-monadic-WHILEIT[id-rules]:
PR-CONST (monadic-WHILEIT I) :: TYPE((‘a ⇒ bool nres) ⇒ (‘a ⇒ ‘a nres)
⇒ ‘a ⇒ ‘a nres)
by simp

lemma monadic-WHILEIT-arities[sepref-monadify-arity]:
PR-CONST (monadic-WHILEIT I) ≡ \(\lambda_b \, f \, s \, . \) SP (PR-CONST (monadic-WHILEIT I))§(\(\lambda_{s_2} \, b \$s\)§(\(\lambda_{s_2} \, f \$s\)§s
by (simp)

lemma monadic-WHILEIT-comb[sepref-monadify-comb]:
PR-CONST (monadic-WHILEIT I)§b§f§s ≡
Refine-Basic.bind$(EVAL§s)§(\(\lambda_{s_2} \, b \$s\)$§(\(\lambda_{s_2} \, f \$s\)$§s
}
by (simp)

definition [simp]: op-ASSERT-bind I m ≡ Refine-Basic.bind (ASSERT I) (\_ \_ m)
lemma pat-op-ASSERT-bind[def-pat-rules]:
Refine-Basic.bind$(ASSERT§I)§(\(\lambda_{m} \, -\) \_ \_ m) \equiv$ UNPROTECT (op-ASSERT-bind I)§m
by simp

term PR-CONST (op-ASSERT-bind I)
lemma id-op-ASSERT-bind[id-rules]:
PR-CONST (op-ASSERT-bind I) :: TYPE(‘a nres ⇒ ‘a nres)
by simp

lemma arity-op-ASSERT-bind[sepref-monadify-arity]:
PR-CONST (op-ASSERT-bind I) ≡ \(\lambda_{m} \, m \) SP (PR-CONST (op-ASSERT-bind I))§m
apply (rule eq-reflection)
by auto

lemma hn-op-ASSERT-bind[sepref-comb-rules]:
assumes I \implies hn-refine Γ c Γ’ R m
shows hn-refine Γ c Γ’ R (PR-CONST (op-ASSERT-bind I)§m)
using assms
apply (cases I)
apply auto
done
**definition** [simp]: \( \text{op-ASSUME-bind } I \ m \equiv \text{Refine-Basic.bind } (\text{ASSUME } I) \ (\lambda \cdot m) \)

**lemma** \( \text{pat-ASSUME-bind}[\text{def-pat-rules}]: \)
\[
\text{Refine-Basic.bind}$(\text{ASSUME}$I$)$(\lambda_2 \cdot m)$ \equiv \text{UNPROTECT} $(\text{op-ASSUME-bind I})$\m
by simp

**lemma** \( \text{id-op-ASSUME-bind}[\text{id-rules}]: \)
\[
\text{PR-CONST} \ (\text{op-ASSUME-bind I}) :: \_ TYPE\ (\_ 
\equiv \_)
\]
by simp

**lemma** \( \text{arity-ASSUME-bind}[\text{sepref-monadify-arity}]: \)
\[
\text{PR-CONST} \ (\text{op-ASSUME-bind I}) \equiv \lambda \_ \_ m. \ \text{SP} \ (\text{PR-CONST} \ (\text{op-ASSUME-bind I})) \m
\]
apply (rule eq-reflection)
by auto

**lemma** \( \text{hn-ASSUME-bind}[\text{sepref-comb-rules}]: \)
assumes \( \text{vassn-tag } \Gamma = \Rightarrow I \)
assumes \( I \Rightarrow \text{hn-refine } \Gamma \ c \Gamma' R m \)
shows \( \text{hn-refine } \Gamma \ c \Gamma' R \ (\text{PR-CONST} \ (\text{op-ASSUME-bind I})) \m \)
apply (rule hn-refine-preI)
using assms
apply (cases I)
apply (auto simp: vassn-tag-def)
done

1.7.1 Import of Parametricity Theorems

**lemma** \( \text{pure-hn-refineI}: \)
assumes \( Q \rightarrow (c,a) \in R \)
shows \( \text{hn-refine } (\uparrow Q) \ (\text{return } c) \ (\uparrow Q) \ (\text{pure } R) \ (\text{RETURN } a) \)
unfolding \( \text{hn-refine-def using assms} \)
by (sep-auto simp: pure-def)

**lemma** \( \text{pure-hn-refineI-no-asm}: \)
assumes \( (c,a) \in R \)
shows \( \text{hn-refine } \text{emp} \ (\text{return } c) \ \text{emp} \ (\text{pure } R) \ (\text{RETURN } a) \)
unfolding \( \text{hn-refine-def using assms} \)
by (sep-auto simp: pure-def)

**lemma** \( \text{import-param-0}: \)
\( (P \rightarrow Q) \equiv \text{Trueprop} \ (\text{PROTECT } P \rightarrow Q) \)
apply (rule, simp+)
done

**lemma** \( \text{import-param-1}: \)
\( (P \rightarrow Q) \equiv \text{Trueprop} \ (P \rightarrow Q) \)
\[(P \rightarrow Q \rightarrow R) \iff (P \land Q \rightarrow R)\]
\[
\text{PROTECT } (P \land Q) \equiv \text{PROTECT } P \land \text{PROTECT } Q
\]
\[
(P \land Q) \land R \equiv P \land Q \land R
\]
\[
(a,c) \in \text{Rel} \land \text{PROTECT } P \iff \text{PROTECT } P \land (a,c) \in \text{Rel}
\]
\[
\text{apply (rule, simp+)}
\]
\[
\text{done}
\]

\textbf{lemma \textit{import-param-2}:}
\[
\text{Trueprop} \ (\text{PROTECT } P \land Q \rightarrow R) \equiv (P \Rightarrow Q \rightarrow R)
\]
\[
\text{apply (rule, simp+)}
\]
\[
\text{done}
\]

\textbf{lemma \textit{import-param-3}:}
\[
\uparrow (P \land Q) = \uparrow P \uparrow \uparrow Q
\]
\[
\uparrow ((a,c) \in \text{R}) = \text{hn-val } R \ a \ c
\]
\[
\text{by (simp-all add: hn-ctxt-def pure-def)}
\]

\textbf{named-theorems-rev \textit{sepref-import-rewrite} (Rewrite rules on importing parametricity theorems)}

\textbf{lemma \textit{to-import-frefD}:}
\[
\text{assumes } (f,g) \in \text{fref } P \text{ R S}
\]
\[
\text{shows } [\text{PROTECT } (P \ y); (x,y) \in \text{R}] \Longrightarrow (f \ x, g \ y) \in S
\]
\[
\text{using assms}
\]
\[
\text{unfolding fref-def}
\]
\[
\text{by auto}
\]

\textbf{lemma \textit{add-PR-CONST}:}
\[
(c,a) \in \text{R} \Longrightarrow (c,PR-CONST \ a) \in \text{R}
\]
\[
\text{by simp}
\]

\textbf{ML} \begin{verbatim}
(\* TODO: Almost clone of Sepref-Rules.to-foparam\*)
fun to-import-fo ctxt thm = let
  val unf-thms = @\{thms split-tupled-all prod-rel-simp uncurry-apply cnv-conj-to-meta Product-Type.split\}
in case Thm.concl-of thm of
  @\{mpat Trueprop ((\_) \in fref - - -)\} =>
    @\{thm to-import-frefD\} OF [thm]
  | forall-intr-vars
  | Local-Defs.unfold0 ctxt unf-thms
  | Variable.gen-all ctxt
  | @\{mpat Trueprop ((\_) \in -)\} =>
    Parametricity.fo-rule thm
  | => raise THM(\text{Expected parametricity or fref theorem},~1,[thm])
end

fun add-PR-CONST thm = case Thm.concl-of thm of
\end{verbatim}
fun import ctxt thm = let
  open Sepref-Basic
  val thm = thm
  |> Conv.fconv-rule Thm.eta-conversion
  |> add-PR-CONST
  |> Local-Defs.unfold0 ctxt @{thms import-param-0}
  |> Local-Defs.unfold0 ctxt @{thms imp-to-meta}
  |> to-import-fo ctxt
  |> Local-Defs.unfold0 ctxt @{thms import-param-1}
  |> Local-Defs.unfold0 ctxt @{thms import-param-2}
  val thm = case Thm.concl-of thm of
    @{mpat Trueprop ((\_. \_)} => thm RS @{thm pure-hn-refineI}
  | ` - => thm
  val thm = Local-Defs.unfold0 ctxt @{thms import-param-3} thm
  |> Conv.fconv-rule (hn-refine-concl-concl-a (K (Id-Op.protect-conv ctxt)) ctxt)
  val thm = Local-Defs.unfold0 ctxt (Named-Theorems-Rev.get ctxt @{named-theorems-rev
    sepref-import-rewrite}) thm
  val thm = Sepref-Rules.add-pure-constraints-rule ctxt thm
  in
    thm
  end

val import-attr = Scan.succeed (Thm.mixed-attribute (fn (context,thm) =>
  let
    val thm = import (Context.proof-of context) thm
    val context = Sepref-Translate.sepref-fr-rules.add-thm thm context
    in (context,thm) end
  ))

val import-attr-rl = Scan.succeed (Thm.rule-attribute [] (fn context =>
  import (Context.proof-of context) => Sepref-Rules.ensure-href (Context.proof-of
    context)
))

val setup = I
1.7.2 Purity

definition import-rel1 \( R \equiv \lambda A c \, c i. \uparrow (\text{is-pure } A \land (ci,c)\in\langle \text{the-pure } A \rangle R) \)
definition import-rel2 \( R \equiv \lambda A B c \, c i. \uparrow (\text{is-pure } A \land \text{is-pure } B \land (ci,c)\in\langle \text{the-pure } A, \text{the-pure } B \rangle R) \)

lemma import-rel1-pure-conv: import-rel1 \( R \) (pure \( A \)) = pure \( \langle A \rangle R \)
  unfolding import-rel1-def
  apply simp
  apply (simp add: pure-def)
done

lemma import-rel2-pure-conv: import-rel2 \( R \) (pure \( A \)) (pure \( B \)) = pure \( \langle A, B \rangle R \)
  unfolding import-rel2-def
  apply simp
  apply (simp add: pure-def)
done

lemma precise-pure[constraint-rules]: single-valued \( R \implies \text{precise } (\text{pure } R) \)
  unfolding precise-def pure-def
  by (auto dest: single-valuedD)

lemma precise-pure-iff-sv: \( \text{precise } (\text{pure } R) \iff \text{single-valued } R \)
  apply (auto simp: precise-pure)
  using preciseD[where \( R=\text{pure } R \) and \( F=\text{emp} \) and \( F'=\text{emp} \)]
  by (sep-auto simp: mod-and-dist intro: single-valuedI)

lemma pure-precise-iff-sv: \[ \text{is-pure } R \] \( \implies \text{precise } R \iff \text{single-valued } (\text{the-pure } R) \)
  by (auto simp: is-pure-conv precise-pure-iff-sv)

lemmas [safe-constraint-rules] = single-valued-Id br-sv
1.8 Sepref-Definition Command

theory Sepref-Definition
imports Sepref-Rules Lib/Pf-Mono-Prover Lib/Term-Synth
keywords sepref-definition :: thy-goal
and sepref-thm :: thy-goal
begin

1.8.1 Setup of Extraction-Tools

declare [[cd-patterns hn-refine - ?f - - -]]

lemma heap-fixp-codegen:
assumes DEF: f ≡ heap.fixp-fun cB
assumes M: (∀x. mono-Heap (λf. cB f x))
sows f x = cB f x
unfolding DEF
apply (rule fun-cong[of - - x])
apply (rule heap.mono-body-fixp)
apply fact
done

ML

structure Sepref-Extraction = struct
val heap-extraction: Refine-Automation.extraction = {
  pattern = Logic.varify-global @{term heap.fixp-fun x},
  gen-thm = @{thm heap-fixp-codegen},
  gen-tac = (fn ctxt =>
    Pf-Mono-Prover.mono-tac ctxt
  )
}

val setup = I
(+#> Refine-Automation.add-extraction trivial triv-extraction*)
#> Refine-Automation.add-extraction heap heap-extraction

end

setup Sepref-Extraction.setup

1.8.2 Synthesis setup for sepref-definition goals

consts UNSPEC::'a

abbreviation hfunspec
:: ('a ⇒ 'b ⇒ assn) ⇒ ('a ⇒ 'b ⇒ assn) × ('a ⇒ 'b ⇒ assn)

where \( R'' \equiv hf-pres R \) UNSPEC

**definition** \( SYNT \) :: ('a ⇒ 'b nres \( R \) heap) \( × \) ('a ⇒ 'b nres \( R \) heap) set ⇒ bool

where \( SYNT \ f \ R \equiv True \)

**definition** [simp]: \( CP-UNCURRY \ - \ - \equiv True \)

**definition** [simp]: \( INTRO-KD \ - \ - \equiv True \)

**definition** [simp]: \( SPEC-RES-ASSN \ - \ - \equiv True \)

**lemma** [synth-rules]: \( CP-UNCURRY \ f \ g \) by simp

**lemma** [synth-rules]: \( CP-UNCURRY \ (uncurry0 \ f) \ (uncurry0 \ g) \) by simp

**lemma** [synth-rules]: \( CP-UNCURRY \ f \ g \Rightarrow CP-UNCURRY \ (uncurry \ f) \ (uncurry \ g) \) by simp

**lemma** [synth-rules]: \( \lfloor INTRO-KD \ R1 \ R1' ; INTRO-KD \ R2 \ R2' \rfloor \Rightarrow INTRO-KD \ (R1^\ast, R2^\ast) \ (R1'^\ast, R2'^\ast) \) by simp

**lemma** [synth-rules]: \( INTRO-KD \ (R') \ (hf-pres \ R \ k) \) by simp

**lemma** [synth-rules]: \( INTRO-KD \ (R^k) \ (R^d) \) by simp

**lemma** [synth-rules]: \( SPEC-RES-ASSN \ R \ R \) by simp

**lemma** [synth-rules]: \( SPEC-RES-ASSN \ UNSPEC \ R \) by simp

**lemma** synth-hnrI:
\[
[CP-UNCURRY \ f \ j ; INTRO-KD \ R \ R'; SPEC-RES-ASSN \ S \ S'] \Rightarrow SYNT-TERM \ (SYNT \ f \ (\lfloor P \rfloor_a \ R \rightarrow S)) \ ((\lfloor j, SDUMMY \rfloor) \in SDUMMY, (\lfloor j, f \rfloor) \in (\lfloor P \rfloor_a \ R' \rightarrow S'))
\]
by (simp add: SYNT-def)

**term** starts-with

**ML**

```
structure Sepref-Definition = struct
  fun make-hnr-goal t ctxt = let
    val ctxt = Variable.declare-term t ctxt
    val (pat,goal) = case Term-Synth.synth-term @\{thms synth-hnrI\} ctxt t of
                      @\{m_pat \ (?pat,?goal)\} => (pat,goal) | t => raise TERM(Synthesized term
does not match,[t])
    val pat = Thm.cterm-of ctxt pat |> Refine-Automation.prepare-cd-pattern ctxt
    val goal = HOLogic.mk_Trueprop goal
    in
    ((pat,goal),ctxt)
  end

  val cfg-prep-code = Attrib.setup-config-bool @\{binding sepref-definition-prep-code\}
                    (K true)
```
local
open Refine-Util
val flags = parse-bool-config prep-code cfg-prep-code
val parse-flags = parse-paren-list flags

in
val sd-parser = parse-flags -- Parse.binding -- Parse.opt-attrs --|
@{keyword is}
    -- Parse.term --| @{keyword ::} -- Parse.term
end

fun mk-synth-term ctxt t-raw r-raw = let
    val t = Syntax.parse-term ctxt t-raw
    val r = Syntax.parse-term ctxt r-raw
    val t = Const (@{const-name SYNTH}, dummyT)$t$r
in
    Syntax.check-term ctxt t
end

fun sd-cmd (((((flags,name),atts),t-raw),r-raw),lthy) = let local
    val ctxt = Refine-Util.apply-configs flags lthy
in
    val flag-prep-code = Config.get ctxt cfg-prep-code
end

val t = mk-synth-term lthy t-raw r-raw
val ((pat,goal),ctxt) = make-hnr-goal t lthy

fun after-qed [[thm]] ctxt = let
    val thm = singleton (Variable.export ctxt lthy) thm
    val (_,lthy) = Local-Theory.note
    ((Refine-Automation.mk-qualified (Binding.name-of name) refine-raw,[]),[thm])
    lthy;
    val ((dthm,rthm),lthy) = Refine-Automation.define-concrete-fun NONE
    name atts [] thm [pat] lthy
    val lthy = lthy
    |> flag-prep-code ? Refine-Automation.extract-recursion-eqs
    [Sepref-Extraction.heap-extraction] (Binding.name-of name) dthm
    val - = Thm.pretty-thm lthy dthm |> Pretty.string-of |> writeln
val = Thm.pretty-thm lthy rthm |> Pretty.string-of |> writeln in
lthy
end |

after-qed thmss = raise THM (After-qed: Wrong thmss structure,~1,flat thmss)
in
Proof theorem NONE after-qed [[ (goal,[]) ]] ctxt end

val = Outer-Syntax.local-theory-to-proof @{command-keyword sepref-definition}
Synthesis of imperative program
(sd-parser >> sd-cmd)

val st-parser = Parse.binding --| @{keyword is} -- Parse.term --| @{keyword ::} -- Parse.term

fun st-cmd ((name,t-raw),r-raw) lthy = let
val t = mk-synth-term lthy t-raw r-raw
val ((-goal),ctxt) = make-hnr-goal t lthy

fun
after-qed [[thm]] ctxt = let
val thm = singleton (Variable.export ctxt lthy) thm

val = Thm.pretty-thm lthy thm |> Pretty.string-of |> tracing

val (-,lthy)
= Local-Theory.note
((Refine-Automation.mk-qualified (Binding.name-of name) refine-raw,[]),[thm])
lthy;
in
lthy
end |

after-qed thmss = raise THM (After-qed: Wrong thmss structure,~1,flat thmss)
in
Proof theorem NONE after-qed [[ (goal,[]) ]] ctxt end

val = Outer-Syntax.local-theory-to-proof @{command-keyword sepref-thm}
Synthesis of imperative program: Only generate raw refinement theorem
(st-parser >> st-cmd)
1.9 Utilities for Interface Specifications and Implementations

theory Sepref-Intf-Util
imports Sepref-Rules Sepref-Translate Lib/Term-Synth Sepref-Combinator-Setup
Lib/Concl-Pres-Clarification
keywords sepref-decl-op :: thy-goal
    and sepref-decl-impl :: thy-goal
begin

1.9.1 Relation Interface Binding

definition INTF-OF-REL :: ('a×'b) set ⇒ 'c itself ⇒ bool
    where [simp]: INTF-OF-REL R I ≡ True

lemma intf-of-relI: INTF-OF-REL (R::(-×'a) set) TYPE('a) by simp
declare intf-of-relI[synth-rules] — Declare as fallback rule

lemma [synth-rules]:
    INTF-OF-REL unit-rel TYPE(unit)
    INTF-OF-REL nat-rel TYPE(nat)
    INTF-OF-REL int-rel TYPE(int)
    INTF-OF-REL bool-rel TYPE(bool)

    INTF-OF-REL R TYPE('a) ⇒ INTF-OF-REL ((R)option-rel) TYPE('a option)
    INTF-OF-REL R TYPE('a) ⇒ INTF-OF-REL ((R)list-rel) TYPE('a list)
    INTF-OF-REL R TYPE('a) ⇒ INTF-OF-REL ((R)nres-rel) TYPE('a nres)
    [INTF-OF-REL R TYPE('a); INTF-OF-REL S TYPE('b)] ⇒ INTF-OF-REL
        (R×,S) TYPE('a×'b)
    [INTF-OF-REL R TYPE('a); INTF-OF-REL S TYPE('b)] ⇒ INTF-OF-REL
        ((R,S)sum-rel) TYPE('a+'b)
    [INTF-OF-REL R TYPE('a); INTF-OF-REL S TYPE('b)] ⇒ INTF-OF-REL
        (R→S) TYPE('a⇒'b)
    by simp-all

lemma synth-intf-of-relI: INTF-OF-REL R I ⇒ SYNTH-TERM R I by simp

1.9.2 Operations with Precondition

definition mop :: ('a⇒bool) ⇒ ('a⇒'b nres) ⇒ 'a ⇒ 'b nres
    — Package operation with precondition
    where [simp]: mop P f ≡ λx. ASSERT (P x) ⇒ f x
lemma param-op-mop-iff:
  assumes \((Q, P) \in R \rightarrow \text{bool-rel}\)
  shows 
  \((f, g) \in [P]_f R \rightarrow \langle S \rangle \nres-rel \iff (\text{mop } Q f, \text{mop } P g) \in R \rightarrow_f \langle S \rangle \nres-rel\)
  using assms
  by (auto
    simp: mop-def fref-def pw-nres-rel-iff refine-pw-simps
    dest: fun-relD)

lemma param-mopI:
  assumes \((f, g) \in [P]_f R \rightarrow \langle S \rangle \nres-rel\)
  assumes \((Q, P) \in R \rightarrow \text{bool-rel}\)
  shows \((\text{mop } Q f, \text{mop } P g) \in R \rightarrow_f \langle S \rangle \nres-rel\)
  using assms by (simp add: param-op-mop-iff)

lemma mop-spec-rl: \(P x \implies \text{mop } P f x \leq f x\) by simp

lemma mop-spec-rl-from-def:
  assumes \(f \equiv \text{mop } P g\)
  assumes \(P x\)
  assumes \(g x \leq z\)
  shows \(f x \leq z\)
  using assms mop-spec-rl by simp

lemma mop-leof-rl-from-def:
  assumes \(f \equiv \text{mop } P g\)
  assumes \(P x \implies g x \leq z\)
  shows \(f x \leq z\)
  using assms
  by (simp add: pw-leof-iff refine-pw-simps)

lemma assert-true-bind-conv: \(\text{ASSERT } \text{True} \Rightarrow m = m\) by simp

lemmas mop-alt-unfolds = curry-def curry0-def mop-def uncurry-apply uncurry0-apply o-apply assert-true-bind-conv

1.9.3 Constraints

lemma add-is-pure-constraint: \([\text{PROP } P; \text{CONSTRAINT } \text{is-pure } A] \Rightarrow \text{PROP } P\).

lemma sepref-relpropI: \(P R = \text{CONSTRAINT } P R\) by simp

Purity

lemmas [constraint-simps] = the-pure-pure
definition \([\text{constraint-abbrevs}]: \text{IS-PURE} \ P \ R \equiv \text{is-pure} \ R \land P \ (\text{the-pure} \ R)\)

lemma \(\text{IS-PURE-pure1} : \)
\[ P \ R \implies \text{IS-PURE} \ P \ (\text{pure} \ R) \]
by (auto simp: \(\text{IS-PURE-def}\))

lemma \([\text{fcomp-norm-simps}]: \) \(\text{CONSTRAINT} \ (\text{IS-PURE} \ \Phi) \ P \implies \text{pure} \ (\text{the-pure} \ P) = P \)
by (simp add: \(\text{IS-PURE-def}\))

lemma \([\text{fcomp-norm-simps}]: \) \(\text{CONSTRAINT} \ (\text{IS-PURE} \ P) \ A \implies P \ (\text{the-pure} \ A) = P \)
by (auto simp: \(\text{IS-PURE-def}\))

lemma \(\text{handle-purity1} : \)
\(\text{CONSTRAINT} \ (\text{IS-PURE} \ \Phi) \ A \implies \text{CONSTRAINT} \ \Phi \ (\text{the-pure} \ A) \)
by (auto simp: \(\text{IS-PURE-def}\))

lemma \(\text{handle-purity2} : \)
\(\text{CONSTRAINT} \ (\text{IS-PURE} \ \Phi) \ A \implies \text{is-pure} \ A \)
by (auto simp: \(\text{IS-PURE-def}\))

1.9.4 Composition

Preconditions

definition \([\text{simp}]: \) \(\text{tcomp-pre} \ Q \ T \ P \equiv \lambda a. \ Q \ a \land (\forall a'. \ (a', \ a) \in T \rightarrow P \ a')\)
definition \(\text{and-pre} P1 \ P2 \equiv \lambda x. \ P1 \ x \land P2 \ x\)
definition \(\text{imp-pre} P1 \ P2 \equiv \lambda x. \ P1 \ x \rightarrow P2 \ x\)

lemma \(\text{and-pre-beta} : PP \rightarrow P \ x \land Q \ x \implies PP \rightarrow \text{and-pre} \ P \ Q \ x \) by (auto simp: \(\text{and-pre-def}\))

lemma \(\text{imp-pre-beta} : PP \rightarrow P \ x \rightarrow Q \ x \implies PP \rightarrow \text{imp-pre} \ P \ Q \ x \) by (auto simp: \(\text{imp-pre-def}\))

definition \(\text{IMP-PRE} \ P1 \ P2 \equiv \forall x. \ P1 \ x \rightarrow P2 \ x\)

lemma \(\text{IMP-PRED} : \) \(\text{IMP-PRE} \ P1 \ P2 \implies P1 \ x \implies P2 \ x \) unfolding \(\text{IMP-PRE-def}\)
by auto

lemma \(\text{IMP-PRE-refl} : \) \(\text{IMP-PRE} \ P \ P \) unfolding \(\text{IMP-PRE-def}\) by auto

definition \(\text{IMP-PRE-CUSTOM} \equiv \text{IMP-PRE}\)

lemma \(\text{IMP-PRE-CUSTOMD} : \) \(\text{IMP-PRE-CUSTOM} \ P1 \ P2 \implies \text{IMP-PRE} \ P1 \ P2\) by (simp add: \(\text{IMP-PRE-CUSTOM-def}\))

lemma \(\text{IMP-PRE-CUSTOMI} : \) \(\forall x. \ P1 \ x \implies P2 \ x \) \(\text{IMP-PRE-CUSTOM} \ P1 \ P2\)
by (simp add: \(\text{IMP-PRE-CUSTOM-def} \text{IMP-PRE-def}\))

lemma \(\text{imp-and-triv-pre} : \) \(\text{IMP-PRE} \ P \ (\text{and-pre} \ (\lambda. \ \text{True}) \ P)\)
unfolding \(\text{IMP-PRE-def and-pre-def}\) by auto

138
Premises

definition ALL-LIST A ≡ (∀x∈set A. x)
definition IMP-LIST A B ≡ ALL-LIST A → B

lemma to-IMP-LISTI:
P ⇒ IMP-LIST [] P
by (auto simp: IMP-LIST-def)

lemma to-IMP-LIST: (P ⇒ IMP-LIST Ps Q) ≡ Trueprop (IMP-LIST (P#Ps) Q)
by (auto simp: IMP-LIST-def ALL-LIST-def intro!: equal-intr-rule)

lemma from-IMP-LIST:
Trueprop (IMP-LIST As B) ≡ (ALL-LIST As ⇒ B)
(ALL-LIST [] ⇒ B) ≡ Trueprop B
(ALL-LIST (A#As) ⇒ B) ≡ (A ⇒ ALL-LIST As ⇒ B)
by (auto simp: IMP-LIST-def ALL-LIST-def intro!: equal-intr-rule)

lemma IMP-LIST-trivial: IMP-LIST A B ⇒ IMP-LIST A B.

Composition Rules

lemma hfcomp-tcomp-pre:
assumes B: (g,h) ∈ [Q]f T → (U)nres-rel
assumes A: (f,g) ∈ [P]a RR' → S
shows (f,h) ∈ [tcomp-pre Q T P]a hrp-comp RR' T → hr-comp S U
using hfcomp[OF A B] by simp

lemma transform-pre-param:
assumes A: IMP-LIST Cns ((f,h) ∈ [tcomp-pre Q T P]a hrp-comp RR' T → hr-comp S U)
assumes P: IMP-LIST Cns ((P,P') ∈ T → bool-rel)
assumes C: IMP-PRE PP' (and-pre P' Q)
shows IMP-LIST Cns ((f,h) ∈ [PP]a hrp-comp RR' T → hr-comp S U)
unfolding from-IMP-LIST
apply (rule hfref-cons)
apply (rule A[unfolded from-IMP-LIST])
apply assumption
apply (erule IMP-PRED[OF C])
using P[unfolded from-IMP-LIST] unfolding and-pre-def
apply (auto dest: fun-relD)]
by simp-all

lemma hfref-mop-conv: ((g,mop P f) ∈ [Q]a R → S) ⇔ (g,f) ∈ [λx. P x ∧ Q x]a R → S
apply (simp add: hfref-to-ASSERT-cons)
apply (fo-rule arg-cong fun-cong)+
by (auto intro!: ext simp: pw-eq-iff refine-pw-simps)

139
lemma hfref-op-to-mop:
assumes \( R: (impl,f) \in [Q]_a R \rightarrow S \)
assumes DEF: \( mf \equiv mop P f \)
assumes C: IMP-PRE PP’ (imp-pre P Q)
shows \( (impl,mf) \in [PP’]_a R \rightarrow S \)
unfolding DEF hfref-mop-conv
apply (rule hfref-cons[OF R])
using C
by (auto simp: IMP-PRE-def imp-pre-def)

lemma hfref-mop-to-op:
assumes \( R: (impl,mf) \in [Q]_a R \rightarrow S \)
assumes DEF: \( mf \equiv mop P f \)
assumes C: IMP-PRE PP’ (and-pre Q P)
shows \( (impl,f) \in [PP’]_a R \rightarrow S \)
using R unfolding DEF hfref-mop-conv
apply (rule hfref-cons)
using C
apply (auto simp: and-pre-def IMP-PRE-def)
done

Precondition Simplification

lemma IMP-PRE-eqI:
assumes \( \forall x. P x \rightarrow Q x \)
assumes CNV P P’
shows IMP-PRE P’ Q
using assms by (auto simp: IMP-PRE-def)

lemma simp-and1:
assumes Q \( \Rightarrow \) CNV P P’
assumes PP \( \rightarrow \) P’ \( \land \) Q
shows PP \( \rightarrow \) P \( \land \) Q
using assms by auto

lemma simp-and2:
assumes P \( \Rightarrow \) CNV Q Q’
assumes PP \( \rightarrow \) P \( \land \) Q’
shows PP \( \rightarrow \) P \( \land \) Q
using assms by auto

lemma triv-and1: Q \( \rightarrow \) True \( \land \) Q by blast

lemma simp-imp:
assumes P \( \Rightarrow \) CNV Q Q’
assumes PP \( \rightarrow \) Q’
shows PP \( \rightarrow \) (P \( \rightarrow \) Q)
using assms by auto

140
lemma \textit{CNV-split}:
assumes \textit{CNV} A A'
assumes \textit{CNV} B B'
shows \textit{CNV} (A \land B) (A' \land B')
using assms by auto

lemma \textit{CNV-prove}:
assumes P
shows \textit{CNV} P True
using assms by auto

lemma \textit{simp-pre-final-simp}:
assumes \textit{CNV} P P'
shows P' \rightarrow P
using assms by auto

lemma \textit{auto-weaken-pre-uncurry-step'}:
assumes \textit{PROTECT} f a \equiv f'
shows \textit{PROTECT} (uncurry f) (a,b) \equiv f' b
using assms
by (auto simp: curry-def dest!: meta-eq-to-obj-eq intro!: eq-reflection)

1.9.5 Protected Constants

lemma \textit{add-PR-CONST-to-def}: x=y \rightarrow \textit{PR-CONST} x \equiv y by simp

1.9.6 Rule Collections

named-theorems-rev sepref-mop-def-thms \langle Sepref: mop – definition theorems \rangle

named-theorems-rev sepref-fref-thms \langle Sepref: fref – theorems \rangle

named-theorems sepref-relprops-transform \langle Sepref: Simp – rules to transform relator properties \rangle

named-theorems sepref-relprops \langle Sepref: Simp – rules to add CONSTRAINT – tags to relator properties \rangle

named-theorems sepref-relprops-simps \langle Sepref: Simp – rules to simplify relator properties \rangle

Default Setup

1.9.7 ML-Level Declarations

ML

(signature SEPREF-INTF-UTIL = sig
(* Miscellaneous*)
val list-filtered-subterms : (term \rightarrow 'a option) \rightarrow term \rightarrow 'a list

(* Interface types for relations *)
val get-intf-of-rel: Proof.context -> term -> typ

(* Constraints *)
(* Convert relations to pure assertions *)
val to-assns-rl: bool -> Proof.context -> thm -> thm

(* Recognize, summarize and simplify CONSTRAINT premises *)
val cleanup-constraints: Proof.context -> thm -> thm

(* Preconditions *)
(* Simplify precondition. Goal must be in IMP-PRE or IMP-PRE-CUSTOM form. *)
val simp-precond-tac: Proof.context -> tactic'

(* Configuration options *)
val cfg-def: bool Config.T (* decl-op: Define constant *)
val cfg-ismop: bool Config.T (* decl-op: Specified term is mop *)
val cfg-map: bool Config.T (* decl-op, decl-impl: Derive mop *)
val cfg-rawgoals: bool Config.T (* decl-op, decl-impl: Do not pre-process/solve goals *)

(* TODO: Make do-cmd usable from ML-level! *)
end

structure Sepref-Intf-Util: SEPREF-INTF-UTIL = struct

val cfg-debug = Attrib.setup-config-bool @{binding sepref-debug-intf-util} (K false)
val dbg-trace = Sepref-Debugging.dbg-trace-msg cfg-debug
val dbg-msg-tac = Sepref-Debugging.dbg-msg-tac cfg-debug

fun list-filtered-subterms f t = let
  fun r t = case f t of
    SOME a => [a]
  | NONE => (case t of
      Abs (\a, t) => r t
    | _ => [])
in r t end

fun get-intf-of-rel ctxt R =
Term-Synth.synth-term @\{ thms synth-intf-of-reII \} ctxt R
|> fastype-of
|> Refine-Util.dest-itselfT

fun add-is-pure-constraint ctxt v thm = let
val v' = Thm.cterm-of ctxt v
val rl = Drule.infer-instantiate' ctxt [NONE, SOME v] @\{ thm add-is-pure-constraint \}
in
thm RS rl
end

fun to-assns-rl add-pure-constr ctxt thm = let
val orig_ctxt = ctxt
val (thm, ctxt) = yield-singleton (apfst snd oo Variable.importT) thm ctxt
val (R, S) = case Thm.concl-of thm of @\{ mpat Trueprop (-∈ fref - ?R ?S) \}
| - => raise THM(to-assns-rl: expected fref - thm, ~1,[thm])

fun mk-cn-subst (fname,(iname,C,A)) = let
val T' = A --> C --> @\{ typ assn \}
val v' = Free(fname,T')
val ct' = @\{ mk-term the-pure ?v' \} | Thm.cterm-of ctxt
in
(v',(iname,ct'))
end

fun relation-flt (name, Type (@\{ type-name set \},[Type (@\{ type-name prod \},[C,A]]))) = SOME (name, C, A)
| relation-flt _ = NONE

val vars = []
|> Term.add-vars R
|> Term.add-vars S
|> map-filter (relation-flt)
val (names, ctxt) = Variable.variant-fixes (map (#1 #> fst) vars) ctxt

val cn-substs = map mk-cn-subst (names ~ vars)

val thm = Drule.infer-instantiate ctxt (map snd cn-substs) thm

val thm = thm |> add-pure-constr ? fold (fn (v,-) => fn thm =>
add-is-pure-constraint ctxt v thm) cn-substs

143
val thm = singleton (Variable.export ctxt orig-ctxt) thm in
  thm
end

fun cleanup-constraints ctxt thm = let
  val orig-ctxt = ctxt
  val (thm, ctxt) = yield-singleton (apfst snd oo Variable.import true) thm ctxt
val xform-thms = Named-Theorems.get ctxt @\{named-theorems sepref-relprops-transform\}
val rprops-thms = Named-Theorems.get ctxt @\{named-theorems sepref-relprops\}
val simp-thms = Named-Theorems.get ctxt @\{named-theorems sepref-relprops-simps\}

fun simp thms = Conv.fconv-rule (Simplifier.asm-full-rewrite (put-simpset HOL-basic-ss ctxt addsimps thms))

(* Check for pure (the-pure R) – patterns *)
local
  val (\_,R,S) = case Thm.concl-of thm of
    | - => raise THM(cleanup-constraints: Expected hfref or fref – theorem,~1,[thm])

  fun flt-pat @\{mpat pure (the-pure ?A)} = SOME A |
    flt-pat - = NONE
val purify-terms = (list-filtered-subterms flt-pat R @ list-filtered-subterms flt-pat S)
  |> distinct op aconv
val thm = fold (add-is-pure-constraint ctxt) purify-terms thm in
  val thm = thm
end

val thm = thm
  |> LocalDefs.unfold0 ctxt xform-thms
  |> LocalDefs.unfold0 ctxt rprops-thms

val insts = map (fn @\{mpat Trueprop (CONSTRAINT - (the-pure -))} => @\{thm handle-purity1\}
    | - => asm-rl
  ) (Thm.prems-of thm)
val thm = (thm OF insts)
  |> Conv.fconv-rule Thm.eta-conversion
  |> simp @{thms handle-parity2}
  |> simp simp-thms

val thm = singleton (Variable.export ctxt orig-ctxt) thm

in
  thm
end

fun simp-precond-tac ctxt = let
  fun simp-only thms = asm-full-simp-tac (put-simpset HOL-basic-ss ctxt
      addsimps thms)
  val rtac = resolve-tac ctxt
val cnv-ss = ctxt delsimps @{thms CNV-def}

(*val uncurry-tac = SELECT-GOAL (ALLGOALS (DETERM o SOLVED'
  (REPEAT' (rtac @{thms auto-weaken-pre-uncurry-step'}))
  THEN' rtac @{thms auto-weaken-pre-uncurry-finish}))*)

val prove-cnv-tac = SOLVED' (rtac @{thms CNV-prove} THEN' SELECT-GOAL
  (auto-tac ctxt))

val do-cnv-tac =
  (cp-clarsimp-tac cnv-ss) THEN-ALL-NEW
  (TRY o REPEAT-ALL-NEW (match-tac ctxt @{thms CNV-split}))
  THEN-ALL-NEW (prove-cnv-tac ORELSE' rtac @{thms CNV-I})

val final-simp-tac =
  rtac @{thms simp-pre-final-simp}
  THEN' eq-clarsimp-tac cnv-ss
  THEN' dbg-msg-tac (Sepref-Debugging.msg-subgoal final-simp-tac: Before CNV-I)
  THEN' rtac @{thms CNV-I}
  THEN' dbg-msg-tac (Sepref-Debugging.msg-text Final−Simp done) ctxt

(*val curry-tac = let open Conv in
  CONVERSION (Refine-Util.HOL-concl-conv (fn ctxt => arg1-conv (top-cone (fn - => try-conv (rewr-cone @{thms uncurry-def}) ctxt)))
  ctxt)
  THEN' REPEAT' (EqSubst.eqsubst-tac ctxt [1] @{thms case-prod-eta})
  THEN' rtac @{thms CNV-I}
end*)
val simp-tupled-pre-tac = 
SELECT-GOAL (Local-Defs.unfold0-tac ctxt @\{thms prod-casesK uncurry0-href-post\})
THEN' REPEAT' (EqSubst.eqsubst-tac ctxt [1] @\{thms case-prod-eta\})
THEN' rtac @\{thms CNV-1\}

val unfold-and-tac = rtac @\{thms and-pre-beta\} THEN-ALL-NEW simp-only
@\{thms split\}

val simp-and1-tac = 
rtac @\{thms simp-and1\} THEN' do-cnv-tac

val simp-and2-tac = 
rtac @\{thms simp-and2\} THEN' do-cnv-tac

val and-plan-tac = 
  simp-and1-tac
  THEN' dbg-msg-tac (Sepref-Debugging.msg-subgoal State after and1) ctxt
  THEN' (
    rtac @\{thms triv-and1\}
    ORELSE'
    dbg-msg-tac (Sepref-Debugging.msg-subgoal Invoking and2 on) ctxt
  )
  THEN' simp-and2-tac
  THEN' dbg-msg-tac (Sepref-Debugging.msg-subgoal State before final-simp-tac)
ctxt
  THEN' final-simp-tac
  )

val unfold-imp-tac = rtac @\{thms imp-pre-beta\} THEN-ALL-NEW simp-only
@\{thms split\}
val simp-imp1-tac = 
rtac @\{thms simp-imp\} THEN' do-cnv-tac

val imp-plan-tac = simp-imp1-tac THEN' final-simp-tac

val imp-pre-tac = APPLY-LIST [
  simp-only @\{thms split-tupled-all\}
  THEN' Refine-Util.instantiate-tuples-subgoal-tac ctxt
  THEN' CASES' [ 
    (unfold-and-tac, ALLGOALS and-plan-tac),
    (unfold-imp-tac, ALLGOALS imp-plan-tac)
  ]
  ,
  simp-tupled-pre-tac
]

val imp-pre-custom-tac = 
SELECT-GOAL (Local-Defs.unfold0-tac ctxt @\{thms and-pre-def\}) THEN'
TRY o SOLVED' (SELECT-GOAL (auto-tac ctxt))
in

CASES' [  
(r tac @\{ thms IMP-PRE-eqI \}, imp-pre-tac 1),  
(r tac @\{ thms IMP-PRE-CUSTOMI \}, ALLGOALS imp-pre-custom-tac)  
]  
end

local  

fun inf-bn-aux name =  
  case String.tokens (fn c => c = #.) name of  
    [] => NONE  
  | [a] => SOME (Binding.name a)  
  | (::a::) => SOME (Binding.name a)

in

fun infer-basename (Const (-type-constraint-\$) t) = infer-basename t  
| infer-basename (Const (name,-)) = inf-bn-aux name  
| infer-basename (Free (name,-)) = inf-bn-aux name  
| infer-basename - = NONE

end

val cfg-mop = Attrib.setup-config-bool @\{ binding sepref-register-mop \} (K true)
val cfg-ismop = Attrib.setup-config-bool @\{ binding sepref-register-ismop \} (K false)
val cfg-rawgoals = Attrib.setup-config-bool @\{ binding sepref-register-rawgoals \} (K false)
val cfg-transfer = Attrib.setup-config-bool @\{ binding sepref-decl-impl-transfer \} (K true)
val cfg-def = Attrib.setup-config-bool @\{ binding sepref-register-def \} (K true)
val cfg-register = Attrib.setup-config-bool @\{ binding sepref-decl-impl-register \} (K true)

local

open Refine-Util
val flags =
| parse-bool-config' mop cfg-mop
| parse-bool-config' ismop cfg-ismop
| parse-bool-config' rawgoals cfg-rawgoals
| parse-bool-config' def cfg-def
val parse-flags = parse-paren-list' flags

val parse-name = Scan.option (Parse.binding -- \@\{ keyword : \})
val parse-relconds = Scan.optional (\@\{ keyword where \} \|-- Parse.and-list1
(Scan.repeat1 Parse.prop) >> flat) []

in
fun do-cmd (((flags, name), opt-raw), relt-raw), relconds-raw)) lthy = let
local
val ctxt = Refine_Util.apply-configs flags lthy
in
val flag-ismop = Config.get ctxt cfg-ismop
val flag-mop = Config.get ctxt cfg-mop andalso not flag-ismop
val flag-rawgoals = Config.get ctxt cfg-rawgoals
val flag-def = Config.get ctxt cfg-def
end

open Sepref-Basic Sepref-Rules

val relt = Syntax.parse-term lthy relt-raw
val relconds = map (Syntax.parse-prop lthy) relconds-raw

val - = dbg-trace lthy Parse relation and relation conditions together
val relt = Const (@{const-name Pure.term}, dummyT) $ relt
local
val l = Syntax.check-props lthy (relt::relconds)
in
val (relt, relconds) = (hd l, tl l)
end
val relt = Logic.dest-term relt

val opt-pre = Syntax.parse-term lthy opt-raw

val - = dbg-trace lthy Infer basename
val name = case name of
SOME name => name
| NONE => (case infer-basename opt-pre of
NONE => (error Could not infer basename: You have to specify a
basename; Binding.empty)
| SOME name => name)
)

fun qname s n = Binding.qualify true (Binding.name-of n) (Binding.name s)

fun def name t-pre attribs lthy = let
val t = Syntax.check-term lthy t-pre
(*|$) Thm.cterm-of lthy
| > Drule.mk-term
val (\texttt{-}, \texttt{lthy}) = \texttt{Local-Theory.open-target lthy}
val ((\texttt{dt},(-, \texttt{thm})), \texttt{lthy}) = \texttt{Local-Theory.define} ((\texttt{name}, \texttt{Mixfix.NoSyn}), ((\texttt{Thm.def-binding name}, @\{\texttt{attributes [code]}\} @\{\texttt{attrs}\}', \texttt{t})))

\texttt{lthy}:

val (\texttt{thm}, \texttt{lthy-old}) = '\texttt{Local-Theory.close-target lthy}'
val \texttt{phi} = \texttt{Proof-Context.export-morphism lthy-old lthy}
val \texttt{thm} = \texttt{Morphism.thm phi thm}
val \texttt{dt} = \texttt{Morphism.term phi dt}

\texttt{in}

\texttt{((\texttt{dt},\texttt{thm}), lthy)}

\texttt{end}

val \texttt{-} = \texttt{dbg-trace lthy Analyze Relation}
val (\texttt{pre}, \texttt{args}, \texttt{res}) = \texttt{analyze-rel relt}
val \texttt{specified-pre} = \texttt{is-some pre}
val \texttt{pre} = \texttt{the-default (mk-triv-precond args) pre}

val \texttt{def-thms} = @\{\texttt{thms PR-CONST-def}\}

val \texttt{-} = \texttt{dbg-trace lthy Define op}
val \texttt{op-name} = \texttt{Binding.prefix-name (if flag-ismop then mop- else op-) name}
val \texttt{(def-thms, opc, lthy)} = 
\texttt{if flag-def then let}
val \texttt{((opc, op-def-thm), lthy)} = \texttt{def-op-name opt-pre @\{\texttt{attributes [simp]}\}}
\texttt{lthy}

val \texttt{opc} = \texttt{Refine-Util.dummify-tvars opc}
val \texttt{def-thms} = \texttt{op-def-thm::def-thms}
\texttt{in}

\texttt{(def-thms, opc, lthy)}

\texttt{end}
\texttt{else let}
val \texttt{-} = \texttt{dbg-trace lthy Refine type of opt-pre to get opc}
val \texttt{opc} = \texttt{Syntax.check-term lthy opt-pre}
val \texttt{new-ctxt} = \texttt{Variable.declare-term opc lthy}
val \texttt{opc} = \texttt{singleton (Variable.export-terms new-ctxt lthy) opc}
\texttt{| > Refine-Util.dummify-tvars}
\texttt{in}

\texttt{(def-thms, opc, lthy)}

\texttt{end}

(* \texttt{PR-CONST Heuristics *})
\texttt{fun pr-const-heuristics basename c-pre lthy = let}
val \texttt{-} = \texttt{dbg-trace lthy (\texttt{PR-CONST heuristics} \texttt{*} @\{\texttt{make-string} c-pre\})}
val c = Syntax.check-term lthy c-pre

in

case c of
  @{mpat PR-CONST -} => ((c-pre,false),lthy)
| Const - => ((c-pre,false),lthy)
| _ => let
  val (f,args) = strip-comb c

val lthy = case f of Const - => let
  val ctxt = Variable.declare-term c lthy
  val lhs = Autoref-Tagging.list-APP (f,args)
  val rhs = @{mk-term UNPROTECT ?c}
  val goal = Logic.mk-equals (lhs,rhs) |> Thm.cterm-of ctxt
  val tac =
    Local-Defs.unfold0-tac ctxt @{thms APP-def UNPROTECT-def}
    THEN ALLGOALS (simp-tac (put-simpset HOL-basic-ss ctxt))
  val thm = Goal.prove-internal ctxt [] goal (K tac)
     |> singleton (Variable.export ctxt lthy)

val (c,lthy) = Local-Theory.note
  ((Binding.suffix-name -def-pat basename, @{attributes [def-pat-rules]}),[thm])

lthy

  val - = Thm.pretty-thm lthy thm |> Pretty.string-of |> writeln
  in
    lthy
  end
| _ => (Pretty.str Complex operation pattern. Added PR-CONST but no pattern rules;
    Pretty.brk 1,Syntax.pretty-term lthy c]
  |> Pretty.string-of |> warning
  ; lthy)

val c-pre = Const(@{const-name PR-CONST},dummyT)$c-pre
in
  ((c-pre,true),lthy)
end
end

val ((opc,-),lthy) = pr-const-heuristics op-name opc lthy

(* Register *)
val arg-intfs = map (get-intf-of-rel lthy) args
val res-intf = get-intf-of-rel lthy res
fun register basename c lthy = let

val _ = dbg-trace lthy Register

open Sepref-Basic

val c = Syntax.check-term lthy c

val ri = case (is-nresT (body-type (fastype-of c)), is-nresT res-intf) of
  (true,false) => mk-nresT res-intf
  | (false,true) => dest-nresT res-intf
  | _ => res-intf

val iT = arg-intfs => ri

val ((-,itype-thm),lthy) = Sepref-Combinator-Setup.sepref-register-single
  (Binding.name-of basename) c iT lthy

val - = Thy-Output.pretty-thm lthy itype-thm |> Pretty.string-of |> writeln

in
  lthy
end

val lthy = register op-name opc lthy

val - = dbg-trace lthy Define pre

val pre-name = Binding.prefix-name pre- name

val ((prec,pre-def-thm),lthy) = def pre-name pre @{attributes [simp]} lthy

val prec = Refine-Util.dummify-tvars prec

val def-thms = pre-def-thm::def-thms

(* Re-integrate pre-constant into type-context of relation. TODO: This is probably not clean/robust *)

val pre = constrain-type-pre (fastype-of prec) prec |> Syntax.check-term lthy

val - = dbg-trace lthy Convert both, relation and operation to uncurried form, and add nres

val - = dbg-trace lthy Convert relation (arguments have already been separated by analyze-rel)

val res = case res of @_nres-rel ==> res | _ => @{mk-term (?res),nres-rel}

val relt = mk-rel (SOME pre,args,res)

val - = dbg-trace lthy Convert operation

val opcT = fastype-of (Syntax.check-term lthy opc)

val op-is-nres = Sepref-Basic.is-nresT (body-type opcT)

val (opcu, op-ar) = let

  val arity = binder-types #> length

  (* Arity of operation is number of arguments before result (which may be a fun-type! !*)

  val res-ar = arity (Relators.rel-absT res |> not op-is-nres ? dest-nresT)
val op-ar = arity opcT - res-ar

val op = op-ar = length args orelse raise TERM(Operation/relation arity mismatch: ^ string-of-int op-ar ^
vs ^ string-of-int (length args),(opc,relt))

(* Add RETURN o...o if necessary *)
val opc =
if op-is-nres then opc
else mk-compN-pre op-ar (Const(@{const-name Refine-Basic.RETURN},dummyT))

opc

(* Add uncurry if necessary *)
val opc = mk-uncurryN-pre op-ar opc
in
(opc, op-ar)
end

(* Build mop-variant *)
val declare-mop = (specified-pre orelse not op-is-nres) andalso flag-mop

val (mop-data,lthy) = if declare-mop then let
val - = dbg-trace lthy mop definition
val mop-rhs = Const(@{const-name mop},dummyT) $ prec $ opcu
|> mk-curryN-pre op-ar
val mop-name = Binding.prefix-name mop-name
val ((mopc,mop-def-thm),lthy) = def mop-name mop-rhs [] lthy
val mopc = Refine-Util.dummify-tvars mopc
val ((mopc,added-pr-const),lthy) = pr-const-heuristics mop-name mopc

lthy

val mop-def-thm' = if added-pr-const then
mop-def-thm RS @{thm add-PR-CONST-to-def}
else mop-def-thm
val (-,lthy) = Local-Theory.note ((Binding.empty, @{attributes [sepref-mop-def-thms]}),[mop-def-thm'])

lthy

val - = dbg-trace lthy mop alternative definition
val alt-unfolds = @{thms mop-alt-unfolds}
|> not specified-pre ? curry op :: pre-def-thm

val mop-alt-thm = Local-Defs.unfold0 lthy alt-unfolds mop-def-thm
|> Refine-Util.shift-lambda-leftN op-ar
val (-,lthy) = Local-Theory.note ((Binding.suffix-name -all mop-name, @{attributes [simp]}),[mop-alt-thm]) lthy
val = dbg-trace lthy mop: register
val lthy = register mop-name mopc lthy

val = dbg-trace lthy mop: veg theorem
local
val Ts = map Relators.rel-absTs args
val ctxt = Variable.declare-thm mop-def-thm lthy
val ctxt = fold Variable.declare-typ Ts ctxt
val (x,ctxt) = Refine-Util.fix-left-tuple-from-Ts x T's ctxt

val mop-def-thm = mop-def-thm
| > Local-Defs.unfold0 ctxt @{thms curry-shl}

fun prep-thm thm = (thm OF [mop-def-thm])
| > Drule.infer-instantiate' ctxt [SOME (Thm.cterm-of ctxt x)]
| > Local-Defs.unfold0 ctxt @{thms uncurry-apply uncurry0-apply o-apply}
| > Local-Defs.unfold0 ctxt (def-thms @{thms Product-Type.split HOL.True-implies-equals})
| > singleton (Variable.export ctxt lthy)

val thms = map prep-thm @{thms mop-spec-rl-from-def mop-leaf-rl-from-def}

in
val (.,lthy) = Local-Theory.note ((qname veg mop-name,@{attributes [refine-veg]}),thms) lthy
end

in
(SOME (mop-name,mopc,mop-def-thm),lthy)
end
else (NONE,lthy)

val = dbg-trace lthy Build Parametricity Theorem
val param-t = mk-pair-in-pre opcu opcu relt
| > Syntax.check-term lthy
| > HOLogic.mk-Trueprop
| > curry Logic.list-implies relconds

val = dbg-trace lthy Build Parametricity Theorem for Precondition
val param-pre-t =
let
val pre-relt = Relators.mk-rel (Relators.list-prodrel-left args) @{term bool-rel}

val param-pre-t = mk-pair-in-pre prec prec pre-relt
| > Syntax.check-term lthy
| > HOLogic.mk_Trueprop
| > curry Logic.list-implies relconds
in
  param-pre-t
end

val - = dbg-trace lthy Build goals
val goals = [[ (param-t, []), (param-pre-t, []) ]]

fun after-qed [[p-thm, pp-thm]] - (*ctxt*) = 
let
  val - = dbg-trace lthy after-qed
  val p-thm' = p-thm | not specified-pre ? Local-Defs.unfold0 lthy
[pre-def-thm]

  val (-,lthy) = Local-Theory ((qname fref op-name,@{attributes [sepref-fref-thms]}), [p-thm']) lthy

val (-,lthy) = Local-Theory ((qname param pre-name,@{attributes [param]}), [pp-thm]) lthy

val p'-unfolds = pre-def-thm :: @{thms True-implies-equals}
val (-,lthy) = Local-Theory ((qname fref' op-name,[]), [Local-Defs.unfold0 lthy p'-unfolds p-thm]) lthy

val lthy = case mop-data of NONE => lthy |
  SOME (mop-name,mopc,mop-def-thm) => let
  (* mop = parametricity theorem: (uncurry^n mopc,uncurry^n mopc) ∈ args → f res *)
  val mopcu = mk-uncurryN-pre op-ar mopc
  val param-mop-t = mk-pair-in-pre mopcu mopcu (mk-rel (NONE,args,res))
| > Syntax.check-term lthy
| > HOLogic.mk-Trueprop
| > curry Logic.list-implies relconds

val ctxt = Variable.auto-fixes param-mop-t lthy

val tac = let
val p-thm = Local-Defs.unfold0 ctxt @{thms PR-CONST-def} p-thm
in
Local-Defs.unfold0-tac ctxt (mop-def-thm :: @{thms PR-CONST-def uncurry-curry-id uncurry-curry0-id})
THEn FIRSTGOAL (dbg-msg-tac (Sepref-Debugging.msg-subgoal Mop−param thm goal after unfolding) ctxt THEN'
  resolve-tac ctxt @{thms param-mopI})
THEN' SOLVED' (resolve-tac ctxt [p-thm] THEN-ALL-NEW

154
assume-tac ctxt

\[ \text{THEN' SOLVED'} (\text{resolve-tac ctxt}\ [\text{pp-thm}] \text{THEN-ALL-NEW}) \]

\]

\)

\}

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)

\)
SOLVED' (plac THEN-ALL-NEW asm-full-simp-tac ctxt)
ORELSE' SOLVED' (cp-clarsimp-tac ctxt THEN-ALL-NEW-FWD
plac THEN-ALL-NEW SELECT- GOAL (auto-tac ctxt))
)
)
)
end

val rf-std = Proof.refine (Method.Basic (fn ctxt => SIMPLE-METHOD
(std-tac ctxt)))

val thm = Drule.infer-instantiate' ctxt [NONE, SOME goal] @{thm
IMP-LIST-trivial}
fun generate-mop-thm ctxt op-thm = let
  val orig-ctxt = ctxt

  val (op-thm, ctxt) = yield-singleton (apfst snd oo Variable.import true) op-thm ctxt

  (val mop-def-thms = Named-Theorems-Rev.get ctxt @ { named-theorems-rev, sepref-mop-def-thms }
   |> map (Local-Defs.unfold0 ctxt @ { thms curry-shl })
   |> resolve-tac ctxt mop-def-thms
   |> fail-mop-def-tac i st
   in
    raise TERM (Found no matching mop-definition, [g])
   end)

  fun fail-hnr-tac _ = raise THM (Invalid hnr−theorem, ~1, [op-thm])

  val tac = APPLY-LIST [ resolve-tac ctxt [op-thm] OREELSE' fail-hnr-tac,
    ((* unfold-PR-CONST-tac ctxt THEN' *) resolve-tac ctxt mop-def-thms)
    OREELSE' fail-mop-def-tac,
    simp-precond-tac ctxt OREELSE' Sepref-Debugging.error-tac' precond
    simplification failed ctxt
  ] 1

  val st = Goal.protect (Thm.nprems-of st) st
  val mop-thm = tac st |> Seq.hd |> Goal.conclude

  val mop-thm = singleton (Variable.export ctxt orig-ctxt) mop-thm
    |> Sepref-Rules.norm-fcomp-rule orig-ctxt
  in mop-thm end

fun generate-op-thm ctxt mop-thm = let
  val orig-ctxt = ctxt

  val (mop-thm, ctxt) = yield-singleton (apfst snd oo Variable.import true) mop-thm ctxt

  (val mop-def-thms = Named-Theorems-Rev.get ctxt @ { named-theorems-rev, sepref-mop-def-thms }
   |> map (Local-Defs.unfold0 ctxt @ { thms curry-shl })
   |> resolve-tac ctxt mop-def-thms
   |> fail-mop-def-tac i st
   in
    raise TERM (Found no matching mop-definition, [g])
   end)

  val tac = APPLY-LIST [ resolve-tac ctxt mop-def-thms
    |> fail-hnr-tac
    | OREELSE' simp-precond-tac ctxt OREELSE' Sepref-Debugging.error-tac' precond
    | simplification failed ctxt
  ] 1

  val st = Goal.protect (Thm.nprems-of st) st
  val mop-thm = tac st |> Seq.hd |> Goal.conclude

  val mop-thm = singleton (Variable.export ctxt orig-ctxt) mop-thm
    |> Sepref-Rules.norm-fcomp-rule orig-ctxt
  in mop-thm end
fun fail-hnr-tac - - = raise THM(Invalid hnr - theorem, ∼ 1, [mop-thm])

fun fail-mop-def-tac i st = let
  val g = nth (Thm.prems_of st) (i - 1)
in
  raise TERM(Found no matching mop - definition, [g])
end

(* Tactic to solve preconditions of hfref-mop-to-op *)
val tac = APPLY-LIST [
  resolve-tac ctxt [mop-thm] ORELSE' fail-hnr-tac,
  ( unfold-PR-CONST-tac ctxt THEN' resolve-tac ctxt mop-def-thms )
  ORELSE' fail-mop-def-tac,
  simp-precond-tac ctxt ORELSE' Sepref-Debugging.error-tac' precond
  simplification failed ctxt
] 1

(* Do synthesis *)
val st = @ { thm hfref-mop-to-op }
val st = Goal.protect (Thm.prems_of st) st
val op-thm = tac st |> Seq.hd |> Goal.conclude

val op-thm = singleton (Variable.export ctxt orig-ctxt) op-thm
  |> Sepref-Rules.norm-fcomp-rule orig-ctxt
in op-thm end

fun chk-result ctxt thm = let
  val (P, R, S) = case Thm.concl_of thm of
    @ { mpat Trueprop (- ∈ hfref ?P ?R ?S) } => (P, R, S)
  | - => raise THM(chk-result: Expected hfref - theorem, ∼ 1, [thm])
fun err t = let
  val ts = Syntax.pretty-term ctxt t |> Pretty.string-of
in
  raise THM(chk-result: Invalid pattern left in assertions: ∼ ts, ∼ 1, [thm])
end

fun check-invalid (t as @ { mpat hr-comp - - }) = err t
| check-invalid (t as @ { mpat hrp-comp - - }) = err t
| check-invalid (t as @ { mpat pure (the-pure - ) }) = err t
| check-invalid (t as @ { mpat - O - }) = err t
| check-invalid - = false

val - = exists-subterm check-invalid R
val - = exists-subterm check-invalid S
in
()
fun to-IMP-LIST ctxt thm =
  (thm RS @{(thm to-IMP-LISTI)}} |> Local-Def.unfold0 ctxt @{thms
to-IMP-LIST})

fun from-IMP-LIST ctxt thm = thm |> Local-Def.unfold0 ctxt @{thms
from-IMP-LIST}

local
  open Refine-Util
  val flags =
    parse-bool-config´ mop cfg-mop
    || parse-bool-config´ ismop cfg-ismop
    || parse-bool-config´ transfer cfg-transfer
    || parse-bool-config´ rawgoals cfg-rawgoals
    || parse-bool-config´ register cfg-register
  val parse-flags = parse-paren-list´ flags
  val parse-precond = Scan.option (@{keyword []} | -- Parse.term -- |
    @ {keyword [ ]})
  val parse-fref-thm = Scan.option (@{keyword uses} -- Parse.thm)

  in
    val di-parser = parse-flags -- Scan.optional (Parse.binding -- | @{keyword
      .}) Binding.empty -- parse-precond -- Parse.thm -- parse-fref-thm
  end

fun di-cmd (((flags,name), precond-raw), i-thm-raw), p-thm-raw) lthy = let
  val i-thm = singleton (Attrib.eval-thms lthy) i-thm-raw
  val p-thm = map-option (singleton (Attrib.eval-thms lthy)) p-thm-raw

  local
    val cctxt = Refine-Util.apply-configs flags lthy
    in
      val flag-mop = Config.get cctxt cfg-mop
      val flag-ismop = Config.get cctxt cfg-ismop
      val flag-rawgoals = Config.get cctxt cfg-rawgoals
      val flag-transfer = Config.get cctxt cfg-transfer
      val flag-register = Config.get cctxt cfg-register
    end

    val fr-attrs = if flag-register then @{attributes [sepref-fr-rules]} else []

    val cctxt = lthy

end
(* Compose with fref-theorem *)
val - = dbg-trace lthy Compose with fref

local
val hf-tcomp-pre = @{thm hfcomp-tcomp-pre} OF [asm-rl,i-thm]
fun compose p-thm = let
  val p-thm = p-thm |> to-assms-rl false lthy
in
  hf-tcomp-pre OF [p-thm]
end

in
val thm = case p-thm of
  SOME p-thm => compose p-thm
| NONE => let
  val p-thms = Named-Theorems-Rev.get ctxt @{
    named-theorems revolver sepref-fref-thms}
  fun err () = let
    val prem-s = nth (Thm.prems-of hf-tcomp-pre) 0 |
      Syntax.pretty-term ctxt |
      Pretty.string_of_in
      error ("Found no fref-theorem matching prem-s")
  end
  in
    case get-first (try compose) p-thms of
      NONE => err ()
    | SOME thm => thm
  end
end

val (thm,ctxt) = yield-singleton (apfst snd oo Variable.import true) thm

val - = dbg-trace lthy Transfer Precond
val thm = to-IMP-LIST ctxt thm
val thm = thm RS @{thm transform-pre-param}

local
val (pre,R,pp-name,pp-type) = case Thm.prems-of thm of
  [@[mpat Trueprop (IMP-LIST - ((?pre,-)∈?R))], @[mpat Trueprop
    (IMP-PRE (mpaq-STRUCT (mpaq-Var ?pp-name ?pp-type)) -))] => (pre,R,pp-name,pp-type)
  | - => raise THM(di-cmd: Cannot recognize first prems of transform-pre-param:
          "", ~1,[thm]])
  in
val thm = if flag-transfer then thm OF [transfer-precond-rl ctxt pre R] else thm

val thm = case precond-raw of
  NONE => thm
| SOME precond-raw => let
  val precond = Syntax.parse-term ctxt precond-raw
  in
    Drule.infer-instantiate ctxt [(pp-name, precond)] thm
  in
    thm
  end
end

val _ = dbg-trace lthy Build goals
val goals = [map (fn x => (x,[])) (Thm.prems-of thm)]

fun after-qed thmss _ = let
  val _ = dbg-trace lthy After QED
  val prems-thms = hd thmss
  val thm = thm OF prems-thms

val thm = from-IMP-LIST ctxt thm

(* Two rounds of cleanup—constraints, norm-fcomp *)
val _ = dbg-trace lthy Cleanup
val thm = thm
  |> cleanup-constraints ctxt
  |> Sepref-Rules.norm-fcomp-rule ctxt
  |> cleanup-constraints ctxt
  |> Sepref-Rules.norm-fcomp-rule ctxt

val thm = thm
  |> singleton (Variable.export ctxt lthy)
  |> zero-var-indexes

val _ = dbg-trace lthy Check Result
val _ = chk-result lthy thm

fun qname suffix = if Binding.is-empty name then name else Binding.suffix-name suffix name
val thm-name = if flag-ismop then qname -hnr-mop else qname -hnr
val (-,lthy) = Local-Theory.note ((thm-name,fr-attribs),[thm]) lthy

val - = Thm.pretty-thm lthy thm |> Pretty.string-of |> writeln

(* Create mop theorem from op-theorem *)
val cr-mop-thm = flag-mop andalso not flag-ismop
val lthy = if cr-mop-thm then
  let
    val - = dbg-trace lthy Create mop-thm
    val mop-thm = thm
    |> generate-mop-thm lthy
    |> zero-var-indexes
  val (-,lthy) = Local-Theory.note ((qname -hnr-mop,fr-attribs),[mop-thm])
  lthy
val - = Thm.pretty-thm lthy mop-thm |> Pretty.string-of |> writeln
in lthy end
else lthy

(* Create op theorem from mop-theorem *)
val cr-op-thm = flag-ismop
val lthy = if cr-op-thm then
  let
    val - = dbg-trace lthy Create op-thm
    val op-thm = thm
    |> generate-op-thm lthy
    |> zero-var-indexes
  val (-,lthy) = Local-Theory.note ((qname -hnr,fr-attribs),[op-thm])
  lthy
val - = Thm.pretty-thm lthy op-thm |> Pretty.string-of |> writeln
in lthy end
else lthy

in
  lthy
end

fun std-tac ctxt = let
  val ptac = REPEAT-ALL-NEW-FWD (Parametricity.net-tac (Parametricity.get-dflt ctxt) ctxt ORELSE'
    assume-tac ctxt)
in
  if flag-rawgoals orelse not flag-transfer then
all-tac
else
  APPLY-LIST [ 
    SELECT-GOAL (Local-Defs.unfold0-tac ctxt @\{thms from-IMP-LIST\}) 
  ] 
THEN' TRY o SOLVED' ptac, 
    simp-precond-tac ctxt 
  ] 1
end

val rf-std = Proof.refine (Method.Basic (fn ctxt => SIMPLE-METHOD (std-tac ctxt)))
  \#> Seq.the-result di-cmd: Standard proof tactic returned empty result sequence

  in 
  Proof theorem NONE after-qed goals ctxt
  \#> rf-std
end

val - = Outer-Syntax.local-theory-to-proof @\{command-keyword sepref-decl-impl\} 
    (di-parser >> di-cmd)
end


1.9.8 Obsolete Manual Specification Helpers

lemma vcg-of-RETURN-np:
  assumes f ≡ RETURN r
  shows SPEC (\lambda x. x=r) ≤ m ⇒ f ≤ m
    and SPEC (\lambda x. x=r) ≤ₙ m ⇒ f ≤ₙ m
  using assms
  by (auto simp: pw-le-iff pw-leof-iff)

lemma vcg-of-RETURN:
  assumes f ≡ do { ASSERT \Phi; RETURN r }
  shows \[\Phi; SPEC (\lambda x. x=r) ≤ m\] ⇒ f ≤ m
    and \[\Phi ⇒ SPEC (\lambda x. x=r) ≤ₙ m\] ⇒ f ≤ₙ m
  using assms
  by (auto simp: pw-le-iff pw-leof-iff refine-pw-simps)

lemma vcg-of-SPEC:
  assumes f ≡ do { ASSERT pre; SPEC post }
  shows \[pre; SPEC post ≤ m\] ⇒ f ≤ m
    and \[pre ⇒ SPEC post ≤ₙ m\] ⇒ f ≤ₙ m
  using assms
by (auto simp: pw-le-iff pw-leof-iff refine-pw-simps)

lemma vcg-of-SPEC-np:
  assumes \( f \equiv \text{SPEC \ post} \)
  shows \( \text{SPEC \ post} \leq m \Rightarrow f \leq m \)
  and \( \text{SPEC \ post} \leq n \Rightarrow f \leq n \ m \)
  using assms
  by auto

lemma mk-mop-rl1:
  assumes \( \forall x. mf x \equiv \text{ASSERT (P x) \Rightarrow RETURN (f x)} \)
  shows \( (\text{RETURN o f, mf}) \in \text{Id} \rightarrow \langle \text{Id} \rangle \text{nres-rel} \)
  unfolding assms[abs-def]
  by (auto intro!: nres-relI simp: pw-le-iff refine-pw-simps)

lemma mk-mop-rl2:
  assumes \( \forall x y. mf x y \equiv \text{ASSERT (P x y) \Rightarrow RETURN (f x y)} \)
  shows \( (\text{RETURN oo f, mf}) \in \text{Id} \rightarrow \text{Id} \rightarrow \langle \text{Id} \rangle \text{nres-rel} \)
  unfolding assms[abs-def]
  by (auto intro!: nres-relI simp: pw-le-iff refine-pw-simps)

lemma mk-mop-rl3:
  assumes \( \forall x y z. mf x y z \equiv \text{ASSERT (P x y z) \Rightarrow RETURN (f x y z)} \)
  shows \( (\text{RETURN ooo f, mf}) \in \text{Id} \rightarrow \text{Id} \rightarrow \text{Id} \rightarrow \langle \text{Id} \rangle \text{nres-rel} \)
  unfolding assms[abs-def]
  by (auto intro!: nres-relI simp: pw-le-iff refine-pw-simps)

lemma mk-mop-rl0-np:
  assumes \( mf \equiv \text{RETURN f} \)
  shows \( (\text{RETURN f, mf}) \in \langle \text{Id} \rangle \text{nres-rel} \)
  unfolding assms[abs-def]
  by (auto intro!: nres-relI simp: pw-le-iff refine-pw-simps)

lemma mk-mop-rl1-np:
  assumes \( \forall x. mf x \equiv \text{RETURN (f x)} \)
  shows \( (\text{RETURN o f, mf}) \in \langle \text{Id} \rangle \text{nres-rel} \)
  unfolding assms[abs-def]
  by (auto intro!: nres-relI simp: pw-le-iff refine-pw-simps)

lemma mk-mop-rl2-np:
  assumes \( \forall x y. mf x y \equiv \text{RETURN (f x y)} \)
  shows \( (\text{RETURN oo f, mf}) \in \text{Id} \rightarrow \text{Id} \rightarrow \langle \text{Id} \rangle \text{nres-rel} \)
  unfolding assms[abs-def]
  by (auto intro!: nres-relI simp: pw-le-iff refine-pw-simps)
lemma mk-op-rl3-np:
  assumes \( \forall x \ y \ z. \, mf \ x \ y \ z \equiv \text{return} \ (f \ x \ y \ z) \)
  shows \( (\text{return} \circ \circ \circ f, \, mf) \in \text{id} \rightarrow \text{id} \rightarrow (\text{id})_{\text{nres-rel}} \)
  unfolding assms [abs-def]
  by (auto intro!: nres-relI simp: pw-le-iff refine-pw-simps)

lemma mk-op-rl0-np:
  assumes \( mf \equiv \text{return} \ f \)
  shows \( \text{uncurry0} \ (mf), \, \text{uncurry0} \ (\text{return} \ f) \) \in \text{unit-rel} \rightarrow_f (\text{id})_{\text{nres-rel}}
  apply (intro frefl nres-relI)
  apply (auto simp: assms)
  done

lemma mk-op-rl1-np:
  assumes \( \forall x. \, mf \ x \equiv \text{return} \ (f \ x) \)
  shows \( \text{uncurry} \ (mf), \, \text{uncurry} \ (\text{return} \ o \ f) \) \in \text{id} \times \text{id} \rightarrow \text{id} \rightarrow (\text{id})_{\text{nres-rel}}
  apply (intro frefl nres-relI)
  apply (auto simp: assms)
  done

lemma mk-op-rl1-np:
  assumes \( \forall x. \, mf \ x \equiv \text{return} \ (f \ x) \)
  shows \( \text{uncurry} \ (mf), \, \text{uncurry} \ (\text{return} \ o \ f) \) \in \text{unit-rel} \rightarrow_f (\text{id})_{\text{nres-rel}}
  apply (intro frefl nres-relI)
  apply (auto simp: assms)
  done

lemma mk-op-rl2-np:
  assumes \( \forall x \ y. \, mf \ x \ y \equiv \text{return} \ (f \ x \ y) \)
  shows \( \text{uncurry} \ (mf), \, \text{uncurry} \ (\text{return} \ oo \ f) \) \in \text{id} \times \text{id} \rightarrow \text{id} \rightarrow (\text{id})_{\text{nres-rel}}
  apply (intro frefl nres-relI)
  apply (auto simp: assms)
  done

lemma mk-op-rl2-np:
  assumes \( \forall x \ y. \, mf \ x \ y \equiv \text{return} \ (f \ x \ y) \)
  shows \( \text{uncurry} \ (mf), \, \text{uncurry} \ (\text{return} \ oo \ f) \) \in \text{id} \times \text{id} \rightarrow \text{id} \rightarrow (\text{id})_{\text{nres-rel}}
  apply (intro frefl nres-relI)
  apply (auto simp: assms)
  done

lemma mk-op-rl3:
  assumes \( \forall x \ y \ z. \, mf \ x \ y \ z \equiv \text{return} \ (f \ x \ y \ z) \)
  shows \( \text{uncurry2} \ (mf), \, \text{uncurry2} \ (\text{return} \ oo \ f) \) \in \text{id} \times \text{id} \rightarrow (\text{id})_{\text{nres-rel}}
  apply (intro frefl nres-relI)

165
apply (auto simp: assms)
done

lemma mk-op-rl3-np:
    assumes \( \forall x y z. \text{mf} x y z \equiv \text{RETURN} (f x y z) \)
    shows \( \text{(uncurry2 mf, uncurry2 (\text{RETURN} \circ o \circ f)}) \in (\text{Id} \times \text{Id}) \times \text{Id} \rightarrow (\text{Id} \times \text{Id}) \)
apply (intro frefl nres-relI)
apply (auto simp: assms)
done

end

1.10 Sepref Tool

theory Sepref-Tool
imports Sepref-Translate Sepref-Definition Sepref-Combinator-Setup Sepref-Intf-Util
begin

In this theory, we set up the sepref tool.

1.10.1 Sepref Method

lemma CONS-init:
    assumes \( \text{hn-refine} \ \Gamma \ c \ \Gamma' \ R \ a \)
    assumes \( \Gamma' \Longrightarrow \Gamma c' \)
    assumes \( \forall a \ c. \ \text{hn-ctxt} \ R \ a \ c \Longrightarrow \text{hn-ctxt} \ Rc \ a \ c \)
    shows \( \text{hn-refine} \ \Gamma \ c \ \Gamma' \ Rc \ a \)
apply (rule hn-refine-cons)
apply (rule entt-refl)
apply (rule assms[unfolded hn-ctxt-def])+
done

lemma ID-init: [ [ID a a' \ \text{TYPE}('T); \ \text{hn-refine} \ \Gamma \ c \ \Gamma' \ R \ a'] ]
\Longrightarrow \text{hn-refine} \ \Gamma \ c \ \Gamma' \ R \ a \ by \ simp

lemma TRANS-init: [ \hn-refine \ \Gamma \ c \ \Gamma' \ R \ a; \ \text{CNV} \ c \ c' ]
\Longrightarrow \text{hn-refine} \ \Gamma \ c' \ \Gamma' \ R \ a
by simp

lemma infer-post-triv: \( P \Longrightarrow P \) by (rule entt-refl)

ML "
structure Sepref = struct
structure sepref-preproc-simps = Named-Thms ( val name = @{binding sepref-preproc} val description = Sepref : Preprocessor simplifications )

structure sepref-opt-simps = Named-Thms ( val name = @{binding sepref-opt-simps} val description = Sepref : Post-Translation optimizations, phase 1 )

structure sepref-opt-simps2 = Named-Thms ( val name = @{binding sepref-opt-simps2} val description = Sepref : Post-Translation optimizations, phase 2 )

fun cons-init-tac ctxt = Sepref-Frame.weaken-post-tac ctxt THEN' resolve-tac ctxt @{thms CONS-init}
fun cons-solve-tac dbg ctxt = let val dbgSOLVED' = if dbg then I else SOLVED' in
dbgSOLVED' ( resolve-tac ctxt @{thms infer-post-triv} ORELSE' Sepref-Translate.side-frame-tac ctxt ) end

fun preproc-tac ctxt = let val ctxt = put-simpset HOL-basic-ss ctxt val ctxt = ctxt addsimps (sepref-preproc-simps.get ctxt) in
Sepref-Rules.prepare-hfref-synth-tac ctxt THEN'
Simplifier.simp-tac ctxt end

fun id-tac ctxt = resolve-tac ctxt @{thms ID-init} THEN' CONVERSION Thm.eta-conversion THEN' DETERM o Id-Op.id-tac Id-Op.Normal ctxt

fun id-init-tac ctxt = resolve-tac ctxt @{thms ID-init} THEN' CONVERSION Thm.eta-conversion THEN' Id-Op.id-tac Id-Op.Init ctxt


fun id-solve-tac ctxt =
Id-Op.id-tac Id-Op.Solve ctzt

(fun id-param-tac ctzt = CONVERSION (Refine-Util.HOL-concl-conv
  (K (Sepref-Param.id-param-conv ctzt)) ctzt)*

fun monadify-tac ctzt = Sepref-Monadify.monadify-tac ctzt

(fun lin-ana-tac ctzt = Sepref-Lin-Ana.lin-ana-tac ctzt*)

fun trans-tac ctzt = Sepref-Translate.trans-tac ctzt

fun opt-tac ctzt = let
  val opt1-ss = put-simpset HOL-basic-ss ctzt
  addsimps sepref-opt-simps
  .get ctxt
  addsimprocs [@{simproc HOL.let-simp}]
  |> Simplifier.add-cong @{thm SP-cong}
  |> Simplifier.add-cong @{thm PR-CONST-cong}
  val unsp-ss = put-simpset HOL-basic-ss ctzt addsimps @{thms SP-def}
  val opt2-ss = put-simpset HOL-basic-ss ctzt
  addsimps sepref-opt-simps2.get ctzt
  addsimprocs [@{simproc HOL.let-simp}]
  in
    simp-tac opt1-ss THEN' simp-tac unsp-ss THEN'
    simp-tac opt2-ss THEN' simp-tac unsp-ss THEN'
    CONVERSION Thm.eta-conversion THEN'
    resolve-tac ctzt @{thms CNV-I}
  end

fun sepref-tac dbg ctzt = (K Sepref-Constraints.ensure-slot-tac)
  THEN'
  Sepref-Basic.PHASES'
  |
    (preproc.preproc-tac,0),
    (cons-init,cons-init-tac,2),
    (id,id-tac,0),
    (monadify,monadify-tac false,0),
    (opt-init_fn ctzt => resolve-tac ctzt @{thms TRANS-init},1),
    (trans,trans-tac,~1),
    (opt, opt-tac,~1),
    (cons-solve1,cons-solve-tac false,~1),
    (cons-solve2,cons-solve-tac false,~1),
    (constraints_fn ctzt => K (Sepref-Constraints.solve-constraint-slot ctzt
    THEN Sepref-Constraints.remove-slot-tac),~1)
  ] (Sepref-Basic.flag-phases-ctrl dbg) ctzt

168
val setup = I
  #> sepref-preproc-simps.setup
  #> sepref-opt-simps.setup
  #> sepref-opt-simps2.setup
end

⟩⟩ setup Sepref

setup method-setup sepref = ⟨ Scan.succeed (fn ctxt =>
  SIMPLE-METHOD (DETERM (SOLVED' (IF-EXGOAL
    (Sepref.sepref-tac false ctxt
    ))) 1))) ⟩
 ⟨Automatic refinement to Imperative/HOL⟩

method-setup sepref-dbg-keep = ⟨ Scan.succeed (fn ctxt => let
    (val ctxt = Config.put Id-Op.cfg-id-debug true ctxt*)
  in
    SIMPLE-METHOD (IF-EXGOAL (Sepref.sepref-tac true ctxt) 1)
  end) ⟩
 ⟨Automatic refinement to Imperative/HOL, debug mode⟩

Default Optimizer Setup

lemma return-bind-eq-let:
do { x←return v; f x } = do { let x=v; f x }
by simp

lemmas [sepref-opt-simps] = return-bind-eq-let bind-return bind-bind id-def

We allow the synthesized function to contain tagged function applications.
This is important to avoid higher-order unification problems when synthesizing
generic algorithms, for example the to-list algorithm for foreach-loops.

lemmas [sepref-opt-simps] = Autoref-Tagging.APP-def

Revert case-pulling done by monadify

lemma case-prod-return-opt[sepref-opt-simps]:
case-prod (λa b. return (f a b)) p = return (case-prod f p)
by (simp split: prod.split)

lemma case-option-return-opt[sepref-opt-simps]:
case-option (return fn) (λs. return (fs s)) v = return (case-option fn fs v)
by (simp split: option.split)

lemma case-list-return[sepref-opt-simps]:
case-list (return fn) (λx xs. return (fc x xs)) l = return (case-list fn fc l)
by (simp split: list.split)

lemma if-return[sepref-opt-simps]:
If b (return t) (return e) = return (If b t e) by simp

In some cases, pushing in the returns is more convenient

169
lemma case-prod-opt2[sepref-opt-simps2]:
(\(\lambda x. \text{return} \ (\text{case} x \text{ of} \ (a,b) \Rightarrow f\ a\ b)\))
= (\(\lambda (a,b). \text{return} \ (f\ a\ b)\))
by auto

1.10.2 Debugging Methods

ML
{
  fun SIMPLE-METHOD-NOPARAM' tac = Scan.succeed (fn ctxt => SIMPLE-METHOD' (IF-EXGOAL (tac ctxt)))
  fun SIMPLE-METHOD-NOPARAM tac = Scan.succeed (fn ctxt => SIMPLE-METHOD (tac ctxt))
}

method-setup sepref-dbg-preproc = (SIMPLE-METHOD-NOPARAM' (fn ctxt => K (Sepref-Constraints.ensure-slot-tac) THEN' Sepref.preproc-tac ctxt))
(Sepref debug: Preprocessing phase)

method-setup sepref-dbg-cons-init = (SIMPLE-METHOD-NOPARAM' Sepref.cons-init-tac)
(Sepref debug: Initialize consequence reasoning)

method-setup sepref-dbg-id = (SIMPLE-METHOD-NOPARAM' (Sepref.id-tac))
(Sepref debug: Identify operations phase)

method-setup sepref-dbg-id-keep = (SIMPLE-METHOD-NOPARAM' (Config.put Id-Op.cfg-id-debug true #> Sepref.id-tac))
(Sepref debug: Identify operations phase. Debug mode, keep intermediate subgoals on failure.)

method-setup sepref-dbg-monadify = (SIMPLE-METHOD-NOPARAM' (Sepref.monadify-tac false))
(Sepref debug: Monadify phase)

method-setup sepref-dbg-monadify-keep = (SIMPLE-METHOD-NOPARAM' (Sepref.monadify-tac true))
(Sepref debug: Monadify phase)

method-setup sepref-dbg-monadify-arity = (SIMPLE-METHOD-NOPARAM' (Sepref-Monadify.arity-tac))
(Sepref debug: Monadify phase: Arity phase)

method-setup sepref-dbg-monadify-comb = (SIMPLE-METHOD-NOPARAM' (Sepref-Monadify.comb-tac))
(Sepref debug: Monadify phase: Comb phase)

method-setup sepref-dbg-monadify-check-EVAL = (SIMPLE-METHOD-NOPARAM' (K (CONCL-COND' (not a Sepref-Monadify.contains-eval))))
(Sepref debug: Monadify phase: check-EVAL phase)

method-setup sepref-dbg-monadify-mark-params = (SIMPLE-METHOD-NOPARAM' (Sepref-Monadify.mark-params-tac))
(Sepref debug: Monadify phase: mark-params phase)

method-setup sepref-dbg-monadify-dup = (SIMPLE-METHOD-NOPARAM' (Sepref-Monadify.dup-tac))
(Sepref debug: Monadify phase: dup phase)

method-setup sepref-dbg-monadify-remove-pass = (SIMPLE-METHOD-NOPARAM' (Sepref-Monadify.remove-pass-tac))
(Sepref debug: Monadify phase: remove-pass phase)
method-setup sepref-dbg-opt-init = (SIMPLE-METHOD-NOPARAM’ (fn ctxt => resolve-tac ctxt @{thms TRANS-init}))
 (Sepref debug: Translation phase initialization)
method-setup sepref-dbg-trans = (SIMPLE-METHOD-NOPARAM’ Sepref-trans-tac)
 (Sepref debug: Translation phase)
method-setup sepref-dbg-opt = (SIMPLE-METHOD-NOPARAM’ (fn ctxt => Sepref-opt-tac ctxt
 THEN’ CONVERSION Thm.eta-conversion
 THEN’ TRY o resolve-tac ctxt @{thms CNV-I})
))
 (Sepref debug: Optimization phase)
method-setup sepref-dbg-cons-solve = (SIMPLE-METHOD-NOPARAM’ (Sepref-cons-solve-tac false))
 (Sepref debug: Solve post-consequences)
method-setup sepref-dbg-cons-solve-keep = (SIMPLE-METHOD-NOPARAM’ (Sepref-cons-solve-tac true))
 (Sepref debug: Solve post-consequences, keep intermediate results)
method-setup sepref-dbg-constraints = (SIMPLE-METHOD-NOPARAM’ (fn ctxt => IF-EXGOAL (K (Sepref-Constraints.solve-constraint-slot ctxt
 THEN Sepref-Constraints.remove-slot-tac
))))
 (Sepref debug: Solve accumulated constraints)

method-setup sepref-dbg-id-init = (SIMPLE-METHOD-NOPARAM’ Sepref-id-init-tac)
 (Sepref debug: Initialize operation identification phase)
method-setup sepref-dbg-id-step = (SIMPLE-METHOD-NOPARAM’ Sepref-id-step-tac)
 (Sepref debug: Single step operation identification phase)
method-setup sepref-dbg-id-solve = (SIMPLE-METHOD-NOPARAM’ Sepref-id-solve-tac)
 (Sepref debug: Complete current operation identification goal)
method-setup sepref-dbg-trans-keep = (SIMPLE-METHOD-NOPARAM’ Sepref-Translate.trans-keep-tac)
 (Sepref debug: Translation phase, stop at failed subgoal)
method-setup sepref-dbg-trans-step = (SIMPLE-METHOD-NOPARAM’ Sepref-Translate.trans-step-tac)
 (Sepref debug: Translation step)
 (Sepref debug: Translation step, keep unsolved subgoals)
method-setup sepref-dbg-side = (SIMPLE-METHOD-NOPARAM’ (fn ctxt =>
 REPEAT-ALL-NEW-FWD (Sepref-Translate.side-cond-dispatch-tac false (K no-tac)
 ctxt))
)
method-setup sepref-dbg-side-unfold = (SIMPLE-METHOD-NOPARAM’ (Sepref-Translate.side-unfold-tac))
method-setup sepref-dbg-side-keep = (SIMPLE-METHOD-NOPARAM’ (fn ctxt =>
 REPEAT-ALL-NEW-FWD (Sepref-Translate.side-cond-dispatch-tac true (K

method-setup sepref-dbg-prepare-frame = ⟨SIMPLE-METHOD-NOPARAM′ Sepref-Frame.prepare-frame-tac⟩
  ⟨Sepref debug: Prepare frame inference⟩

method-setup sepref-dbg-frame = ⟨SIMPLE-METHOD-NOPARAM′ (Sepref-Frame.frame-tac (Sepref-Translate.side-fallback-tac))⟩
  ⟨Sepref debug: Frame inference⟩

method-setup sepref-dbg-merge = ⟨SIMPLE-METHOD-NOPARAM′ (Sepref-Frame.merge-tac (Sepref-Translate.side-fallback-tac))⟩
  ⟨Sepref debug: Frame inference, merge⟩

method-setup sepref-dbg-frame-step = ⟨SIMPLE-METHOD-NOPARAM′ (Sepref-Frame.frame-step-tac (Sepref-Translate.side-fallback-tac) false)⟩
  ⟨Sepref debug: Frame inference, single-step⟩

method-setup sepref-dbg-frame-step-keep = ⟨SIMPLE-METHOD-NOPARAM′ (Sepref-Frame.frame-step-tac (Sepref-Translate.side-fallback-tac) true)⟩
  ⟨Sepref debug: Frame inference, single-step, keep partially solved side conditions⟩

1.10.3 Utilities

Manual hhref-proofs

method-setup sepref-to-hnr = ⟨SIMPLE-METHOD-NOPARAM′ (fn ctxt => Sepref.preproc-tac ctxt THEN' Sepref-Frame.weaken-post-tac ctxt)⟩
  ⟨Sepref: Convert to hnr−goal and weaken postcondition⟩

method-setup sepref-to-hoare = ⟨
  let
    fun sepref-to-hoare-tac ctxt = let
      val ss = put-simpset HOL-basic-ss ctxt
      addsimps @'{thms hn-ctxt-def pure-def}
    in
      Sepref.preproc-tac ctxt
      THEN' Sepref-Frame.weaken-post-tac ctxt
      THEN' resolve-tac ctxt @'{thms hn-refine1}
      THEN' asm-full-simp-tac ss
    end
  in
    SIMPLE-METHOD-NOPARAM′ sepref-to-hoare-tac
  end
⟩ ⟨Sepref: Convert to hoare−triple⟩

Copying of Parameters

lemma fold-COPY: x = COPY x by simp
Copy is treated as normal operator, and one can just declare rules for it!

**lemma hnr-pure-COPY[sepref-fr-rules]:**

\[
\text{CONSTRAINT is-pure } R \implies (\text{return}, \text{RETURN o } COPY) \in R^k \rightarrow a R
\]

by (sep-auto simp: is-pure-conv pure-def intro: hfrefI hn-refineI)

**Short-Circuit Boolean Evaluation**

Convert boolean operators to short-circuiting. When applied before monadify, this will generate a short-circuit execution.

**lemma short-circuit-conv:**

\[
(a \land b) \iff (\text{if } a \text{ then } b \text{ else } \text{False})
\]

\[
(a \lor b) \iff (\text{if } a \text{ then } \text{True} \text{ else } b)
\]

\[
(a \rightarrow b) \iff (\text{if } a \text{ then } b \text{ else } \text{True})
\]

by auto

**Eliminating higher-order**

**lemma ho-prod-move[sepref-preproc]:**

\[
\text{case-prod } (\lambda a b x. f x a b) = (\lambda p x. \text{case-prod } (f x) p)
\]

by (auto intro!: ext)

**declare o-apply[sepref-preproc]**

**Precision Proofs**

We provide a method that tries to extract equalities from an assumption of the form \(- \models P_1 * \ldots * P_n \land_A P_1' * \ldots * P_n',\) if it find a precision rule for \(P_i\) and \(P_i'.\) The precision rules are extracted from the constraint rules.

TODO: Extracting the precision rules from the constraint rules is not a clean solution. It might be better to collect precision rules separately, and feed them into the constraint solver.

**definition prec-spec h \Gamma \Gamma' \equiv h \models \Gamma \land true \land_A \Gamma' \land true**

**lemma prec-specI:**

\[
h \models \Gamma \land_A \Gamma' \implies \text{prec-spec } h \Gamma \Gamma'
\]

unfolding prec-spec-def

by (auto simp: mod-and-dist mod-star-trueI)

**lemma prec-split1-aux:**

\[
A * B * true \Longrightarrow_A A * true
\]

apply (fr-rot 2, fr-rot-rhs 1)

apply (rule ent-star-mono)

by simp-all

**lemma prec-split2-aux:**

\[
A * B * true \Longrightarrow_A B * true
\]

apply (fr-rot 1, fr-rot-rhs 1)

apply (rule ent-star-mono)

by simp-all

173
lemma \textit{prec-spec-splitE}:
assumes \textit{prec-spec} \(h\ (A*B)\ (C*D)\)
obtains \textit{prec-spec} \(h\ A\ C\ \textit{prec-spec} \ h\ B\ D\)
apply (\textit{thin-tac} \([-\ -\] $$\implies$$ -))
apply (rule that)
using \textit{assms}
apply –
unfolding \textit{prec-spec-def}
apply (erule entailsD[\textit{rotated}])
apply (rule ent-conjI)
apply (rule ent-conjE1)
apply (rule prec-split1-aux)
apply (rule ent-conjE2)
apply (rule prec-split1-aux)
apply (erule entailsD[\textit{rotated}])
apply (rule ent-conjI)
apply (rule ent-conjE1)
apply (rule prec-split2-aux)
apply (rule ent-conjE2)
apply (rule prec-split2-aux)
done

lemma \textit{prec-specD}:
assumes \textit{precise} \(R\)
assumes \textit{prec-spec} \(h\ (R\ a\ p)\ (R\ a'\ p)\)
shows \(a\ =\ a'\)
using \textit{assms} unfolding \textit{precise-def} \textit{prec-spec-def CONSTRAINT-def} by blast

ML \(\langle\langle\)

fun prec-extract-eqs-tac ctxt = let
  fun is-precise thm = case Thm.concl_of thm of
    @\{mpat Trueprop \{\textit{precise} -\} \Rightarrow\ true
    | - => false

  val thms = Sepref-Constraints.get-constraint-rules ctxt
  @ Sepref-Constraints.get-safe-constraint-rules ctxt
  val thms = thms
  |> filter is-precise
  val thms = @\{thms snga-prec sngr-prec\} @ thms
  val thms = map (fn thm => thm RS @\{thm \textit{prec-specD}\}) thms

  val thin-prec-spec-rls = @\{thms thin-rl[\textit{Pure.of prec-spec} a b c for a b c]\}

  val tac =
    forward-tac ctxt @\{thms \textit{prec-specI}\}
    THEN' REPEAT-ALL-NEW (ematch-tac ctxt @\{thms \textit{prec-spec-splitE}\})
    THEN' REPEAT o (dresolve-tac ctxt thms)

\(\rangle\rangle\)}
THEN' REPEAT o (eresolve-tac ctxt thin-prec-spec-rls )
in tac end

method-setup prec-extract-eqs = SIMPLE-METHOD-NOPARAM' prec-extract-eqs-tac
⟨Extract equalities from - |= - & - assumption, using precision rules⟩

Combinator Rules

lemma split-merge: [ A \lor A B \Rightarrow t X; X \lor A C \Rightarrow t D ] \Rightarrow ( A \lor A B \lor A C \Rightarrow t D )
proof -
  assume a1: X \lor A C \Rightarrow t D
  assume A \lor A B \Rightarrow t X
  then have A \lor A B \Rightarrow A D \ast true
  using a1 by (meson ent-disjI1-direct ent-frame-fwd enttD entt-def-true)
  then show ?thesis
  using a1 by (metis (no-types) Assertions.ent-disjI2 ent-disjE enttD enttI semigroup.assoc sup.semigroup-axioms)
qed

ML

fun prep-comb-rule thm = let
  fun mrg t = case Logic.strip-assums-concl t of
    @{mpat Trueprop (- \lor A - \lor A - \Rightarrow t -)} => (@{thm split-merge},true)
    | @{mpat Trueprop (hn-refine - - ?G - -)} => ( 
      if not (is-Var (head-of G)) then (@{thm hn-refine-cons-post}, true) 
      else (asm-rl,false) 
    )
    | _ => (asm-rl,false)
  val inst = Thm.prems-of thm |> map mrg
  in
    if exists snd inst then
      prep-comb-rule (thm OF (map fst inst))
    else
      thm |> zero-var-indexes
  end

attribute-setup sepref-prep-comb-rule = (Scan.succeed (Thm.rule-attribut e []) (K prep-comb-rule))
⟨Preprocess combinator rule: Split merge-rules and add missing frame rules⟩

end

175
Chapter 2

Basic Setup

This chapter contains the basic setup of the Sepref tool.

2.1 HOL Setup

theory Sepref-HOL-Bindings
imports Sepref-Tool
begin

2.1.1 Assertion Annotation

Annotate an assertion to a term. The term must then be refined with this assertion.

definition ASSN-ANNOT :: ('a ⇒ 'ai ⇒ assn) ⇒ 'a ⇒ 'a where [simp]: ASSN-ANNOT
A x ≡ x

context fixes A :: 'a ⇒ 'ai ⇒ assn begin

sepref-register PR-CONST (ASSN-ANNOT A)

lemma [def-pat-rules]: ASSN-ANNOT$A ≡ UNPROTECT (ASSN-ANNOT A)
by simp

lemma [sepref-fr-rules]: (return o (λx. x), RETURN o PR-CONST (ASSN-ANNOT A)) ∈ A^d→_aA
  by separ-to-hoare sep-auto
end

lemma annotate-assn: x ≡ ASSN-ANNOT A x by simp

2.1.2 Shortcuts

abbreviation nat-assn ≡ (id-assn::nat ⇒ -)
abbreviation int-assn ≡ (id-assn::int ⇒ -)
abbreviation bool-assn ≡ (id-assn::bool ⇒ -)
2.1.3 Identity Relations

definition IS-ID $R \equiv R=\text{Id}$
definition IS-BELOW-ID $R \equiv R\subseteq\text{Id}$

lemma [safe-constraint-rules]:
   IS-ID $\text{Id}$
   IS-ID $R_1 \Rightarrow IS-ID R_2 \Rightarrow IS-ID (R_1 \rightarrow R_2)$
   IS-ID $R \Rightarrow IS-ID ((R)\text{option-rel})$
   IS-ID $R \Rightarrow IS-ID ((R)\text{list-rel})$
   IS-ID $R_1 \Rightarrow IS-ID R_2 \Rightarrow IS-ID (R_1 \times_r R_2)$
   IS-ID $R_1 \Rightarrow IS-ID R_2 \Rightarrow IS-ID ((R_1,R_2)\text{sum-rel})$
by (auto simp: IS-ID-def)

lemma [safe-constraint-rules]:
   IS-BELOW-ID $\text{Id}$
   IS-BELOW-ID $R \Rightarrow IS-BELOW-ID ((R)\text{option-rel})$
   IS-BELOW-ID $R_1 \Rightarrow IS-BELOW-ID R_2 \Rightarrow IS-BELOW-ID (R_1 \times_r R_2)$
   IS-BELOW-ID $R_1 \Rightarrow IS-BELOW-ID R_2 \Rightarrow IS-BELOW-ID ((R_1,R_2)\text{sum-rel})$
by (auto simp: IS-BELOW-ID-def option-rel-def sum-rel-def list-rel-def)

lemma IS-BELOW-ID-fun-rel-aux: $R_1 \supseteq \text{Id} \Rightarrow IS-BELOW-ID R_2 \Rightarrow IS-BELOW-ID (R_1 \rightarrow R_2)$
by (auto simp: IS-BELOW-ID-def dest: fun-relD)

corollary IS-BELOW-ID-fun-rel[safe-constraint-rules]:
   IS-ID $R_1 \Rightarrow IS-BELOW-ID R_2 \Rightarrow IS-BELOW-ID (R_1 \rightarrow R_2)$
using IS-BELOW-ID-fun-rel-aux[of $\text{Id} R_2$]
by (auto simp: IS-ID-def)

lemma IS-BELOW-ID-list-rel[safe-constraint-rules]:
   IS-BELOW-ID $R \Rightarrow IS-BELOW-ID ((R)\text{list-rel})$
unfolding IS-BELOW-ID-def
proof safe
   fix $l l'$
   assume $A: R\subseteq\text{Id}$
   assume $(l,l')\in(R)\text{list-rel}$
   thus $l=l'$
      apply induction
      using $A$ by auto
qed

lemma IS-ID-imp-BELOW-ID[constraint-rules]:
   IS-ID $R \Rightarrow IS-BELOW-ID R$
by (auto simp: IS-ID-def IS-BELOW-ID-def)

2.1.4 Inverse Relation

lemma inv-fun-rel-eq[simp]: $(A\rightarrow B)^{-1} = A^{-1}\rightarrow B^{-1}$
by (auto dest: fun-relD)

lemma inv-option-rel-eq[simp]: \((K\text{option-rel})^{-1} = (K^{-1}\text{option-rel})\)
  by (auto simp: option-rel-def)

lemma inv-prod-rel-eq[simp]: \((P \times_r Q)^{-1} = P^{-1} \times_r Q^{-1}\)
  by (auto)

lemma inv-sum-rel-eq[simp]: \(((P,Q)\text{sum-rel})^{-1} = (P^{-1},Q^{-1})\text{sum-rel}\)
  by (auto simp: sum-rel-def)

lemma inv-list-rel-eq[simp]: \(((R)\text{list-rel})^{-1} = (R^{-1})\text{list-rel}\)
  unfolding list-rel-def
  apply safe
  apply (subst list.rel-flip[symmetric])
  apply (simp add: conversep-iff [abs-def])
  apply (subst list.rel-flip[symmetric])
  apply (simp add: conversep-iff [abs-def])
  done

lemmas [constraint-simps] =
  Relation.converse-Id
  inv-fun-rel-eq
  inv-option-rel-eq
  inv-prod-rel-eq
  inv-sum-rel-eq
  inv-list-rel-eq

2.1.5 Single Valued and Total Relations

definition IS-LEFT-UNIQUE R \equiv single-valued \((R^{-1})\)
definition IS-LEFT-TOTAL R \equiv Domain R = UNIV
definition IS-RIGHT-TOTAL R \equiv Range R = UNIV
abbreviation (input) IS-RIGHT-UNIQUE \equiv single-valued

lemmas IS-RIGHT-UNIQUED = single-valuedD
lemma IS-LEFT-UNIQUED: \[(IS-LEFT-UNIQUE r; (y, x) \in r; (z, x) \in r) \implies y = z\]
  by (auto simp: IS-LEFT-UNIQUE-def dest: single-valuedD)

lemma prop2p:
  IS-LEFT-UNIQUE R = left-unique (rel2p R)
  IS-RIGHT-UNIQUE R = right-unique (rel2p R)
  right-unique (rel2p (R^{-1})) = left-unique (rel2p R)
  IS-LEFT-TOTAL R = left-total (rel2p R)
  IS-RIGHT-TOTAL R = right-total (rel2p R)
  by (auto
    simp: IS-LEFT-UNIQUE-def left-unique-def single-valued-def
    simp: right-unique-def

178
simp: \text{IS-LEFT-TOTAL-def left-total-def}
simp: \text{IS-RIGHT-TOTAL-def right-total-def}
simp: \text{rel2p-def}

\textbf{lemma } \text{p2prop}:
left-unique \( P \) = \text{IS-LEFT-UNIQUE} (p2rel \( P \))
right-unique \( P \) = \text{IS-RIGHT-UNIQUE} (p2rel \( P \))
left-total \( P \) = \text{IS-LEFT-TOTAL} (p2rel \( P \))
right-total \( P \) = \text{IS-RIGHT-TOTAL} (p2rel \( P \))
bi-unique \( P \) \iff left-unique \( P \) \land \right-unique \( P \)

\textbf{by} \ (\text{auto}
\simp: \text{IS-LEFT-UNIQUE-def left-unique-def single-valued-def}
\simp: \text{right-unique-def bi-unique-alt-def}
\simp: \text{IS-LEFT-TOTAL-def left-total-def}
\simp: \text{IS-RIGHT-TOTAL-def right-total-def}
\simp: \text{p2rel-def}
)

\textbf{lemmas} \ [\text{safe-constraint-rules}]=
\text{single-valued-Id}
\text{prod-rel-sv}
\text{list-rel-sv}
\text{option-rel-sv}
\text{sum-rel-sv}

\textbf{lemma} \ [\text{safe-constraint-rules}]:
\text{IS-LEFT-UNIQUE Id}
\text{IS-LEFT-UNIQUE R1 } \Longrightarrow \text{IS-LEFT-UNIQUE R2 } \Longrightarrow \text{IS-LEFT-UNIQUE} (R1 \times, R2)
\text{IS-LEFT-UNIQUE R1 } \Longrightarrow \text{IS-LEFT-UNIQUE R2 } \Longrightarrow \text{IS-LEFT-UNIQUE} ((R1, R2) \text{sum-rel})
\text{IS-LEFT-UNIQUE R } \Longrightarrow \text{IS-LEFT-UNIQUE} ((R) \text{option-rel})
\text{IS-LEFT-UNIQUE R } \Longrightarrow \text{IS-LEFT-UNIQUE} ((R) \text{list-rel})

\textbf{by} \ (\text{auto simp: IS-LEFT-UNIQUE-def prod-rel-sv sum-rel-sv option-rel-sv list-rel-sv})

\textbf{lemma} \ IS-LEFT-TOTAL-alt: IS-LEFT-TOTAL \( R \) \iff (\forall x. \exists y. (x,y) \in R)

\textbf{by} \ (\text{auto simp: IS-LEFT-TOTAL-def})

\textbf{lemma} \ IS-RIGHT-TOTAL-alt: IS-RIGHT-TOTAL \( R \) \iff (\forall x. \exists y. (y,x) \in R)

\textbf{by} \ (\text{auto simp: IS-RIGHT-TOTAL-def})

\textbf{lemma} \ [\text{safe-constraint-rules}]:
\text{IS-LEFT-TOTAL Id}
\text{IS-LEFT-TOTAL R1 } \Longrightarrow \text{IS-LEFT-TOTAL R2 } \Longrightarrow \text{IS-LEFT-TOTAL} (R1 \times, R2)
\text{IS-LEFT-TOTAL R1 } \Longrightarrow \text{IS-LEFT-TOTAL R2 } \Longrightarrow \text{IS-LEFT-TOTAL} ((R1, R2) \text{sum-rel})
\text{IS-LEFT-TOTAL R } \Longrightarrow \text{IS-LEFT-TOTAL} ((R) \text{option-rel})
\textbf{apply} \ (\text{auto simp: IS-LEFT-TOTAL-alt sum-rel-def option-rel-def list-rel-def})
\textbf{apply} \ (\text{rename-tac x}; \text{case-tac x}; \text{auto})
\textbf{apply} \ (\text{rename-tac x}; \text{case-tac x}; \text{auto})
\textbf{done}
lemma [safe-constraint-rules]: \( IS\-LEFT\-TOTAL \, R \implies IS\-LEFT\-TOTAL \,(\langle R \rangle\text{list-rel}) \)
unfolding \( IS\-LEFT\-TOTAL\)-alt
proof safe
  assume \( A \): \( \forall \, x.\, \exists \, y. \, (x,y)\in R \)
  fix \( l \)
  show \( \exists \, l'. \, (l',l)\in (R)\text{list-rel} \)
    apply (induction \( l \))
    using \( A \)
    by (auto simp: list-rel-split-right-iff)
qed

lemma [safe-constraint-rules]:
\( IS\-RIGHT\-TOTAL \, \text{Id} \)
\( IS\-RIGHT\-TOTAL \, R1 \implies IS\-RIGHT\-TOTAL \, R2 \implies IS\-RIGHT\-TOTAL \,(R1 \times R2) \)
\( IS\-RIGHT\-TOTAL \, R1 \implies IS\-RIGHT\-TOTAL \, R2 \implies IS\-RIGHT\-TOTAL \,(\langle R1,R2 \rangle\text{sum-rel}) \)
\( IS\-RIGHT\-TOTAL \, R \implies IS\-RIGHT\-TOTAL \,(\langle R \rangle\text{option-rel}) \)
apply (auto simp: IS\-RIGHT\-TOTAL-alt IS\-LEFT\-TOTAL-alt IS\-LEFT\-UNIQUE-def)

lemma [safe-constraint-rules]:
\( IS\-RIGHT\-TOTAL \, R \implies IS\-RIGHT\-TOTAL \,(\langle R \rangle\text{list-rel}) \)
unfolding IS\-RIGHT\-TOTAL-alt
proof safe
  assume \( A \): \( \forall \, x.\, \exists \, y. \, (y,x)\in R \)
  fix \( l \)
  show \( \exists \, l'. \, (l',l)\in (R)\text{list-rel} \)
    apply (induction \( l \))
    using \( A \)
    by (auto simp: list-rel-split-left-iff)
qed

lemma [constraint-simps]:
\( IS\-LEFT\-TOTAL \,(R^{-1}) \iff IS\-RIGHT\-TOTAL \, R \)
\( IS\-RIGHT\-TOTAL \,(R^{-1}) \iff IS\-LEFT\-TOTAL \, R \)
\( IS\-LEFT\-UNIQUE \,(R^{-1}) \iff IS\-RIGHT\-UNIQUE \, R \)
\( IS\-RIGHT\-UNIQUE \,(R^{-1}) \iff IS\-LEFT\-UNIQUE \, R \)
by (auto simp: IS\-RIGHT\-TOTAL-alt IS\-LEFT\-TOTAL-alt IS\-LEFT\-UNIQUE-def)

lemma [safe-constraint-rules]:
\( IS\-RIGHT\-UNIQUE \, A \implies IS\-RIGHT\-TOTAL \, B \implies IS\-RIGHT\-TOTAL \,(A\rightarrow B) \)
\( IS\-RIGHT\-TOTAL \, A \implies IS\-RIGHT\-UNIQUE \, B \implies IS\-RIGHT\-UNIQUE \,(A\rightarrow B) \)
\( IS\-LEFT\-UNIQUE \, A \implies IS\-LEFT\-TOTAL \, B \implies IS\-LEFT\-TOTAL \,(A\rightarrow B) \)
\( IS\-LEFT\-TOTAL \, A \implies IS\-LEFT\-UNIQUE \, B \implies IS\-LEFT\-UNIQUE \,(A\rightarrow B) \)
apply (simp-all add: prop2p rel2p)
apply (blast intro: transfer-raw)+
done

lemma [constraint-rules]:
  IS-BELLOW-ID R \implies IS-RIGHT-UNIQUE R
  IS-BELLOW-ID R \implies IS-LEFT-UNIQUE R
  IS-ID R \implies IS-RIGHT-TOTAL R
  IS-ID R \implies IS-LEFT-TOTAL R
by (auto simp: IS-BELOW-ID-def IS-ID-def IS-LEFT-UNIQUE-def IS-RIGHT-TOTAL-def
     IS-LEFT-TOTAL-def
     intro: single-valuedI)

thm constraint-rules

**Additional Parametricity Lemmas**

lemma param-distinct[param]: [IS-LEFT-UNIQUE A; IS-RIGHT-UNIQUE A] \implies (distinct, distinct) \in \langle A \rangle list-rel \to bool-rel
apply (fold rel2p-def)
apply (simp add: rel2p)
apply (rule distinct-transfer)
apply (simp add: p2prop)
done

lemma param-Image[param]:
  assumes IS-LEFT-UNIQUE A IS-RIGHT-UNIQUE A
  shows ((\", \")) \in \langle A \times B \rangle set-rel \to \langle A \rangle set-rel \to \langle B \rangle set-rel
apply (clarsimp simp: set-rel-def; intro conjI)
apply (fastforce dest: IS-RIGHT-UNIQUED[OF assms(2)]
apply (fastforce dest: IS-LEFT-UNIQUED[OF assms(1)]
done

lemma pres-eq-iff-svb: ((=), (=)) \in K \to K \to bool-rel \iff (single-valued K \land single-valued (K^{-1}))
apply (safe intro!: single-valuedI)
apply (metis (full-types) IdD fun-relD1)
apply (metis (full-types) IdD fun-relD1)
by (auto dest: single-valuedD)

definition IS-PRES-EQ R \equiv ((=), (=)) \in R \to R \to bool-rel
lemma [constraint-rules]: [single-valued R; single-valued (R^{-1})] \implies IS-PRES-EQ R
by (simp add: pres-eq-iff-svb IS-PRES-EQ-def)

**2.1.6 Bounded Assertions**

definition b-rel R P \equiv R \cap UNIV \times Collect P
definition b-assn A P \equiv \lambda x y. A x y * \uparrow (P x)
lemma b-assn-pure-conv [constraint-simps]: b-assn (pure R) P = pure (b-rel R P)
  by (auto intro: ext simp: b-assn-def pure-def)

lemmas [sepref-import-rewrite, sepref-frame-normrel-eqs, fcomp-norm-unfold]
  = b-assn-pure-conv [symmetric]

lemma b-rel-nesting [simp]:
  b-rel (b-rel R P1) P2 = b-rel (λx. P1 x ∧ P2 x)
  by (auto simp: b-rel-def)

lemma b-rel-triv [simp]:
  b-assn A (λx. True) = A
  by (auto simp: b-assn-def)

lemma b-assn-nesting [simp]:
  b-assn (b-assn A P1) P2 = b-assn A (λx. P1 x ∧ P2 x)
  by (auto simp: b-assn-def intro!: ext)

lemma b-assn-triv [simp]:
  b-assn A (λx. True) = A
  by (auto simp: b-assn-def intro!: ext)

lemmas [simp, constraint-simps, sepref-import-rewrite, sepref-frame-normrel-eqs, fcomp-norm-unfold]
  = b-rel-nesting b-assn-nesting

lemma b-rel-simp [simp]:
  (x, y) ∈ b-rel R P ←→ (x, y) ∈ R ∧ P y
  by (auto simp: b-rel-def)

lemma b-assn-simp [simp]:
  b-assn A P x y = A x y ∧ (P x)
  by (auto simp: b-assn-def)

lemma b-rel-Range [simp]:
  Range (b-rel R P) = Range R ∩ Collect P
  by auto

lemma b-assn-rdom [simp]:
  rdom (b-assn R P) x ←→ rdom R x ∧ P x
  by (auto simp: rdom-def)

lemma b-rel-below-id [constraint-rules]:
  IS-BELOW-ID R ⇒ IS-BELOW-ID (b-rel R P)
  by (auto simp: IS-BELOW-ID-def)

lemma b-rel-left-unique [constraint-rules]:
  IS-LEFT-UNIQUE R ⇒ IS-LEFT-UNIQUE (b-rel R P)
  by (auto simp: IS-LEFT-UNIQUE-def single-valued-def)

lemma b-rel-right-unique [constraint-rules]:
  IS-RIGHT-UNIQUE R ⇒ IS-RIGHT-UNIQUE (b-rel R P)
  by (auto simp: single-valued-def)

— Registered as safe rule, although may loose information in the odd case that purity depends condition.

lemma b-assn-is-pure [safe-constraint-rules]:
  is-pure A ⇒ is-pure (b-assn A P)
  by (auto simp: is-pure-conv b-assn-pure-conv)

182
— Most general form

**lemma** b-assn-subtyping-match[sepref-frame-match-rules]:

assumes \( \text{hn-ctxt} (b\text{-assn} A P) \ x \ y \implies \text{hn-ctxt} A' x y \)

assumes \( [\text{vassn-tag} (\text{hn-ctxt} A x y); \ vassn-tag (\text{hn-ctxt} A' x y); P x] \implies P' x \)

shows \( \text{hn-ctxt} (b\text{-assn} A P) \ x \ y \implies \text{hn-ctxt} (b\text{-assn} A' P') \ x \ y \)

using assms

unfolding \( \text{hn-ctxt-def} \ b\text{-assn-def} \) entails-def

by (fastforce simp: vassn-tag-def mod-star-conv)

— Simplified forms:

**lemma** b-assn-subtyping-match-eqA[sepref-frame-match-rules]:

assumes \( [\text{vassn-tag} (\text{hn-ctxt} A x y); P x] \implies P' x \)

shows \( \text{hn-ctxt} (b\text{-assn} A P) \ x \ y \implies \text{hn-ctxt} (b\text{-assn} A P') \ x \ y \)

apply (rule b-assn-subtyping-match)

subgoal

unfolding \( \text{hn-ctxt-def} \ b\text{-assn-def} \) entails-def

by (fastforce simp: vassn-tag-def mod-star-conv)

subgoal

using assms .

done

**lemma** b-assn-subtyping-match-tR[sepref-frame-match-rules]:

assumes \( [P x] \implies \text{hn-ctxt} A x y \implies \text{hn-ctxt} A' x y \)

shows \( \text{hn-ctxt} (b\text{-assn} A P) \ x \ y \implies \text{hn-ctxt} (b\text{-assn} A P') \ x \ y \)

using assms

unfolding \( \text{hn-ctxt-def} \ b\text{-assn-def} \) entails-def

by (fastforce simp: vassn-tag-def mod-star-conv)

**lemma** b-assn-subtyping-match-tL[sepref-frame-match-rules]:

assumes \( \text{hn-ctxt} A x y \implies \text{hn-ctxt} A' x y \)

assumes \( [\text{vassn-tag} (\text{hn-ctxt} A x y)] \implies P' x \)

shows \( \text{hn-ctxt} A x y \implies \text{hn-ctxt} (b\text{-assn} A' P') \ x \ y \)

using assms

unfolding \( \text{hn-ctxt-def} \ b\text{-assn-def} \) entails-def

by (fastforce simp: vassn-tag-def mod-star-conv)

**lemma** b-assn-subtyping-match-eqA-tR[sepref-frame-match-rules]:

\( \text{hn-ctxt} (b\text{-assn} A P) \ x \ y \implies \text{hn-ctxt} A x y \)

unfolding \( \text{hn-ctxt-def} \ b\text{-assn-def} \)

by (sep-auto intro: enttI)

**lemma** b-assn-subtyping-match-eqA-tL[sepref-frame-match-rules]:

assumes \( [\text{vassn-tag} (\text{hn-ctxt} A x y)] \implies P' x \)

shows \( \text{hn-ctxt} A x y \implies \text{hn-ctxt} (b\text{-assn} A P') \ x \ y \)

using assms

unfolding \( \text{hn-ctxt-def} \ b\text{-assn-def} \) entails-def

by (fastforce simp: vassn-tag-def mod-star-conv)
— General form

**Lemma b-rel-subtyping-merge[sepref-frame-merge-rules]:**

**Assumes** \( \text{hn-ctxt} \; A \; x \; y \; \lor \; A \; \text{hn-ctxt} \; A' \; x \; y \implies_t \; \text{hn-ctxt} \; A \; x \; y \)

**Shows** \( \text{hn-ctxt} \; (b-assn \; A \; P) \; x \; y \; \lor \; A \; \text{hn-ctxt} \; (b-assn \; A' \; P') \; x \; y \implies_t \; \text{hn-ctxt} \; (b-assn \; A \; (\lambda x. \; P \; x \; \lor \; P' \; x)) \; x \; y \)

**Using** assms

**Unfolding** \( \text{hn-ctxt-def} \; b-assn-def \; entailst-def \; entails-def \)

**By** (fastforce simp: vassn-tag-def)

— Simplified forms

**Lemma b-rel-subtyping-merge-eqA[sepref-frame-merge-rules]:**

**Shows** \( \text{hn-ctxt} \; (b-assn \; A \; P) \; x \; y \; \lor \; A \; \text{hn-ctxt} \; (b-assn \; A' \; P') \; x \; y \implies_t \; \text{hn-ctxt} \; (b-assn \; A \; (\lambda x. \; P \; x \; \lor \; P' \; x)) \; x \; y \)

**Apply** (rule b-rel-subtyping-merge)

**By** simp

**Lemma b-rel-subtyping-merge-tL[sepref-frame-merge-rules]:**

**Assumes** \( \text{hn-ctxt} \; A \; x \; y \; \lor \; A \; \text{hn-ctxt} \; A' \; x \; y \implies_t \; \text{hn-ctxt} \; A \; x \; y \)

**Shows** \( \text{hn-ctxt} \; (b-assn \; A \; P) \; x \; y \; \lor \; A \; \text{hn-ctxt} \; (b-assn \; A' \; P') \; x \; y \implies_t \; \text{hn-ctxt} \; A \; x \; y \)

**Using** b-rel-subtyping-merge[of \( A \; x \; y \; A' \; Am \; \lambda \)-. True \( P' \), simplified] assms .

**Lemma b-rel-subtyping-merge-tR[sepref-frame-merge-rules]:**

**Assumes** \( \text{hn- ctxt} \; A \; x \; y \; \lor \; A \; \text{hn-ctxt} \; A' \; x \; y \implies_t \; \text{hn-ctxt} \; A \; x \; y \)

**Shows** \( \text{hn-ctxt} \; (b-assn \; A \; P) \; x \; y \; \lor \; A \; \text{hn-ctxt} \; (b-assn \; A' \; P') \; x \; y \implies_t \; \text{hn-ctxt} \; A \; x \; y \)

**Using** b-rel-subtyping-merge[of \( A \; x \; y \; A' \; Am \; \lambda \)-. True, simplified] assms .

**Lemma b-rel-subtyping-merge-eqA-tL[sepref-frame-merge-rules]:**

**Shows** \( \text{hn-ctxt} \; (b-assn \; A \; P) \; x \; y \; \lor \; A \; \text{hn- ctxt} \; (b-assn \; A \; P') \; x \; y \implies_t \; \text{hn-ctxt} \; (b-assn \; A \; (\lambda x. \; P \; x \; \lor \; P' \; x)) \; x \; y \)

**Using** b-rel-subtyping-merge[of \( A \; x \; y \; A' \; Am \; \lambda \)-. True, simplified] assms .

**Lemma b-rel-subtyping-merge-eqA-tR[sepref-frame-merge-rules]:**

**Shows** \( \text{hn-ctxt} \; (b-assn \; A \; P) \; x \; y \; \lor \; A \; \text{hn-ctxt} \; (b-assn \; A \; P') \; x \; y \implies_t \; \text{hn-ctxt} \; A \; x \; y \)

**Using** b-rel-subtyping-merge[of \( A \; x \; y \; A' \; Am \; \lambda \)-. True, simplified] assms .

**Lemma b-assn-invalid-merge1: hn-invalid \( (b-assn \; A \; P) \; x \; y \; \lor \; A \; \text{hn-invalid} \; (b-assn \; A \; P') \; x \; y \)

**Implies** \( \text{hn-invalid} \; (b-assn \; A \; (\lambda x. \; P \; x \; \lor \; P' \; x)) \; x \; y \)

**By** (sep-auto simp: hn-ctxt-def invalid-assn-def entailst-def)

**Lemma b-assn-invalid-merge2: hn-invalid \( (b-assn \; A \; P) \; x \; y \; \lor \; A \; \text{hn-invalid} \; A \; x \; y \)

**Implies** \( \text{hn-invalid} \; A \; x \; y \)

**By** (sep-auto simp: hn-ctxt-def invalid-assn-def entailst-def)

**Lemma b-assn-invalid-merge3: hn-invalid \( A \; x \; y \; \lor \; A \; \text{hn-invalid} \; (b-assn \; A \; P) \; x \; y \)

**Implies** \( \text{hn-invalid} \; A \; x \; y \)

**By** (sep-auto simp: hn-ctxt-def invalid-assn-def entailst-def)

**Lemma b-assn-invalid-merge4: hn-invalid \( (b-assn \; A \; P) \; x \; y \; \lor \; A \; \text{hn-ctxt} \; (b-assn \; A \; P') \; x \; y \)

**Implies** \( \text{hn-invalid} \; (b-assn \; A \; (\lambda x. \; P \; x \; \lor \; P' \; x)) \; x \; y \)

**By** (sep-auto simp: hn-ctxt-def invalid-assn-def entailst-def)

184
$P' \ x \ y$

$\Rightarrow_t \ hn-invalid \ (b-assn \ A \ (\lambda x. \ P \ x \lor P' \ x)) \ x \ y$

by (sep-auto simp: hn-ctxt-def invalid-assn-def entailst-def)

**Lemma b-assn-invalid-merge5:** $hn-ctxt \ (b-assn \ A \ P') \ x \ y \ \lor_A \ hn-invalid \ (b-assn \ A \ P) \ x \ y$

$\Rightarrow_t \ hn-invalid \ (b-assn \ A \ (\lambda x. \ P \ x \lor P' \ x)) \ x \ y$

by (sep-auto simp: hn-ctxt-def invalid-assn-def entailst-def)

**Lemma b-assn-invalid-merge6:** $hn-invalid \ (b-assn \ A \ P) \ x \ y \ \lor_A \ hn-ctxt \ A \ x \ y$

$\Rightarrow_t \ hn-invalid \ A \ x \ y$

by (sep-auto simp: hn-ctxt-def invalid-assn-def entailst-def)

**Lemma b-assn-invalid-merge7:** $hn-ctxt \ A \ x \ y \ \lor_A \ hn-invalid \ (b-assn \ A \ P) \ x \ y$

$\Rightarrow_t \ hn-invalid \ A \ x \ y$

by (sep-auto simp: hn-ctxt-def invalid-assn-def entailst-def)

**Lemmas b-assn-invalid-merge[sepref-frame-merge-rules] =**

- b-assn-invalid-merge1
- b-assn-invalid-merge2
- b-assn-invalid-merge3
- b-assn-invalid-merge4
- b-assn-invalid-merge5
- b-assn-invalid-merge6
- b-assn-invalid-merge7
- b-assn-invalid-merge8
- b-assn-invalid-merge9

**Abbreviation nbn-rel :: nat ⇒ (nat × nat) set**

— Natural numbers with upper bound.

where nbn-rel n ≡ b-rel nat-rel (λx::nat. x<n)

**Abbreviation nbn-assn :: nat ⇒ nat ⇒ nat ⇒ assn**

— Natural numbers with upper bound.

where nbn-assn n ≡ b-assn nat-assn (λx::nat. x<n)
2.1.7 Tool Setup

lemmas [sepref-relprops] =
sepref-relpropI[of IS-LEFT-UNIQUE]
sepref-relpropI[of IS-RIGHT-UNIQUE]
sepref-relpropI[of IS-LEFT-TOTAL]
sepref-relpropI[of IS-RIGHT-TOTAL]
sepref-relpropI[of is-pure]
sepref-relpropI[of IS-PURE Φ for Φ]
sepref-relpropI[of IS-ID]
sepref-relpropI[of IS-BELOW-ID]

lemma [sepref-relprops-simps]:

CONSTRAINT (IS-PURE IS-ID) A ⇒ CONSTRAINT (IS-PURE IS-BELOW-ID) A

CONSTRAINT (IS-PURE IS-ID) A ⇒ CONSTRAINT (IS-PURE IS-LEFT-TOTAL) A

CONSTRAINT (IS-PURE IS-ID) A ⇒ CONSTRAINT (IS-PURE IS-RIGHT-TOTAL) A

CONSTRAINT (IS-PURE IS-BELOW-ID) A ⇒ CONSTRAINT (IS-PURE IS-LEFT-UNIQUE) A

CONSTRAINT (IS-PURE IS-BELOW-ID) A ⇒ CONSTRAINT (IS-PURE IS-RIGHT-UNIQUE) A

by (auto
  simp: IS-ID-def IS-BELOW-ID-def IS-PURE-def IS-LEFT-UNIQUE-def
  simp: IS-LEFT-TOTAL-def IS-RIGHT-TOTAL-def
  simp: single-valued-below-Id)

declare True-implies-equals[sepref-relprops-simps]

lemma [sepref-relprops-transform]: single-valued (R⁻¹) = IS-LEFT-UNIQUE R

by (auto simp: IS-LEFT-UNIQUE-def)

2.1.8 HOL Combinators

lemma hn-if[sepref-comb-rules]:

assumes P: Γ ⇒ Γ1 * hn-val bool-rel a a'

assumes RT: a ⇒ hn-refine (Γ1 * hn-val bool-rel a a') b' Γ2 b b

assumes RE: ¬a ⇒ hn-refine (Γ1 * hn-val bool-rel a a') c' Γ2 c c

assumes IMP: TERM If ⇒ Γ2b ∀a Γ2c ⇒ Γ' Γ'

shows hn-refine Γ (if a' then b' else c') Γ' R (If$a$b$c)

using P RT RE IMP[OF TERMl]

unfolding APP-def PROTECT2-def

by (rule hrn-If)

lemmas [sepref-opt-simps] = if-True if-False

lemma hn-let[sepref-comb-rules]:

186
assumes \( P; \Gamma \Rightarrow t \Gamma 1 \ast \text{hn-ctxt} R v v' \)
assumes \( R: \forall x x'. x = v \Rightarrow \text{hn-refine} (\Gamma 1 \ast \text{hn-ctxt} R x x') (f' x') \)
(\( \Gamma' x x' \)) \( R2 (f x) \)
assumes \( F: \forall x x'. \Gamma' x x' \Rightarrow t \Gamma 2 \ast \text{hn-ctxt} R' x x' \)
shows
\( \text{hn-refine} \Gamma (\text{Let} v' f') (\Gamma 2 \ast \text{hn-ctxt} R' v v') R2 (\text{Let}\$v\$$(\lambda_2 x. f x)) \)
apply (rule \( \text{hn-refine-cons} (\text{OF} P - F \text{entt-refl}) \))
apply (simp)
apply (rule \( R \))
by simp

2.1.9 Basic HOL types

lemma \( \text{hrn-default[sepref-import-param]}: (\text{default}, \text{default}) \in \text{Id by simp} \)

lemma \( \text{unit-hnr[sepref-import-param]}: ((),()) \in \text{unit-rel by auto} \)

lemmas [\( \text{sepref-import-param} \)] =
  param-bool
  param-nat
  param-int

lemmas [\( \text{id-rules} \)] =
  itypeI[Pure.of 0 TYPE (nat)]
  itypeI[Pure.of 0 TYPE (int)]
  itypeI[Pure.of 1 TYPE (nat)]
  itypeI[Pure.of 1 TYPE (int)]
  itypeI[Pure.of numeral TYPE (num \Rightarrow nat)]
  itypeI[Pure.of numeral TYPE (num \Rightarrow int)]
  itype-self[of num.One]
  itype-self[of num.Bit0]
  itype-self[of num.Bit1]

lemma \( \text{param-min-nat[param,sepref-import-param]}: (\text{min},\text{min}) \in \text{nat-rel} \rightarrow \text{nat-rel} \rightarrow \text{nat-rel by auto} \)
lemma \( \text{param-max-nat[param,sepref-import-param]}: (\text{max},\text{max}) \in \text{nat-rel} \rightarrow \text{nat-rel} \rightarrow \text{nat-rel by auto} \)

lemma \( \text{param-min-int[param,sepref-import-param]}: (\text{min},\text{min}) \in \text{int-rel} \rightarrow \text{int-rel} \rightarrow \text{int-rel by auto} \)
lemma \( \text{param-max-int[param,sepref-import-param]}: (\text{max},\text{max}) \in \text{int-rel} \rightarrow \text{int-rel} \rightarrow \text{int-rel by auto} \)

lemma \( \text{uminus-hnr[sepref-import-param]}: (\text{uminus},\text{uminus}) \in \text{int-rel} \rightarrow \text{int-rel by auto} \)

lemma \( \text{nat-param[param,sepref-import-param]}: (\text{nat},\text{nat}) \in \text{int-rel} \rightarrow \text{nat-rel by auto} \)
lemma \( \text{int-param[param,sepref-import-param]}: (\text{int},\text{int}) \in \text{nat-rel} \rightarrow \text{int-rel by auto} \)
2.1.10 Product


lemma prod-assn-precise[constraint-rules]:
  precise P1 \implies precise P2 \implies precise (prod-assn P1 P2)
  apply rule
  apply (clarsimp simp: prod-assn-def star-assoc)
  apply safe
  apply (erule (1) prec-frame) apply frame-inference+
  apply (erule (1) prec-frame) apply frame-inference+
  done

lemma
  precise P1 \implies precise P2 \implies precise (prod-assn P1 P2) — Original proof
  apply rule
  apply (clarsimp simp: prod-assn-def)

proof (rule conjI)
  fix F F' h as a b a' b' ap bp
  assume P1: precise P1 and P2: precise P2
  assume F: (h, as) = P1 a ap * P2 b bp * F \land_A P1 a' ap * P2 b' bp * F'

  from F have (h, as) = P1 a ap * (P2 b bp * F) \land_A P1 a' ap * (P2 b' bp * F')
    by (simp only: mult_assoc)
  with preciseD[OF P1] show a=a'.
  from F have (h, as) = P2 b bp * (P1 a ap * F) \land_A P2 b' bp * (P1 a' ap * F')
    by (simp only: mult_assoc[where 'a=assn] mult.commute[where 'a=assn]
      mult.left-commute[where 'a=assn])
  with preciseD[OF P2] show b=b'.
  qed

lemma intf-of-prod-assn[intf-of-assn]:
  assumes intf-of-assn A TYPE('a) intf-of-assn B TYPE('b)
  shows intf-of-assn (prod-assn A B) TYPE('a * 'b)
  by simp

lemma pure-prod[constraint-rules]:
  assumes P1: is-pure P1 and P2: is-pure P2
  shows is-pure (prod-assn P1 P2)
proof —
  from P1 obtain P1' where P1': \\( x x' \). P1 x x' = \uparrow(P1' x x')
    using is-pureE by blast
  from P2 obtain P2' where P2': \\( x x' \). P2 x x' = \uparrow(P2' x x')
    using is-pureE by blast
show thesis proof
fix \( x x' \)
show prod-assn \( P_1 P_2 x x' \) = 
\[ \uparrow (\text{case } (x, x') \text{ of } ((a_1, a_2), c_1, c_2) \Rightarrow P_1' a_1 c_1 \land P_2' a_2 c_2) \]
unfolding prod-assn-def
apply (simp add: \( P_1' P_2' \) split: prod.split)
done
qed

lemma prod-frame-match[sepref-frame-match-rules]:
assumes \( \text{hn-ctxt } A \ (\text{fst } x) \ (\text{fst } y) \Rightarrow t \ \text{hn-ctxt } A' \ (\text{fst } x) \ (\text{fst } y) \)
assumes \( \text{hn-ctxt } B \ (\text{snd } x) \ (\text{snd } y) \Rightarrow t \ \text{hn-ctxt } B' \ (\text{snd } x) \ (\text{snd } y) \)
shows \( \text{hn-ctxt } (\text{prod-assn } A B) \ x y \Rightarrow t \ \text{hn-ctxt } (\text{prod-assn } A' B') \ x y \)
apply (cases \( x \); cases \( y \); simp)
apply (simp add: hn-ctxt-def)
apply (rule entt-star-mono)
using assms apply (auto simp: hn-ctxt-def)
done

lemma prod-frame-merge[sepref-frame-merge-rules]:
assumes \( \text{hn-ctxt } A \ (\text{fst } x) \ (\text{snd } y) \vee A \ \text{hn-ctxt } A' \ (\text{fst } x) \ (\text{snd } y) \Rightarrow t \ \text{hn-ctxt } A m \ (\text{fst } x) \ (\text{fst } y) \)
assumes \( \text{hn-ctxt } B \ (\text{snd } x) \ (\text{snd } y) \vee A \ \text{hn-ctxt } B' \ (\text{snd } x) \ (\text{snd } y) \Rightarrow t \ \text{hn-ctxt } B m \ (\text{snd } x) \ (\text{snd } y) \)
shows \( \text{hn-ctxt } (\text{prod-assn } A B) \ x y \vee A \ \text{hn-ctxt } (\text{prod-assn } A' B') \ x y \Rightarrow t \ \text{hn-ctxt } (\text{prod-assn } A m B m) \ x y \)
by (blast intro: entt-disjE prod-frame-match
entt-disjD1[OF assms(1)] entt-disjD2[OF assms(1)]
entt-disjD1[OF assms(2)] entt-disjD2[OF assms(2)])

lemma entt-invalid-prod: \( \text{hn-invalid } (\text{prod-assn } A B) \ p p' \Rightarrow t \ \text{hn-ctxt } (\text{prod-assn} \ (\text{invalid-assn } A) \ (\text{invalid-assn } B)) \ p p' \)
apply (simp add: \( \text{hn-ctxt-def invalid-assn-def} \) [abs-def])
apply (rule enttI)
apply clarsimp
apply (cases \( p \); cases \( p' \); auto simp: mod-star-conv pure-def)
done


lemma prod-assn-ctxt: \( \text{prod-assn } A1 A2 x y = z \Rightarrow t \ \text{hn-ctxt } (\text{prod-assn } A1 A2) \ x y = z \)
by (simp add: hn-ctxt-def)

lemma hn-case-prod[sepref-prep-comb-rule,sepref-comb-rules]:
assumes \( \text{FR: } \Gamma \Rightarrow t, \text{hn-ctxt } (\text{prod-assn } P_1 P_2) \ p p' \Rightarrow t \ \Gamma' \)
assumes \( \text{Pair: } \wedge a_1 a_2 a_1' a_2'. [p'=\{a_1',a_2\}] \Rightarrow t \ \text{hn-refine } (\text{hn-ctxt } P_1 a_1' a_1 \ * \ \text{hn-ctxt } P_2 a_2' a_2 \ * \ \Gamma' \ * \ \text{hn-invalid} \)
(prod-assn P1 P2) p′ p) (f a1 a2)
(hn-ctxt P1′ a1′ a1 * hn-ctxt P2′ a2′ a2 * hn-ctxt XX1 p′ p * Γ1′) R (f′
a1′ a2′)
shows hn-refine Γ (case-prod f p) (hn-ctxt (prod-assn P1′ P2′) p′ p * Γ1′)
R (case-prod$(\lambda a b. f′ a b)$p′) (is ?Γ Γ)
apply1 (rule hn-refine-cons-pre[OF FR])
apply1 (cases p; cases p′; simp add: prod-assn-pair-conv)
applyS (simp add: hn-ctxt-def)
applyS simp
applyS (simp add: hn-ctxt-def)
done

lemma hn-case-prod-old:
assumes P: Γ⇒Γ1 * hn-ctxt (prod-assn P1 P2) p′ p
assumes R: \⊆\lambda a1 a2 a1′ a2′. [p′=(a1′,a2′)]
⇒ hn-refine (Γ1 * hn-ctxt P1 a1′ a1 * hn-ctxt P2 a2′ a2 * hn-invalid
(prod-assn P1 P2) p′ p) (f a1 a2)
(Th a1 a1′ a2 a2′) R (f′ a1′ a2′)
assumes M: \subseteq\lambda a1 a1′ a2 a2′. Γh a1 a1′ a2 a2′
⇒ Γ′ * hn-ctxt P1′ a1′ a1 * hn-ctxt P2′ a2′ a2 * hn-ctxt Pxx p′ p
shows hn-refine Γ (case-prod f p) (Γ′ * hn-ctxt (prod-assn P1′ P2′) p′ p)
R (case-prod$(\lambda a b. f′ a b)$p′)
apply1 (cases p; cases p′; simp)
apply1 (rule hn-refine-cons-pre[OF P])
apply (rule hn-refine-preI)
apply (simp add: hn-ctxt-def)
apply (rule hn-refine-cons[OF - R])
apply1 (rule entI)
applyS (sep-auto simp add: hn-ctxt-def)
done

lemma hn-Pair[sepref-fr-rules]: hn-refine
(hn-ctxt P1 x1 x1′ * hn-ctxt P2 x2 x2′)
(return (x1′,x2′))
(hn-invalid P1 x1 x1′ * hn-invalid P2 x2 x2′)
(prod-assn P1 P2)
(RETURN$(Pair$x1$x2))
unfolding hn-refine-def
apply (sep-auto simp: hn-ctxt-def)
apply (rule ent-frame-fwd[OF invalidate-clone[of P1]], frame-inference)
apply (rule ent-frame-fwd[OF invalidate-clone[of P2]], frame-inference)
apply sep-auto
done

lemma fst-hnr[sepref-fr-rules]: (return o fst, RETURN o fst) ∈ \(\prodassn AB\)^d \(\rightarrow_a A\)
by sepref-to-hoare sep-auto
lemma snd-hnr[sepref-fr-rules]: (return o snd, RETURN o snd) ∈ \(\prodassn AB\)^d \(\rightarrow_a B\)
by sepref-to-hoare sep-auto

lemmas [constraint-simps] = prod-assn-pure-conv
lemmas [sepref-import-param] = param-prod-swap

lemma rdomp-prodD[desf!]: rdomp \(\prodassn AB\) \((a,b)\) \(\Rightarrow\) rdomp \(A\) a \(\land\) rdomp \(B\) b
unfolding rdomp-def prod-assn-def
by (sep-auto simp: mod-star-conv)

2.1.11 Option

fun option-assn :: \('a ⇒ 'c ⇒ assn\) ⇒ 'a option ⇒ 'c option ⇒ assn where
option-assn P None None = emp
| option-assn P (Some a) (Some c) = P a c
| option-assn - - - = false

lemma option-assn-simps[simp]:
  option-assn P None \(v'\) = \(\uparrow(\!v'=\!\text{None})\)
  option-assn P \(v\) None = \(\uparrow(\!v=\!\text{None})\)
apply (cases \(v'\), simp-all)
apply (cases \(v\), simp-all)
done

lemma option-assn-alt-def: option-assn R a b =
  (case \((a,b)\) of \((\text{Some} \, x, \text{Some} \, y)\) ⇒ R x y
  | \((\text{None}, \text{None})\) ⇒ emp
  | - ⇒ false)
by (auto split: option.split)

lemma option-assn-pure-conv[constraint-simps]: option-assn \(\text{pure} \, R\) = \(\text{pure} \, ((R)\text{option-rel})\)
apply (intro ext)
apply (rename-tac a c)
apply (case-tac \(\text{pure} \, R, a, c\) rule: option-assn.cases)
by (auto simp: pure-def)

lemma hr-comp-option-conv\[simp, fcomp-norm-unfold\]:
hr-comp (option-assn R) ((R′)option-rel)
= option-assn (hr-comp R R′)
unfolding hr-comp-def[abs-def]
apply (intro ext ext-iffI)
apply solve-entails
apply (case-tac (R,b,c) rule: option-assn.cases)
apply clarsimp-all
apply (sep-auto simp: option-assn-alt-def split: option.splits)
apply (clar simp: option-assn-alt-def split: option.splits; safe)
apply (sep-auto split: option.splits)
apply (intro ent-ex-preI)
apply (rule ent-ex-postI)
apply (sep-auto split: option.splits)
done

lemma option-assn-precise[safe-constraint-rules]:
assumes precise P
shows precise (option-assn P)
proof
fix a a′ p h F F′
assume A: h = option-assn P a p * F ∧ A option-assn P a′ p * F′
thus a=a′ proof (cases (P,a,p) rule: option-assn.cases)
  case (2 - av pv)
  hence \[simp\]: \(a = \text{Some av p = Some pv}\) by simp-all
  from A obtain av′ where \[simp\]: a′ = Some av′ by (cases a′, simp-all)

  from A have h = P av pv * F ∧ A P av′ pv * F′ by simp
  with \(\text{precise P}\) have av=av′ by (rule preciseD)
  thus ?thesis by simp
qed simp-all
qed

lemma pure-option[safe-constraint-rules]:
assumes P: is-pure P
shows is-pure (option-assn P)
proof
from P obtain P′ where \(\forall x x′. P x x′ = \uparrow(P′ x x′)\)
  using is-pureE by blast

show ?thesis proof
fix x x′
show option-assn P x x′ =
  \(\uparrow (\text{case } (x, x′) \text{ of}
  \quad (\text{None,None}) ⇒ \text{True } \left| \ (\text{Some v, Some v′}) ⇒ \text{P′v v′ } \left| \ - ⇒ \text{False}\)
  )\)
apply (simp add: P′ split: prod.split option.split)
lemma \( hn\text{-}ctxt\text{-}option \): \( option\text{-}assn \ A \ x \ y = z \implies hn\text{-}ctxt \ (option\text{-}assn \ A) \ x \ y = z \)
by \((simp \ add: \ hn\text{-}ctxt\text{-}def)\)

lemma \( hn\text{-}case\text{-}option\): \( \text{sepref\text{-}prep\text{-}comb\text{-}rule, sepref\text{-}comb\text{-}rules} \):
\[
\begin{align*}
\text{fixes} & p \ p' P \\
\text{defines \ [simp]} & : INVE \equiv hn\text{-}invalid \ (option\text{-}assn \ P) \ p \ p' \\
\text{assumes} & FR: \Gamma \implies \ (hn\text{-}ctxt \ (option\text{-}assn \ P) \ p \ p' * F) \ f1' \\
\text{assumes} & Rn: p=\text{None} \implies hn\text{-}refine \ (hn\text{-}ctxt \ (option\text{-}assn \ P) \ p \ p' * F) \ f1' \\
\text{assumes} & Rs: \forall x x'. \ [ p=} \text{Some} \ x; p'=\text{Some} \ x' \implies \\
& \quad hn\text{-}refine \ (hn\text{-}ctxt \ P \ x \ x' * INVE * F) \ (f2' x') \ (hn\text{-}ctxt \ P' \ x \ x' * hn\text{-}ctxt XX2) \\
\text{assumes} & MERGE1: \Gamma 1' \lor_A \Gamma 2' \implies \Gamma' \\
\text{shows} & hn\text{-}refine \Gamma \ (case\text{-}option \ f1' f2' p') \ (hn\text{-}ctxt \ (option\text{-}assn \ P') \ p \ p' * \Gamma') \ R \\
\text{apply} & (rule \ hn\text{-}refine\text{-}cons\text{-}pre[OF \ FR]) \\
\text{apply1} & (cases \ p; \ cases \ p'; \ simp \ add: \ option\text{-}assn\text{-}simps[THEN \ hn\text{-}ctxt\text{-}option]) \\
\text{subgoal} & \text{apply} \ (rule \ hn\text{-}refine\text{-}cons[OF - Rn - \text{entt\text{-}refl}]; \ assumption?) \\
\text{applyS} & (simp \ add: \ hn\text{-}ctxt\text{-}def) \\
\text{apply} & (subst \ \text{mult\text{-}commute, rule \ entt\text{-}fr\text{-}drop}) \\
\text{apply} & (rule \ entt\text{-}trans[OF - MERGE1]) \\
\text{apply} & (simp \ add: \ \text{ent\text{-}disjI1' \ ent\text{-}disjI2'}) \\
\text{done} \\
\text{subgoal} & \text{apply} \ (rule \ hn\text{-}refine\text{-}cons[OF - Rs - \text{entt\text{-}refl}]; \ assumption?) \\
\text{applyS} & (simp \ add: \ hn\text{-}ctxt\text{-}def) \\
\text{apply} & (rule \ entt\text{-}star\text{-}mono) \\
\text{apply1} & (rule \ entt\text{-}fr\text{-}drop) \\
\text{applyS} & (simp \ add: \ hn\text{-}ctxt\text{-}def) \\
\text{apply1} & (rule \ entt\text{-}trans[OF - MERGE1]) \\
\text{applyS} & (simp \ add: \ hn\text{-}ctxt\text{-}def) \\
\text{done} \\
\text{done} \\
\text{lemma} \ hn\text{-}None[sepref\text{-}fr\text{-}rules]: \hn\text{-}refine \ emp \ (\text{return} \ \text{None}) \ emp \ (option\text{-}assn \ P) \ \text{(RETURN$\text{None})} \\
\text{by} \ \text{rule sep\text{-}auto} \\
\text{lemma} \ hn\text{-}Some[sepref\text{-}fr\text{-}rules]: \ hn\text{-}refine \\
(hn\text{-}ctxt \ P \ v \ v')
Some v′)
(hn-invalid P v v′)
(option-assn P)
(RETURN$(Some$v))
by rule (sep-auto simp: hn-ctxt-def invalidate-clone′)
definition imp-option-eq eq a b \equiv \begin{cases}
    \text{None},\text{None} & \Rightarrow \text{return True} \\
    \text{Some } a, \text{Some } b & \Rightarrow \text{eq } a \text{ } b \\
    \text{-} & \Rightarrow \text{return False}
\end{cases}

lemma option-assn-eq[sepref-comb-rules]:
fixes a b :: 'a option
assumes F1: \Gamma \Rightarrow t\ hn-ctxt (option-assn P) a a′ * hn-ctxt (option-assn P) b b′
assumes EQ: \forall va va' vb vb'. hn-refine
    (hn-ctxt P va va' * hn-ctxt P vb vb' * \Gamma 1)
    (eq' va' vb')
    (\Gamma' va va' vb vb')
    bool-assn
    (RETURN$(=) $va$vb))
assumes F2:
\forall va va' vb vb'.
\Gamma' va va' vb vb' \Rightarrow t\ hn-ctxt P va va' * hn-ctxt P vb vb' * \Gamma 1
shows hn-refine
\Gamma
(imp-option-eq eq' a' b')
(hn-ctxt (option-assn P) a a' * hn-ctxt (option-assn P) b b' * \Gamma 1)
bool-assn
(RETURN$(=) $a$b))
apply (rule hn-refine-cons-pre[OF F1])
unfolding imp-option-eq-def
apply rule
apply (simp split: option.split add: hn-ctxt-def, intro impI conjI)
apply (sep-auto split: option.split simp: hn-ctxt-def pure-def)
apply (cases a, (sep-auto split: option.split simp: hn-ctxt-def pure-def)+)
apply (cases a, (sep-auto split: option.split simp: hn-ctxt-def pure-def)+)
apply (cases b, (sep-auto split: option.split simp: hn-ctxt-def pure-def)+)
apply (rule cons-post-rule)
apply (rule hn-refineD[OF EQ[unfolded hn-ctxt-def]])
apply simp
apply (rule ent-frame-fwd[OF F2[THEN enttD,unfolded hn-ctxt-def]])
apply (fr-rot 2)
apply (fr-rot-rhs 1)
apply (rule fr-refl)
apply (rule ent-refl)
apply (sep-auto simp: pure-def)
done

lemma \texttt{[pat-rules]}:
\( (= ) \ $a \ None \equiv is-None \ a \)
\( (= ) \ $None \ a \equiv is-None \ a \)
apply (rule eq-reflection, simp split: option.split)+
done

lemma \texttt{hn-is-None[sepref-fr-rules]}: \texttt{hn-refine}
\( (hn-context \ (option-assn \ P) \ a \ a') \)
\( \text{return} \ (is-None \ a') \)
\( \text{bool-assn} \)
\( (RETURN$(is-None\ a)) \)
apply rule
apply (sep-auto split: option.split simp: hn-context-def pure-def)
done

lemma \texttt{(in --) sepref-the-complete[sepref-fr-rules]}:
\texttt{assumes} \( x \neq None \)
\texttt{shows} \texttt{hn-refine}
\( (hn-context \ (option-assn \ R) \ x \ xi) \)
\( \text{return} \ (\text{the} \ xi) \)
\( (hn-invalid \ (option-assn \ R) \ x \ xi) \)
\( (R) \)
\( (RETURN$(\text{the$x$})) \)
using \texttt{assms}
apply \texttt{(cases x)}
apply simp
apply \texttt{(cases xi)}
apply \texttt{(simp add: hn-context-def)}
apply rule
apply \texttt{(sep-auto simp: hn-context-def invalidate-clone’ vassn-tag\! invalid-assn-const)}
done

lemma \texttt{(in --) sepref-the-id}:
\texttt{assumes} \texttt{CONSTRAINT \ (IS-PURE \ IS-ID \ R} \)
\texttt{shows} \texttt{hn-refine}
\( (hn-context \ (option-assn \ R) \ x \ xi) \)
\( \text{return} \ (\text{the} \ xi) \)
\( (hn-context \ (option-assn \ R) \ x \ xi) \)
\( (R) \)
\( (RETURN$(\text{the$x$})) \)
using \texttt{assms}
apply \texttt{(clarsimp simp: IS-PURE-def IS-ID-def hn-context-def is-pure-cone)’}
apply \texttt{(cases x)}
apply simp
apply \texttt{(cases xi)}
apply (simp add: hn-ctxt-def invalid-assn-def)
apply rule apply (sep-auto simp: pure-def)
apply rule apply (sep-auto)
apply (simp add: option-assn-pure-conv)
apply rule apply (sep-auto simp: pure-def invalid-assn-def)
done

2.1.12 Lists

fun list-assn :: (′a ⇒ ′c ⇒ assn) ⇒ ′a list ⇒ ′c list ⇒ assn where
list-assn P [] [] = emp
| list-assn P (a#as) (c#cs) = P a c * list-assn P as cs
| list-assn _ _ _ = false

lemma list-assn-aux-simps[simp]:
list-assn P [] l' = (↑(l'=[]))
list-assn P l [] = (↑(l=[]))
unfolding hn-ctxt-def
apply (cases l')
simp
apply simp
apply (cases l)
simp
apply simp
done

lemma list-assn-aux-append[simp]:
length l1=length l1' ⟹
list-assn P (l1@l2) (l1'@l2')
= list-assn P l1 l1' * list-assn P l2 l2'
apply (induct rule: list-induct2)
simp
apply simp
apply simp add: star-assoc
done

lemma list-assn-aux-ineq-len: length l ≠ length l1 ⟹ list-assn A l l1 = false
proof (induction l arbitrary: l1)
case (Cons x l l1) thus ?case by (cases l1; auto)
qed simp

lemma list-assn-aux-append2[simp]:
assumes length l2=length l2'
sows list-assn P (l1@l2) (l1'@l2')
= list-assn P l1 l1' * list-assn P l2 l2'
apply (cases length l1 = length l1')
apply (erule list-assn-aux-append)
simp add: list-assn-aux-ineq-len assms
done
lemma list-assn-pure-conv: list-assn (pure R) = pure ((R)list-rel)
proof (intro ext)
  fix l li
  show list-assn (pure R) l li = pure ((R)list-rel) l li
    apply (induction pure R l li rule: list-assn.induct)
    by (auto simp: pure-def)
qed

lemmas [sepref-import-rewrite, sepref-frame-normrel-eqs, fcomp-norm-unfold] =
list-assn-pure-conv[symmetric]

lemma list-assn-simps[simp]:
hn-ctxt (list-assn P) [] l' = (↑(l'=[]))
hn-ctxt (list-assn P) l [] = (↑(l=[]))
hn-ctxt (list-assn P) l [] = emp
hn-ctxt (list-assn P) (a#as) (c#cs) = hn-ctxt P a c * hn-ctxt (list-assn P) as cs
hn-ctxt (list-assn P) (a#as) [] = false
hn-ctxt (list-assn P) [] (c#cs) = false
unfolding hn-ctxt-def
apply (cases l')
simp
apply simp
apply (cases l)
simp
apply simp
apply simp-all
done

lemma list-assn-precise[constraint-rules]: precise P ⟹ precise (list-assn P)
proof
  fix l1 l2 l h F1 F2
  assume P: precise P
  assume h| list-assn P l1 l * F1 ∧ A list-assn P l2 l * F2
  thus l1 =l2
proof (induct l arbitrary: l1 l2 F1 F2)
  case Nil thus ?case by simp
next
  case (Cons a ls)
  from Cons obtain a1 ls1 where [simp]: l1=a1#ls1
    by (cases l1, simp)
  from Cons obtain a2 ls2 where [simp]: l2=a2#ls2
    by (cases l2, simp)

  from Cons.prems have M:
    h | P a1 a * list-assn P ls1 ls * F1
      ∧ A P a2 a * list-assn P ls2 ls * F2 by simp
  have a1=a2
    apply (rule preciseD[OF P, where a=a1 and a'=a2 and p=a

197
and $F = \text{list-assn} \ P \ l \ l * F_1$

and $F' = \text{list-assn} \ P \ l \ l * F_2$

)]

using $M$

by (simp add: star-assoc)

moreover have $l_1 = l_2$

apply (rule Cons.hyps[where $?F_1.0 = P \ a1 \ a * F_1$ and $?F_2.0 = P \ a2 \ a * F_2$])

using $M$

by (simp only: star-aci)

ultimately show ?case by simp

qed

lemma list-assn-pure[constraint-rules]:

assumes $P$: is-pure $P$

shows is-pure (list-assn $P$)

proof −

from $P$ obtain $P'$ where P-eq:
    $\forall x \ x'. P x x' = \uparrow (P' x x')$

by (rule is-pureE) blast

{ 
  fix $l \ l'$  
  have list-assn $P \ l \ l' = \uparrow (l \equiv P \ l \ l)$  
    by (induct $P \equiv P \ l \ l'$ rule: list-assn.induct)
    (simp-all add: P-eq)
} thus ?thesis by rule

qed

lemma list-assn-mono:  
  \[ \forall x \ x'. P x x' \Rightarrow A P' x x' \] \Rightarrow \text{list-assn} \ P \ l \ l' \Rightarrow_A \text{list-assn} \ P' \ l \ l'

unfolding \text{hn-ctxt-def}

apply (induct $P \equiv P \ l \ l'$ rule: list-assn.induct)
by (auto intro: ent-star-mono)

lemma list-assn-monot:  
  \[ \forall x \ x'. P x x' \Rightarrow \text{list-assn} \ P \ l \ l' \] \Rightarrow \text{list-assn} \ P' \ l \ l'

unfolding \text{hn-ctxt-def}

apply (induct $P \equiv P \ l \ l'$ rule: list-assn.induct)
by (auto intro: entt-star-mono)

lemma list-match-cong[sepref-frame-match-rules]:

\[ \forall x \ x'. \forall x' \in \text{set} \ l; \forall x' \in \text{set} \ l' \Rightarrow \text{hn-ctxt A} x x' \Rightarrow \text{hn-ctxt A'} x x' \] \Rightarrow \text{hn-ctxt}

(list-assn $A$) \ l \ l' \Rightarrow \text{hn-ctxt (list-assn $A'$)} \ l \ l'

unfolding \text{hn-ctxt-def}

by (induct $A \ l \ l'$ rule: list-assn.induct) (simp-all add: entt-star-mono)

lemma list-merge-cong[sepref-frame-merge-rules]:

assumes $\forall x \ x'. (x \in \text{set} \ l; x' \in \text{set} \ l') \Rightarrow \text{hn-ctxt} A x x' \Rightarrow \text{hn-ctxt} A' x x' \] \Rightarrow \text{hn-ctxt}$

(list-assn $A$) \ l \ l' \Rightarrow \text{hn-ctxt} (list-assn $A'$) \ l \ l'

unfolding \text{hn-ctxt-def}

by (induct $A \ l \ l'$ rule: list-assn.induct) (simp-all add: entt-star-mono)
shows \( hn-ctxt (\text{list-assn} \ A) \ l l' \vee_A hn-ctxt (\text{list-assn} \ A') \ l l' \implies_t hn-ctxt (\text{list-assn} \ A) \ l l' \)

apply (\text{blast intro: entt-disjE list-match-cong entt-disjD1} [OF \text{assms}] entt-disjD2 [OF \text{assms}])
done

lemma invalid-list-split:
invalid-assn (\text{list-assn} \ A) (x\#xs) (y\#ys) \implies_t invalid-assn \ A x y * invalid-assn (\text{list-assn} \ A) xs ys
by (fastforce simp: invalid-assn-def intro: enttI simp: mod-star-conv)

lemma entt-invalid-list: \( hn-invalid (\text{list-assn} \ A) \ l l' \implies_t hn-ctxt (\text{list-assn} \ (invalid-assn \ A)) \ l l' \)
apply (induct \ A \ l \ l' \ rule: list-assn.induct)
applyS simp

subgoal
apply1 (simp add: \text{hn-ctxt-def cong del: invalid-assn-cong})
apply1 (rule entt-trans [OF invalid-list-split])
apply (rule entt-star-mono)
applyS simp

apply (rule entt-trans)
applyS assumption
applyS simp
done

applyS (simp add: \text{hn-ctxt-def invalid-assn-def})
applyS (simp add: \text{hn-ctxt-def invalid-assn-def})
done

lemmas invalid-list-merge [sepref-frame-merge-rules] = gen-merge-cons [OF entt-invalid-list]

lemma list-assn-comp [fcomp-norm-unfold]: \( hr-comp (\text{list-assn} \ A) \ ((B)\text{list-rel}) = \text{list-assn} \ (hr-comp \ A \ B) \)
proof (intro ext)
{ fix \ x \ l \ y \ m
have hr-comp (\text{list-assn} \ A) ((B)\text{list-rel}) (x \# l) (y \# m) = hr-comp \ A \ B \ x \ y * hr-comp (\text{list-assn} \ A) ((B)\text{list-rel}) \ l \ m
by (sep-auto
  simp: hr-comp-def list-rel-split-left-iff
intro!: ent-ex-prel ent-iffI)
}
note aux = this

fix \ l \ li
show hr-comp (\text{list-assn} \ A) ((B)\text{list-rel}) \ l \ li = \text{list-assn} \ (hr-comp \ A \ B) \ l \ li
apply (induction \ l \ arbitrary: \ li; case-tac \ li; intro ent-iffI)
apply (sep-auto simp: hr-comp-def; fail)+
by (simp-all add: aux)

qed

lemma hn-ctxt-eq: A x y = z \implies hn-ctxt A x y = z by (simp add: hn-ctxt-def)

lemmas hn-ctxt-list = hn-ctxt-eq[of list-assn A for A]

lemma hn-case-list[sepref-prep-comb-rule, sepref-comb-rules]:
  fixes p p' P
  defines [simp]: INVE ≡ hn-invalid (list-assn P) p p'
  assumes FR: Γ \implies hn-ctxt (list-assn P) p p' * F
  assumes Rn: p = [] \implies hn-refine (hn-ctxt (list-assn P) p p' * F) f1' (hn-ctxt XX1 p p' * Γ'1') R f1
  assumes Rs: \( \lambda x l x' l'. [ p=x#1; p'=x'#1'] \implies hn-refine (hn-ctxt P x x' * hn-ctxt (list-assn P2') l l' * INVE * F) (f2' x' l') (hn-ctxt P1' x x' * hn-ctxt (list-assn P2') l l' * hn-ctxt XX2 p p' * Γ'2') R (f2 x l)
  assumes MERGE1[unfolded hn-ctxt-def]: \( \lambda x x'. hn-ctxt P1' x x' \lor A \land A \) hn-ctxt P2' x x' \implies \( \lambda x x'. hn-ctxt P1' x x' \lor A \land A \) R (case-list $f1$ (\( \lambda x l. f2 x l \)) p)
  shows hn-refine Γ (case-list f1' f2' p') (hn-ctxt (list-assn P') p p' * Γ') R (case-list$ f1$ (\( \lambda x l. f2 x l \)) p)
apply (rule hn-refine-cons-pre[of FR])
apply1 (extract-hnr-invalids)
apply (cases p; cases p'; simp add: list-assn.simps[THEN hn-ctxt-list!])
subgoal
apply (rule hn-refine-cons[of R - Rn - entt-refl]; assumption?)
applyS (simp add: hn-ctxt-def)

subgoal (rule hn-refine-cons[of R - Rs - entt-refl]; assumption?)
applyS (simp add: hn-ctxt-def)
apply (rule entt-star-mono)
apply1 (rule entt-fr-drop)
apply (rule entt-star-mono)

apply1 (simp add: hn-ctxt-def)
apply1 (rule entt-trans[of OF - MERGE1])
applyS (simp)

apply1 (simp add: hn-ctxt-def)
apply (rule list-assn-monot)
apply1 (rule entt-trans[of OF - MERGE1])
applyS (simp)
apply1 (rule entt-trans[OF - MERGE2])
applyS (simp)
done
done

lemma hn-Nil[sepref-fr-rules]:
hn-refine emp (return []) emp (list-assn P) (RETURN$[])
unfolding hn-refine-def
by sep-auto

lemma hn-Cons[sepref-fr-rules]: hn-refine (hn-ctxt P x x’ * hn-ctxt (list-assn P) xs xs’)
(return (x’#xs’)) (hn-invalid P x x’ * hn-invalid (list-assn P) xs xs’) (list-assn P)
(RETURN$((#) $x$xs))
unfolding hn-refine-def
apply (sep-auto simp: hn-ctxt-def)
apply (rule ent-frame-fwd[OF invalidate-clone[of P]], frame-inference)
apply (rule ent-frame-fwd[OF invalidate-clone[of list-assn P], frame-inference])
apply solve-entails
done

lemma list-assn-aux-len:
list-assn P l l’ = list-assn P l l’ * (length l = length l’)
apply (induct P≡P l l’ rule: list-assn.induct)
apply simp-all
subgoal for a as c cs
  by (erule-tac t=list-assn P as cs in subst[OF sym]) simp
done

lemma list-assn-aux-eqlen-simp:
vassn-tag (list-assn P l l’) ⇒ length l’ = length l
h | (list-assn P l l’) ⇒ length l’ = length l
apply (subst (asm) list-assn-aux-len; auto simp: vassn-tag-def)+
done

lemma hn-append[sepref-fr-rules]: hn-refine (hn-ctxt (list-assn P) l1 l1’ * hn-ctxt (list-assn P) l2 l2’)
(return (l1@l2’)) (hn-invalid (list-assn P) l1 l1’ * hn-invalid (list-assn P) l2 l2’)
(list-assn P)
(RETURN$(@) $l1$l2))
apply rule
apply (sep-auto simp: hn-ctxt-def)
apply (subst list-assn-aux-len)
apply (sep-auto)
apply (rule ent-frame-fwd[OF invalidate-clone[of list-assn P], frame-inference])
apply (rule ent-frame-fwd[OF invalidate-clone[of list-assn P], frame-inference])
apply solve-entails

201
done

**lemma** list-assn-aux-cons-conv1:
list-assn R (a#l) m = (∃A b m’. R a b * list-assn R l m’ * ↑(m=b#m’))
  apply (cases m)
  apply sep-auto
  apply (sep-auto intro! : ent-iffI)
done

**lemma** list-assn-aux-cons-conv2:
list-assn R l (b#m) = (∃A a l’. R a b * list-assn R l’ l m * ↑(l=a#l’))
  apply (cases l)
  apply sep-auto
  apply (sep-auto intro! : ent-iffI)
done

**lemmas** list-assn-aux-cons-conv = list-assn-aux-cons-conv1 list-assn-aux-cons-conv2

**lemma** list-assn-aux-append-conv1:
list-assn R (l1@l2) m = (∃A m1 m2. list-assn R l1 m1 * list-assn R l2 m2 * ↑(m=m1@m2))
  apply (induction l1 arbitrary: m)
  apply (sep-auto intro! : ent-iffI)
  apply (sep-auto intro! : ent-iffI simp: list-assn-aux-cons-conv)
done

**lemma** list-assn-aux-append-conv2:
list-assn R l (m1@m2) = (∃A l1 l2. list-assn R l1 m1 * list-assn R l2 m2 * ↑(l1=l@l2))
  apply (induction m1 arbitrary: l)
  apply (sep-auto intro! : ent-iffI)
  apply (sep-auto intro! : ent-iffI simp: list-assn-aux-cons-conv)
done

**lemmas** list-assn-aux-append-conv = list-assn-aux-append-conv1 list-assn-aux-append-conv2

declare param-apt[sepref-import-param]

2.1.13 Sum-Type

**fun** sum-assn :: (‘a ⇒ ‘b ⇒ assn) ⇒ (‘b ⇒ ‘a ⇒ assn) ⇒ (‘a + ‘b) ⇒ assn where
  sum-assn A B (Inl ai) (Inl a) = A ai a
| sum-assn A B (Inr bi) (Inr b) = B bi b
| sum-assn A B - - = false

**notation** sum-assn (infixr +_a 67)

**lemma** sum-assn-pure[safe-constraint-rules]: [[is-pure A; is-pure B]] → is-pure (sum-assn A B)
  apply (auto simp: is-pure-iff-pure-assn)
  apply (rename_tac x x’)

202
apply (case-tac x; case-tac x'; simp add: pure-def)
done

lemma sum-assn-id [simp]: sum-assn id-assn id-assn = id-assn
apply (intro ext)
subgoal for x y by (cases x; cases y; simp add: pure-def)
done

lemma sum-assn-pure-conv [simp]: sum-assn (pure A) (pure B) = pure ((A,B)sum-rel)
apply (intro ext)
subgoal for a b by (cases a; cases b; auto simp: pure-def)
done

lemma sum-match-cong [sepref-frame-match-rules]:
[ \A x y. [ e = Inl x; e' = Inl y ] \implies hn-ctxt A x y \implies_t hn-ctxt A' x y;
\A x y. [ e = Inr x; e' = Inr y ] \implies hn-ctxt B x y \implies_t hn-ctxt B' x y
] \implies hn-ctxt (sum-assn A B) e e' \implies_t hn-ctxt (sum-assn A' B') e e'
by (cases e; cases e'; simp add: hn-ctxt-def entt-star-mono)

lemma enum-merge-cong [sepref-frame-merge-rules]:
assumes \A x y. [ e = Inl x; e' = Inl y ] \implies hn-ctxt A x y \lor A \implies hn-ctxt A' x y \implies_t hn-ctxt Am x y
assumes \A x y. [ e = Inr x; e' = Inr y ] \implies hn-ctxt B x y \lor A \implies hn-ctxt B' x y \implies_t hn-ctxt Bm x y
shows hn-ctxt (sum-assn A B) e e' \lor A \implies hn-ctxt (sum-assn A' B') e e' \implies_t hn-ctxt (sum-assn Am Bm) e e'
apply (rule entt-disjE)
apply (rule sum-match-cong)
apply (rule entt-disjD1 [OF assms(1)]; simp)
apply (rule entt-disjD1 [OF assms(2)]; simp)
done

lemma entt-invalid-sum; hn-invalid (sum-assn A B) e e' \implies_t hn-ctxt (sum-assn (invalid-assn A) (invalid-assn B)) e e'
apply (simp add: hn-ctxt-def invalid-assn-def [abs-def])
apply (rule enttI)
apply clarsimp
apply (cases e; cases e'; auto simp: mod-star-conv pure-def)
done


sepref-register Inr Inl
lemma [sepref-fr-rules]: \((\text{return} \circ \text{Inl}, \text{return} \circ \text{Inl}) \in A^d \rightarrow a\) sum-assn A B
by sepref-to-hoare sep-auto

lemma [sepref-fr-rules]: \((\text{return} \circ \text{Inr}, \text{return} \circ \text{Inr}) \in B^d \rightarrow a\) sum-assn A B
by sepref-to-hoare sep-auto

sepref-register case-sum

In the monadify phase, this eta-expands to make visible all required arguments

lemma [sepref-monadify-arity]: \(\text{case-sum} \equiv \lambda 2 f1 f2 x. SP \text{case-sum} (\lambda 2 x. f1 x) (\lambda 2 x. f2 x) x\)
by simp

This determines an evaluation order for the first-order operands

lemma [sepref-monadify-comb]: \(\text{EVAL} (\text{case-sum} (\lambda 2 x. f1 x) (\lambda 2 x. f2 x) x) \equiv (\gg \gg) \text{EVAL} x (\lambda 2 x. SP \text{case-sum} (\lambda 2 x. EVAL f1 x) (\lambda 2 x. EVAL f2 x) x)\)
apply (rule eq-reflection)
by (simp split: sum.splits)

Auxiliary lemma, to lift simp-rule over hn-ctxt

lemma sum-assn-ctxt: \(\text{sum-assn } A B x y = z \implies \text{hn-ctxt } \text{sum-assn } A B x y = z\)
by (simp add: hn-ctxt-def)

The cases lemma first extracts the refinement for the datatype from the precondition. Next, it generate proof obligations to refine the functions for every case. Finally the postconditions of the refinement are merged.

Note that we handle the destructed values separately, to allow reconstruction of the original datatype after the case-expression.

Moreover, we provide (invalidated) versions of the original compound value to the cases, which allows access to pure compound values from inside the case.

lemma sum-cases-hnr:
fixes A B e e'
defines [simp]: \(\text{INVe} \equiv \text{hn-invalid } (\text{sum-assn } A B) e e'\)
assumes FR: \(\Gamma = \Rightarrow, \text{hn-ctxt } (\text{sum-assn } A B) e e' * F\)
assumes E1: \(\land x1 x1a. [e = \text{Inl } x1; e' = \text{Inl } x1a] \implies \text{hn-refine } (\text{hn-ctxt } A x1 x1a * \text{INVe} * F) (f1' x1a) (\text{hn-ctxt } A' x1 x1a * \text{hn-ctxt } XX1 e e' * \Gamma' ) R (f1 x1)\)
assumes E2: \(\land x2 x2a. [e = \text{Inr } x2; e' = \text{Inr } x2a] \implies \text{hn-refine } (\text{hn-ctxt } B x2 x2a * \text{INVe} * F) (f2' x2a) (\text{hn-ctxt } B' x2 x2a * \text{hn-ctxt } XX2 e e' * \Gamma' ) R (f2 x2)\)
assumes MERGE[unfolded hn-ctxt-def]: \(\Gamma' \lor_A \Gamma' \Rightarrow_A, \Gamma'\)

204
shows \( \text{hn-refine} \Gamma \ (\text{case-sum} \ f_1' f_2' \ e') \ (\text{hn-ctxt} \ (\text{sum-assn} \ A' B') \ e \ e' * \Gamma) \ R \) \\
\( \text{apply} \ (\text{rule \ hn-refine-cons-pre} \ [\text{OF FR}]) \) \\
\( \text{apply1} \ \text{extract-hnr-invalids} \) \\
\( \text{apply} \ (\text{cases} \ e; \ \text{cases} \ e'; \ \text{simp add: sum-assn.simps[THEN sum-assn-ctxt]}) \) \\
\( \text{subgoal} \) \\
\( \text{apply} \ (\text{rule \ hn-refine-cons} \ [\text{OF - E1 - entt-refl}; \ \text{assumption}]) \) \\
\( \text{applyS} \ (\text{simp add: \ hn-ctxt-def \ entt-disjI1' entt-disjI2'}) \) \\
\( \text{apply1} \ (\text{rule \ entt-fr-drop}) \) \\
\( \text{applyS} \ (\text{simp add: \ entt-disjI1' entt-disjI2'}) \) \\
\( \text{done} \) \\
\( \text{subgoal} \) \\
\( \text{apply} \ (\text{rule \ hn-refine-cons} \ [\text{OF - E2 - entt-refl}; \ \text{assumption}]) \) \\
\( \text{applyS} \ (\text{simp add: \ hn-ctxt-def \ entt-disjI1' entt-disjI2'}) \) \\
\( \text{apply1} \ (\text{rule \ entt-fr-drop}) \) \\
\( \text{applyS} \ (\text{simp add: \ entt-disjI1' entt-disjI2'}) \) \\
\( \text{done} \) \\
\( \text{done} \) \\

After some more preprocessing (adding extra frame-rules for non-atomic postconditions, and splitting the merge-terms into binary merges), this rule can be registered \\
\( \text{lemmas} \ [\text{sepref-comb-rules}] = \text{sum-cases-hnr}[\text{sepref-prep-comb-rule}] \) \\
\( \text{sepref-register} \ \text{isl \ projl \ projr} \) \\
\( \text{lemma} \ \text{isl-hnr}[\text{sepref-fr-rules}]: \ (\text{return o isl,RETURN o isl}) \in (\text{sum-assn} \ A B)^k \rightarrow_a \text{bool-assn} \) \\
\( \text{apply \ sepref-to-hoare} \) \\
\( \text{subgoal for} \ a \ b \ \text{by} \ (\text{cases} \ a; \ \text{cases} \ b; \ \text{sep-auto}) \) \\
\( \text{done} \) \\
\( \text{lemma} \ \text{projl-hnr}[\text{sepref-fr-rules}]: \ (\text{return o projl,RETURN o projl}) \in [\text{isl}]_a (\text{sum-assn} \ A B)^d \rightarrow A \) \\
\( \text{apply \ sepref-to-hoare} \) \\
\( \text{subgoal for} \ a \ b \ \text{by} \ (\text{cases} \ a; \ \text{cases} \ b; \ \text{sep-auto}) \) \\
\( \text{done} \) \\
\( \text{lemma} \ \text{projr-hnr}[\text{sepref-fr-rules}]: \ (\text{return o projr,RETURN o projr}) \in [\text{Not o isl}]_a (\text{sum-assn} \ A B)^d \rightarrow B \) \\
\( \text{apply \ sepref-to-hoare} \) \\
\( \text{subgoal for} \ a \ b \ \text{by} \ (\text{cases} \ a; \ \text{cases} \ b; \ \text{sep-auto}) \)
2.1.14 String Literals

sepref-register PR-CONST String.empty-literal

lemma empty-literal-hnr [sepref-import-param]:
  (String.empty-literal, PR-CONST String.empty-literal) ∈ Id
  by simp

lemma empty-literal-pat [def-pat-rules]:
  String.empty-literal ≡ UNPROTECT String.empty-literal
  by simp

context
  fixes b0 b1 b2 b3 b4 b5 b6 :: bool
  and s :: String.literal
begin

sepref-register PR-CONST (String.Literal b0 b1 b2 b3 b4 b5 b6 s)

lemma Literal-hnr [sepref-import-param]:
  (String.Literal b0 b1 b2 b3 b4 b5 b6 s,
   PR-CONST (String.Literal b0 b1 b2 b3 b4 b5 b6 s)) ∈ Id
  by simp
end

lemma Literal-pat [def-pat-rules]:
  String.Literal $ b0 $ b1 $ b2 $ b3 $ b4 $ b5 $ b6 $ s ≡ UNPROTECT (String.Literal $ b0 $ b1 $ b2 $ b3 $ b4 $ b5 $ b6 $ s)
  by simp
end

2.2 Setup for Foreach Combinator

done

theory Sepref-Foreach
imports Sepref-HOL-Bindings Lib/Pf-Add HOL–Library.Rewrite
begin

2.2.1 Foreach Loops

Monadic Version of Foreach

In a first step, we define a version of foreach where the continuation condition is also monadic, and show that it is equal to the standard version for continuation conditions of the form λx. RETURN (c x)
**definition** FOREACH-inv $xs \Phi s \equiv$

\[ \text{case } s \text{ of } (it, \sigma) \Rightarrow \exists xs'. \; xs = xs' @ it \land \Phi (\text{set } it) \sigma \]

**definition** monadic-FOREACH $R \Phi S c f \sigma 0 \equiv$

\[
\text{do } \{
\text{ASSERT (finite } S); \\
x0 \leftarrow \text{it-to-sorted-list } R S; \\
(\cdot, \sigma) \leftarrow \text{RECT } (\lambda W (xs, \sigma). \text{do } \{
\text{ASSERT (FOREACH-inv } xs0 \Phi (xs, \sigma)); \\
\text{if } xs \neq [] \text{ then do } \{
\text{b } \leftarrow c \sigma; \\
\text{if } b \text{ then }
\text{FOREACH-body } f (xs, \sigma) \gg= W \\
\text{else }
\text{RETURN } (xs, \sigma)
\}) \text{ else RETURN } (xs, \sigma)
\}) \text{ (x0, } \sigma 0); \\
\text{RETURN } \sigma
\}\}
\]

**lemma** FOREACH-oci-to-monadic:

$\text{FOREACHoci } R \Phi S c f \sigma 0 = \text{monadic-FOREACH } R \Phi S (\lambda \sigma. \text{RETURN } (c \sigma)) f \sigma 0$

**unfolding** FOREACHoci-def monadic-FOREACH-def WHILEIT-def WHILEI-body-def

**unfolding** it-to-sorted-list-def FOREACH-cond-def FOREACH-inv-def

**apply** simp

**apply** (if-rule any-cong[THEN cong] | rule refl ext)+

**apply** (simp split: prod.split)

**apply** (rule refl)+

**done**

Next, we define a characterization w.r.t. nfoldli

**definition** monadic-nfoldli $l c f s \equiv$ \[
\text{RECT } (\lambda D (l, s). \text{case } l \text{ of } \\
[] \Rightarrow \text{RETURN } s \\
x \# ls \Rightarrow \text{do } \{
\text{b } \leftarrow c \; s; \\
\text{if } b \text{ then do } \{
\text{s' } \leftarrow f x \; s; \\
D (ls, s') \}
\text{else RETURN } s
\}) \text{ (l, s)}
\]

**lemma** monadic-nfoldli-eq:

$\text{monadic-nfoldli } l c f s = (\text{case } l \text{ of } \\
[] \Rightarrow \text{RETURN } s \\
x \# ls \Rightarrow \text{do } \{
\text{b } \leftarrow c \; s; \\
\text{if } b \text{ then do } \{
\text{s' } \leftarrow f x \; s; \\
D (ls, s') \}
\text{else RETURN } s
\})\text{ (l, s)}$

**apply** (subst monadic-nfoldli-def)
apply (subst RECT-unfold)
apply (tagged-solver)
apply (subst monadic-nfoldli-def[symmetric])
apply simp
done

lemma monadic-nfoldli-simp[simp]:
monadic-nfoldli [] c f s = RETURN s
monadic-nfoldli (x#ls) c f s = do {
  b <- c s;
  if b then f x s >>= monadic-nfoldli ls c f else RETURN s
}
apply (subst monadic-nfoldli-eq, simp)
apply (subst monadic-nfoldli-eq, simp)
done

lemma nfoldli-to-monadic:
nfoldli l c f = monadic-nfoldli l (λx. RETURN (c x)) f
apply (induct l)
apply auto
done

definition nfoldli-alt l c f s ≡ RECT (λD (l, s).
case l of
  [] ⇒ RETURN s
  | x#ls ⇒ do {
      let b = c s;
      if b then do { s' ← f x s; D (ls, s') } else RETURN s
    }
} (l, s)

lemma nfoldli-alt-eq:
nfoldli-alt [] c f s = (nfoldli-alt ls c f s)
apply (subst nfoldli-alt-def)
apply (subst RECT-unfold)
apply (tagged-solver)
apply (subst nfoldli-alt-def[symmetric])
apply simp
done

lemma nfoldli-alt-simp[simp]:
nfoldli-alt [] c f s = RETURN s
nfoldli-alt (x#ls) c f s = do {
  let b = c s;
  if b then f x s >>= nfoldli-alt ls c f else RETURN s
}
apply (subst nfoldli-alt-eq, simp)
apply (subst nfoldli-alt-eq, simp)
done

lemma nfoldli-alt:
(nfoldli::'a list ⇒ ('b ⇒ bool) ⇒ ('a ⇒ 'b ⇒ 'b nres) ⇒ 'b ⇒ 'b nres) = nfoldli-alt

proof (intro ext)
fix l::'a list and c::'b ⇒ bool and f::'a ⇒ 'b ⇒ 'b nres and s::'b
have nfoldli l c f = nfoldli-alt l c f
  by (induct l) auto
thus nfoldli l c f s = nfoldli-alt l c f s by simp
qed

lemma monadic-nfoldli-rec:
monadic-nfoldli x' c f σ ≤⇓ (REC T (λ W (xs, σ).
  ASSERT (FOREACH-inv xs∅ I (xs, σ)))
  (λ- if xs = [] then RETURN (xs, σ)
  else c σ ≫
    (λb. if b then FOREACH-body f (xs, σ) ≫ W
    else RETURN (xs, σ))))

apply (induct x' arbitrary: σ)
apply (subst RECT-unfold, refine-mono)
apply (simp)
apply (rule le-ASSERTI)
apply simp

apply (subst RECT-unfold, refine-mono)
apply (subst monadic-nfoldli-simp)
apply (simp del: conc-Id cong: if-cong)
apply refine-rcg
apply simp
apply (clar simp simp add: FOREACH-body-def)
apply (rule-tac R=br (Pair x') (λ- True) in intro-prgR)
apply (simp add: pw-le-iff refine-pw-simps br-def)

apply (rule order-trans)
apply rprems
apply (simp add: br-def)
done

lemma monadic-nfoldli-arities[seppref-monadify-arity]:
monadic-nfoldli ≡ λ x f. σ. SP (monadic-nfoldli) s (λ x σ. f s σ) σ
by (simp-all)

lemma monadic-nfoldli-comb[sepref-monadify-comb]:
\[ \forall s \ c \ f \ \sigma. \ (\text{monadic-nfoldli}\ s s c f \ \sigma) \equiv \ 
\text{Refine-Basic.bind}(\text{EVAL}\ s) (\lambda_2. \ \text{Refine-Basic.bind}(\text{EVAL}\ \sigma) (\lambda_2. \ SP (\text{monadic-nfoldli}\ s s c f \ \sigma)) \] 
by (simp-all)

lemma list-rel-congD:
assumes A: \((li, l)\in (S)\)list-rel
shows \((li, l)\in (S\cap (\text{set} \ li \times \text{set} \ l))\)list-rel
proof –

\{ fix \ Si0 \ S0 \\
  assume set li \subseteq Si0 set l \subseteq S0 \\
  with A have \((li, l)\in (S\cap (Si0 \times S0))\)list-rel \\
  by (induction rule: list-rel-induct) auto 
\} from this[OF order-refl order-refl] show ?thesis .
qed

lemma monadic-nfoldli-refine[refine]:
assumes L: \((li, l)\in (S)\)list-rel 
and \[ \text{simp}: \ (si, s) \in R \] 
and CR[refine]: \(\forall s. \ (si, s)\in R \implies ci \ s \leq \text{bool-rel} \ (c \ s) \) 
and [refine]: \(\exists x \ s. \ [ (x, s)\in S; \ x\in \text{set} \ l; (si, s)\in R; \ \text{inres} \ (c \ s) \ \text{True} ] \implies fi \ x \ s \leq \text{R} \ (f \ x) \) 
shows monadic-nfoldli \(li\ ci \ fi \ si \leq \text{R} \ (\text{monadic-nfoldli} \ l c f s) \)

supply RELATESI[of \(S\cap (\text{set} \ li \times \text{set} \ l)\), refine-dref-RELATES]
supply RELATESI[of \(R\), refine-dref-RELATES]
unfolding monadic-nfoldli-def
apply (refine-reg bind-refine’)
apply refine-dref-type
apply (vc-solve simp: list-rel-congD[OF L])
done

lemma monadic-FOREACH-itsl:
fixes \( R \ I \ tsl \)
shows \( \text{do} \{ l \leftarrow \text{it-to-sorted-list} \ R \ s; \ \text{monadic-nfoldli} \ l c f \ \sigma \ \} \leq \text{monadic-FOREACH} \ R \ I \ s c f \ \sigma \)
apply (rule refine-IdD)
unfolding monadic-FOREACH-def it-to-sorted-list-def
apply (refine-reg)
apply simp
apply (rule monadic-nfoldli-rec[simplified])
done

210
lemma FOREACHoci-itsl:
fixes R I tsl
shows
do { l ← it-to-sorted-list R s; nfoldli l c f σ }
≤ FOREACHoci R I s c f σ
apply (rule refine-IdD)
unfolding FOREACHoci-def it-to-sorted-list-def
apply refine-rcq
apply simp
apply (rule nfoldli-while)
done

lemma [def-pat-rules]:
FOREACHci ≡ PR-CONST (FOREACHoci (λ-. True) (λ-. True))
FOREACHci$\bar{s}$ ≡ PR-CONST (FOREACHoci (λ-. True) $\bar{s}$)
FOREACH ci $\bar{s}$ ≡ PR-CONST (FOREACHoci (λ-. True) $\bar{s}$)
by (simp-all add: FOREACHci-def FOREACHc-def FOREACH-def [abs-def])

term FOREACHoci R I

lemma id-FOREACHoci [id-rules]: PR-CONST (FOREACHoci R I) ::;
TYPE(\'s set ⇒ (\'d ⇒ bool) ⇒ (\'c ⇒ \'d ⇒ \'d nres) ⇒ \'d ⇒ \'d nres)
by simp

We set up the monadify-phase such that all FOREACH-loops get rewritten
to the monadic version of FOREACH

lemma FOREACH-arities [sepref-monadify-arity]:
PR-CONST (FOREACHoci R I) ≡ \$s$ (\$\lambda x. c x\$) (\$\lambda x. f x\$ $\sigma\$ $\sigma\$)
by (simp-all)

lemma FOREACHoci-comb [sepref-monadify-comb]:
\$\lambda s c f . (PR-CONST (FOREACHoci R I))\$ $\sigma$ (\$\lambda x. c x\$) $\sigma$ $\sigma$
by (simp-all add: FOREACH-oci-to-monadic)

Imperative Version of nfoldli

We define an imperative version of nfoldli. It is the equivalent to the monadic
version in the nres-monad

definition imp-nfoldli l c f s ≡ heapfixp-fun (λD (l,s). case l of
  [] ⇒ return s
| x#ls ⇒ do {
\[
\begin{align*}
&b \leftarrow c \ s; \\
&\text{if } b \text{ then do } \{ s' \leftarrow f \ x \ s; D \ (ls, s') \} \text{ else return } s \\
&\}
\end{align*}
\]

\text{declare} \ \text{imp-nfoldli-def[code def]}

\text{lemma} \ \text{imp-nfoldli-simps[simp,code]}:
\begin{align*}
\text{imp-nfoldli} \ [\ ] c \ f \ s &= \text{return} \ s \\
\text{imp-nfoldli} \ (x \#ls) \ c \ f \ s &= (\text{do} \
\begin{align*}
&b \leftarrow c \ s; \\
&\text{if } b \text{ then do } \{ \\
&s' \leftarrow f \ x \ s; \\
&\text{imp-nfoldli} \ ls \ c \ f \ s' \\
&\} \text{ else return } s
\}
\end{align*}
\)
\end{align*}

apply –

unfolding \ \text{imp-nfoldli-def}
apply (subst heap\_mono\_body\_fixp)
apply pf-mono
apply simp
apply (subst heap\_mono\_body\_fixp)
apply pf-mono
apply simp
done

\text{lemma} \ \text{monadic-nfoldli-refine-aux}:
\begin{align*}
\text{assumes} \ c\text{-ref}: \forall s \ s'. h n\text{-refine} \\
\text{(} \Gamma \ * \ h n\text{-ctxt} \ Rs \ s \ s' \text{)} \\
(c \ s) \\
\text{(} \Gamma \ * \ h n\text{-ctxt} \ Rs \ s \ s' \text{)} \\
\text{bool-assn} \\
(c' \ s')
\end{align*}

\text{assumes} \ f\text{-ref}: \forall x \ x' \ s \ s'. h n\text{-refine} \\
\text{(} \Gamma \ * \ h n\text{-ctxt} \ Rl \ x' \ x * \ h n\text{-ctxt} \ Rs \ s \ s' \text{)} \\
(f \ x \ s) \\
\text{(} \Gamma \ * \ h n\text{-invalid} \ Rl \ x' \ x * \ h n\text{-invalid} \ Rs \ s \ s' \text{)} \ Rs \\
(f' \ x' \ s')

\text{shows} \ \text{hn-refine} \\
\text{(} \Gamma \ * \ h n\text{-ctxt} \ (list-assn \ Rl) \ l' \ l * \ h n\text{-ctxt} \ Rs \ s \ s' \text{)} \\
(\text{imp-nfoldli} \ l \ c \ f \ s) \\
\text{(} \Gamma \ * \ h n\text{-invalid} \ (list-assn \ Rl) \ l' \ l * \ h n\text{-invalid} \ Rs \ s \ s' \text{)} \ Rs \\
(\text{monadic-nfoldli} \ l' \ c' \ f' \ s')
\text{applyF} \ (\text{induct} \ p \equiv Rl \ l' \ l \\
\text{arbitrary; } s \ s' \\
\text{rule: list-assn.induct})
\]
applyF simp
apply (rule hn-refine-cons-post)
apply (rule hn-refine-frame[OF hnr-RETURN-pass])
apply (tactic ⟨⟨ Sepref-Frame.frame-tac (K (K no-tac)) @{context} 1 ⟩⟩)
apply (simp add: hn-ctxt-def ent-true-drop invalid-assn-const)
solved

apply1 weaken-hnr-post
apply1 (simp only: imp-nfoldli-simps monadic-nfoldli-simp)
applyF (rule hnr-bind)
apply1 (rule hn-refine-frame[OF c-ref])
applyS (tactic ⟨⟨ Sepref-Frame.frame-tac (K (K no-tac)) @{context} 1 ⟩⟩)
applyF (rule hnr-If)
applyS (tactic ⟨⟨ Sepref-Frame.frame-tac (K (K no-tac)) @{context} 1 ⟩⟩)
applyF (rule hnr-bind)
apply1 (rule hn-refine-frame[OF f-ref])
apply1 (simp add: assn-assoc)
apply1 (rule ent-imp-entt)
apply1 (fr-rot 1, rule fr-refl)
apply1 (fr-rot 2, rule fr-refl)
apply1 (fr-rot 1, rule fr-refl)
applyS (rule ent-refl)
applyF (rule hn-refine-frame)
applyS rprems
apply1 (simp add: assn-assoc)
apply1 (rule ent-imp-entt)
apply (rule fr-refl)
apply1 (fr-rot 3, rule fr-refl)
apply1 (fr-rot 3, rule fr-refl)
applyS (rule ent-refl)
solved

apply simp
applyS (tactic ⟨⟨ Sepref-Frame.frame-tac (K (K no-tac)) @{context} 1 ⟩⟩)
solved

apply1 (rule hn-refine-frame[OF hnr-RETURN-pass])
applyS (tactic ⟨⟨ Sepref-Frame.frame-tac (K (K no-tac)) @{context} 1 ⟩⟩)
apply1 (simp add: assn-assoc)
applyS (tactic ⟨⟨ Sepref-Frame.merge-tac (K (K no-tac)) @{context} 1 ⟩⟩)
solved
apply (rule enttI)
apply (fr-rot-rhs 1)
apply (fr-rot 3, rule fr-refl)
applyS (fr-rot 3, rule ent-star-mono[rotated]; sep-auto simp: hn-ctxt-def)
solved
applyS (simp add: hn-ctxt-def invalid-assn-def)
applyS (rule, sep-auto)
solved
done

lemma hn-monadic-nfoldli:
assumes FR: P \implies \Gamma * hn-ctxt (list-assn Rl) l' l * hn-ctxt Rs s' s
assumes c-ref: \forall s s'. hn-refine
(\Gamma * hn-ctxt Rs s' s)
(c s)
(\Gamma * hn-ctxt Rs s' s)
bool-assn
(c''s')
assumes f-ref: \forall x x' s s'. hn-refine
(\Gamma * hn-invalid Rl x' x * hn-ctxt Rs s' s)
(f x s)
(\Gamma * hn-invalid Rl x' x * hn-invalid Rs s' s) Rs
(f''x''s'')
shows hn-refine
p
(imp-nfoldli l c f s)
(\Gamma * hn-invalid (list-assn Rl) l' l * hn-invalid Rs s' s)
Rs
(monadic-nfoldli$!c$f''s')
apply (rule hn-refine-cons-pre[OF FR])
unfolding APP-def
apply (rule monadic-nfoldli-refine-aux)
apply (rule c-ref[unfolded APP-def])
apply (rule f-ref[unfolded APP-def])
done

definition
imp-foreach :: ('b \Rightarrow 'c list Heap) \Rightarrow 'b \Rightarrow ('a \Rightarrow bool Heap) \Rightarrow ('c \Rightarrow 'a \Rightarrow 'a Heap) \Rightarrow 'a \Rightarrow 'a Heap
where
imp-foreach tsl s c f \sigma \equiv do \{ l \leftarrow tsl s; imp-nfoldli l c f \sigma \}

lemma heap-fixp-mono[partial-function-mono]:
assumes [partial-function-mono]:
\forall x d. mono-Heap (\lambda x a. B x xa d)
\[ Z \text{ xa. mono-Heap } (\lambda a. B a Z \text{ xa}) \]

shows \( \text{mono-Heap } (\lambda x. \text{ heap.fixp-fun } (\lambda D \sigma. B x D \sigma) \sigma) \)

apply rule
apply (rule ccpo.fixp-mono[OF heap.ccpo, THEN fun-ordD])
apply (rule mono-fun-fun-cnv, erule thin-rl, pf-mono)+
apply (rule fun-ordI)
apply (erule monotoneD[of fun-ord Heap-ord Heap-ord, rotated])
apply pf-mono
done

lemma \text{imp-nfoldli-mono}[partial-function-mono]:
assumes \([partial-function-mono]: \forall x \sigma. \text{mono-Heap } (\lambda f a \text{ fa } x \sigma) \]
shows \( \text{mono-Heap } (\lambda x. \text{ imp-nfoldli } l c (f x) \sigma) \)
unfolding \text{imp-nfoldli-def} by pf-mono

lemma \text{imp-foreach-mono}[partial-function-mono]:
assumes \([partial-function-mono]: \forall x \sigma. \text{mono-Heap } (\lambda f a \text{ fa } x \sigma) \]
shows \( \text{mono-Heap } (\lambda x. \text{ imp-foreach } tsl l c (f x) \sigma) \)
unfolding \text{imp-foreach-def} by pf-mono

lemmas \text{[sepref-opt-simps] = imp-foreach-def}

definition \text{IS-TO-SORTED-LIST } \Omega \text{ Rs Rk tsl } \equiv (\text{tsl,it-to-sorted-list } \Omega ) \in (\text{Rs})^k \rightarrow_a \text{ list-assn } Rk

lemma \text{IS-TO-SORTED-LISTI}:  
assumes \( \text{INDEP } Rk \text{ INDEP } \text{Rs } \text{ INDEP } R \)
assumes \( \text{FR: } P \implies \Gamma * \text{ hn-ctxt } \text{ Rs } s' s * \text{ hn-ctxt } \text{ R } \sigma' \sigma \)
assumes \( \text{STL: } \text{GEN-ALGO } \text{ tsl } (\text{IS-TO-SORTED-LIST } \text{ ordR } \text{ Rs Rk}) \)
assumes \text{c-ref}: \( \forall \sigma \sigma'. \text{hn-refine } (\Gamma * \text{ hn-ctxt } \text{ Rs } s' s * \text{ hn-ctxt } \text{ R } \sigma \sigma' \sigma)\)
(\( c \sigma \))
(\( \Gamma c \sigma' \sigma \))
bool-assn
(\( c' \sigma' \))
assumes \text{C-FR}:  
(\( \forall \sigma' \sigma. \text{TERM monadic-FOREACH } \implies \Gamma c \sigma' \sigma \implies \Gamma * \text{ hn-ctxt } \text{ Rs } s' s * \text{ hn-ctxt } \text{ R } \sigma \sigma' \sigma \)
assumes \text{f-ref}: \( \forall x' x \sigma' \sigma. \text{hn-refine } \)

215
\[(\Gamma \ast \text{hn-context } R s \ast s \ast \text{hn-context } R k \ast x \ast \text{hn-context } P f \ast x \ast \text{hn-context } P f \sigma \ast \sigma)\)
\[(f \ast x \ast \sigma)\]
\[(\Gamma f \ast x \ast \sigma) R\]
\[(f' \ast x' \ast \sigma')\]

**assumes** F-FR: \(\bigwedge x' x \ast \sigma \ast \sigma\)
\[\rightarrow (f x \ast x \ast \sigma \ast \sigma)\]
\[R\]
\[\rightarrow (f' x' \ast x' \ast \sigma' \ast \sigma)\]

**shows** hn-refine
\[\Gamma^{\text{imp-foreach}} tsl s c f \sigma\]
\[(\Gamma \ast \text{hn-context } R s \ast s \ast \text{hn-context } P f \ast x \ast \text{hn-context } P f \sigma \ast \sigma)\]

**proof**

- **from** STL **have** STL: \((tsl, \text{it-to-sorted-list } ordR) \in (R s)^k \rightarrow list-assn R k\)
- **unfolding** GEN-ALGO-def IS-TO-SORTED-LIST-def **by** simp

**show** \(\text{thesis}\)
- **apply** (rule hn-refine-cons-pre[OF FR])
- **apply** weaken-hnr-post
- **unfolding** APP-def PROTECT2-def PR-CONST-def imp-foreach-def
- **apply** (rule hn-refine-ref[OF monadic-FOR EACH-itsl])
- **apply** (rule hn-refine-guessI)
- **apply** (rule hnr-bind)
- **apply** (rule hn-refine-frame)
- **apply** (rule STL[to-hnr, unfolded APP-def])
- **apply** (tactic \(\langle\langle \text{Sepref-Frame.frame-tac } (K \ast (K \text{ no-tac})) @\{\text{context}\} 1\rangle\rangle\))
- **apply** (rule hn-refine-cons-post)
- **apply** (rule c-ref[unfolded APP-def])
- **apply** (rule C-FR)
- **apply** (rule TERMII)
- **apply** weaken-hnr-post
- **apply** (rule hn-refine-cons-post)
- **apply** (rule f-ref[unfolded APP-def])
- **apply** (rule entt-trans[OF F-FR])
- **apply** (rule TERMII)
- **applyS** (tactic \(\langle\langle \text{Sepref-Frame.frame-tac } (K \ast (K \text{ no-tac})) @\{\text{context}\} 1\rangle\rangle\))
- **applyS** (tactic \(\langle\langle \text{Sepref-Frame.frame-tac } (K \ast (K \text{ no-tac})) @\{\text{context}\} 1\rangle\rangle\))

**apply** simp
- **done**

**qed**

**lemma** monadic-nfoldli-assert-aux:

216
assumes set \( l \subseteq S \)
shows monadic-nfoldli \( l \ c \ (\lambda x \ s. \text{ASSERT } (x \in S) \Rightarrow f x \ s) \ s = \text{monadic-nfoldli } l \ c \ f \ s \)
using assms
apply (induction \( l \) arbitrary: \( s \))
apply (auto simp: pw-eq-iff refine-pw-simps)
done

lemmas monadic-nfoldli-assert = monadic-nfoldli-assert-aux[OF order-refl]

lemma nfoldli-arities[sepref-monadify-arity]:
\[
\text{nfoldli} \equiv \lambda s \ c \ f \ x. \text{SP} (\text{nfoldli})(s) (\lambda x. c x) (\lambda x. f x) \sigma
\]
by (simp-all)

lemma nfoldli-comb[sepref-monadify-comb]:
\[
\forall s \ c \ f \ x. (\text{nfoldli})(s) (\lambda x. c x) (\lambda x. f x) \equiv
\text{Refine-Basic.bind}(\text{EVAL}(s))(\lambda x. \text{Refine-Basic.bind}(\text{EVAL}(\sigma))(\lambda x. f x) \sigma)
\]
by (simp-all add: nfoldli-to-monadic)

lemma monadic-nfoldli-refine-aux':
assumes SS: set \( l' \subseteq S \)
assumes c-ref: \( \lambda s' \ s. \text{hn-refine} \)
\( (\Gamma \ast \text{hn-ctxt } R s \ s') \)
\( (c \ s) \)
\( (\Gamma \ast \text{hn-ctxt } R s \ s') \)
\( \text{bool-assn} \)
\( (c' \ s') \)
assumes f-ref: \( \lambda x \ x'. \text{hn-refine } \)
\( (\Gamma \ast \text{hn-ctxt } R l x' x \ast \text{hn-ctxt } R s \ s') \)
\( (f x s) \)
\( (\Gamma \ast \text{hn-ctxt } R l' x' x \ast \text{hn-invalid } R s \ s') \)
\( (f' x' s') \)
assumes merge[sepref-frame-merge-rules]: \( \lambda x \ x'. \text{hn-ctxt } R l' x' x \ast \text{hn-ctxt } R l \)
\( x' x \Rightarrow \text{hn-ctxt } R l'' x' x \)
notes [sepref-frame-merge-rules] = merge-sat2[OF merge]
shows hn-refine
\( (\Gamma \ast \text{hn-ctxt } (\text{list-assn } R l') l' \ast \text{hn-ctxt } R s \ s') \)
\( (\text{imp-nfoldli } l' \ c \ f \ s) \)
\( (\Gamma \ast \text{hn-ctxt } (\text{list-assn } R l'') l' \ast \text{hn-invalid } R s \ s') \)
\( (\text{monadic-nfoldli } l' c' f' s') \)

217
apply1 \( (\text{subst monadic-nfoldli-assert-aux}[OF SS,\text{symmetric}]) \)

applyF (induct \( p \equiv Rl \ l' \ l \)
  arbitrary: \( s \ s' \)
  rule: list-assn.induct)

applyF simp
apply (rule hn-refine-cons-post)
apply (rule hn-refine-frame[OF hnr-RETURN-pass])
apply (tactic \( \langle \langle \text{Sepref-Frame.frame-tac} \ (K \ (K \ no-tac)) \ @\{\text{context} \ 1 \} \rangle \rangle \))
apply (simp add: hn-ctxt-def ent-true-drop)
solved

apply (simp only: imp-nfoldli-simps monadic-nfoldli-simp)
apply (rule hnr-bind)
apply (rule hn-refine-frame[OF c-ref])
apply (tactic \( \langle \langle \text{Sepref-Frame.frame-tac} \ (K \ (K \ no-tac)) \ @\{\text{context} \ 1 \} \rangle \rangle \))

apply (rule hnr-If)
apply (tactic \( \langle \langle \text{Sepref-Frame.frame-tac} \ (K \ (K \ no-tac)) \ @\{\text{context} \ 1 \} \rangle \rangle \))
apply (simp only: nres-monad-laws)
apply (rule hnr-ASSERT)
apply (rule hnr-bind)
apply (rule hn-refine-frame[OF f-ref])
apply assumption
apply (simp add: assn-aci)
apply (rule ent-imp-entt)
apply (fr-rot-rhs 1)
apply (fr-rot 2)
apply (rule fr-refl)
apply (rule fr-refl)
apply (rule fr-refl)
apply (rule ent-refl)

applyF (rule hn-refine-frame)
applyS rprems

focus
apply (simp add: assn-aci)
apply (rule ent-imp-entt)

apply (fr-rot-rhs 1, rule fr-refl)
apply (fr-rot 2, rule fr-refl)
apply (fr-rot 1, rule fr-refl)
apply (rule ent-refl)
solved
solved
focus (simp add: assn-assoc)
apply (rule ent-imp-entt)
apply (rule fr-refl)
apply (rule ent-refl)
solved

apply1 (rule hn-refine-frame[OF hnr-RETURN-pass])
applyS (tactic ⟨⟨ Sepref-Frame.frame-tac (K (K no-tac)) @{context} 1 ⟩⟩)

apply1 (simp add: assn-assoc)
applyS (tactic ⟨⟨ Sepref-Frame.merge-tac (K (K no-tac)) @{context} 1 ⟩⟩)

apply simp
apply (rule ent-imp-entt)
apply solve-entails
apply (rule, sep-auto)
solved
done

lemma hn-monadic-nfoldli-rl[sepref-comb-rules]:
assumes INDEP Rk INDEP Rσ
assumes FR: P ⟹ Γ * hn-ctxt (list-assn Rk) s' s * hn-ctxt Rσ σ' σ
assumes c-ref: \( \land \sigma' \sigma'. \hn-refine \)
(Γ * hn-ctxt Rσ σ' σ)
(c σ)
(Γ c σ')
bool-assn
(c' σ')
assumes C-FR:
\( \land \sigma' \sigma. \TERM \hn-monadic-nfoldli \imp \)
Γ c σ' σ ⟹ Γ * hn-ctxt Rσ σ' σ
assumes f-ref: \( \land x' x \sigma' \sigma. [x' \in \text{set } s] \imp \hn-refine \)
(Γ * hn-ctxt Rk x' x * hn-ctxt Rσ σ' σ)
(f x σ)
(Γ f x' x σ' σ) Rσ
(f' x' σ')
assumes F-FR: \( \land x' x \sigma' \sigma. \TERM \hn-monadic-nfoldli \imp \Gamma f x' x \sigma' σ \imp \)
Γ * hn-ctxt Rk x' x * hn-ctxt Pf σ σ' σ
assumes MERGE: \( \land x x'. \hn-ctxt Rk' x' x \vee_A \hn-ctxt Rk \Gamma x' x \imp \) hn-ctxt Rk'' x' x
shows hn-refine
P
(imp-nfoldli s c f σ)
(Γ * hn-ctxt (list-assn Rk'') s' s * hn-invalid Rσ σ' σ)
Rσ

219
\[(\text{monadic-nfoldli})\]
\[s'' = (\lambda_2 \sigma'. \ c' \sigma')(\lambda_2 x' \sigma'. \ f' \ x' \sigma')s'\]

unfolding \textit{APP-def} \textit{PROTECT2-def} \textit{PR-CONST-def}

apply1 (rule \textit{hn-refine-cons-pre}[OF \textit{FR}])
apply1 \textit{weaken-hnr-post}
applyF (rule \textit{hn-refine-cons}[rotated])
applyF (rule \textit{monadic-nfoldli-refine-aux}[OF \textit{order-refl}])

focus
apply (rule \textit{hn-refine-cons-post})
applyS (rule \textit{c-ref})
apply1 (rule \textit{entt-trans}[OF \textit{C-FR}[OF \textit{TERMI}]])
applyS (rule \textit{entt-refl})
solved

apply1 \textit{weaken-hnr-post}
applyF (rule \textit{hn-refine-cons-post})
applyS (rule \textit{f-ref}; \textit{simp})

apply1 (rule \textit{entt-trans}[OF \textit{F-FR}[OF \textit{TERMI}]])
applyS (tactic (\texttt{Sepref-Frame.frame-tac} \texttt{(K (K no-tac)) @{\texttt{context} I}}))
solved

apply (rule \textit{MERGE})
solved

applyS (tactic (\texttt{Sepref-Frame.frame-tac} \texttt{(K (K no-tac)) @{\texttt{context} I}}))
applyS (tactic (\texttt{Sepref-Frame.frame-tac} \texttt{(K (K no-tac)) @{\texttt{context} I}}))
solved

done

\textbf{lemma} \textit{nfoldli-assert}:
\begin{itemize}
  \item \textit{assumes} set \(l \subseteq S\)
  \item \textit{shows} \textit{nfoldli} \(l \ c \ (\lambda \ x \ s. \ \text{ASSERT} \ (x \in S) \Rightarrow f \ x \ s) \ s = \textit{nfoldli} \ l \ c \ f \ s\)
\end{itemize}
\textit{using} \textit{assms} by (induction \(l\) arbitrary: \(s\)) (auto simp: \textit{pw-eq-iff refine-pw-simps})

\textbf{lemmas} \(\textit{nfoldli-assert}' = \textit{nfoldli-assert}[OF \textit{order-refl}]\)

\textbf{lemma} \textit{fold-eq-nfoldli}:
\[\text{RETURN} \ (\textit{fold} \ f \ l \ s) = \textit{nfoldli} \ l \ (\lambda -. \ True) \ (\lambda x \ s. \ \text{RETURN} \ (f \ x \ s)) \ s\]
apply (induction \(l\) arbitrary: \(s\)) apply (auto) done

\textbf{lemma} \textit{fold-eq-nfoldli-assert}:
\[\text{RETURN} \ (\textit{fold} \ f \ l \ s) = \textit{nfoldli} \ l \ (\lambda -. \ True) \ (\lambda x \ s. \ \text{ASSERT} \ (x \in \textit{set} \ l) \Rightarrow \text{RETURN} \ (f \ x \ s)) \ s\]
by (simp add: \textit{nfoldli-assert}' \textit{fold-eq-nfoldli})

\textbf{lemma} \textit{fold-arity}[\textit{sepref-monadify-arity}]: \textit{fold} \equiv \lambda x \ l \ s. \ \textit{SP} \ \textit{fold}\$(\lambda_2 x \ s. \ f \ x \ s)s)$l$s
by auto

lemma monadify-plain-fold[sepref-monadify-comb]:
\[
\text{EVAL} \left( \text{fold} \left( \lambda x. f x s \right) \right) \equiv (\Rightarrow) \left( \text{EVAL} \right) \left( \lambda l. (\Rightarrow) \left( \text{EVAL} \left( \lambda x. f x s \right) \right) \right)
\]
by (simp add: fold-eq-nfoldli)

lemma monadify-plain-fold-old-rl:
\[
\text{EVAL} \left( \text{fold} \left( \lambda x. f x s \right) \right) \equiv (\Rightarrow) \left( \text{EVAL} \left( \lambda l. (\Rightarrow) \left( \text{EVAL} \left( \lambda x. f x s \right) \right) \right) \right)
\]
by (simp add: fold-eq-nfoldli-assert)

foldli

lemma foldli-eq-nfoldli:
\[
\text{return} \left( \text{foldli} l c f s \right) = \text{nfoldli} l c \left( \text{return} f x s \right)
\]
by (induction l arbitrary: s) auto

lemma foldli-arities[sepref-monadify-arity]:
\[
\text{foldli} \equiv \lambda s. \text{SP} \left( \text{foldli} \right) \left( \lambda x. c x \right) \left( \lambda x. f x s \right)
\]
by (simp)

lemma monadify-plain-foldli[sepref-monadify-comb]:
\[
\text{EVAL} \left( \text{nfoldli} \left( \text{filter} P \left( x \in S \right) \right) \right) \equiv (\Rightarrow) \left( \text{EVAL} \right) \left( \lambda l. (\Rightarrow) \left( \text{EVAL} \left( \lambda x. f x s \right) \right) \right)
\]
by (simp add: foldli-eq-nfoldli)

Deforestation

lemma nfoldli-filter-deforestation:
\[
\text{nfoldli} \left( \text{filter} \ P \left( x \in S \right) \right) \equiv \text{nfoldli} \left( \text{return} f x s \right)
\]
apply (induction xs arbitrary: s) auto
by (auto simp: pw-eq-iff refine-pw-simps)

lemma extend-list-of-filtered-set:
\[
\text{assumes} \ [\text{simp, intro}!] \ : \ \text{finite} \ S \quad \text{and} \ A : \ \text{distinct} \ \text{xs}' \ \text{set} \ \text{xs}' = \ \{ x \in S. \ P \ x \} \\
\text{obtains} \ \text{xs} \ \text{where} \ \text{xs}' = \ \text{filter} \ P \ \text{xs} \ \text{distinct} \ \text{xs} \ \text{set} \ \text{xs} = S
\]
proof -
obtain \text{xs2} \ \text{where} \ \{ x \in S. \ \neg P \ x \} = \ \text{set} \ \text{xs2} \ \text{distinct} \ \text{xs2}
using finite-distinct-list[\text{where} \ A = \{ x \in S. \ \neg P \ x \}] \ by \ auto
with \ A \ \text{have} \ \text{xs}' = \ \text{filter} \ P \ \text{xs}' \ \text{distinct} \ \text{xs}' \ \text{set} \ \text{xs}' = S
by (auto simp: filter-empty-conv)
from that[\text{OF this}] \ \text{show} \ \text{thesis} .
qed
lemma FOREACHc-filter-deforestation:
assumes FIN[simp, intro]: finite S
shows \( \text{FOREACHc } \{ x \in S. \ P x \} \ c \ f \ s \) = \( \text{FOREACHc } S \ c (\lambda x s. \ if \ P x \ then \ f x s \ else \ RETURN s) \ s \)
unfolding FOREACHc-def FOREACHci-def FOREACHoci-by-LIST-FOREACH LIST-FOREACH'-eq
LIST-FOREACH'-def it-to-sorted-list-def
subgoal
proof (induction rule: antisym[consumes 0, case-names 1 2])
  case 1
    then show ?case
    apply (rule le-ASSERTI)
    apply (rule ASSERT-leI, simp)
    apply (rule intro-spec-refine[where R=Id, simplified]; clarsimp)
subgoal for \(xs'\) \(xs\)
    apply (rule rhs-step-bind-SPEC[where R=Id and \(x'=xs\), simplified])
    apply simp
    apply simp (simp add: nfoldli-filter-deforestation)
done
next
  case 2
    then show ?case
    apply (rule le-ASSERTI)
    apply (rule ASSERT-leI, simp; fail)
    apply (rule intro-spec-refine[where R=Id, simplified]; clarsimp)
subgoal for \(xs\)
    apply (rule rhs-step-bind-SPEC[where R=Id and \(x'=\text{filter } P \ xs\), simplified])
    apply simp
    apply (simp add: nfoldli-filter-deforestation)
done
qed
done

lemma FOREACHc-filter-deforestation2:
assumes [simp]: distinct \(xs\)
shows \( \text{FOREACHc } (\text{set } (\text{filter } P \ xs)) \ c \ f \ s \) = \( \text{FOREACHc } (\text{set } xs) \ c (\lambda x s. \ if \ P x \ then \ f x s \ else \ RETURN s) \ s \)
using FOREACHc-filter-deforestation[of set xs, simplified, folded set-filter]

2.2.2 For Loops

partial-function (heap) imp-for :: nat \Rightarrow nat \Rightarrow ('a \Rightarrow bool Heap) \Rightarrow (nat \Rightarrow 'a \Rightarrow 'a Heap) \Rightarrow 'a Heap where
  imp-for i u c f s = (if i \geq u then return s else do \{ ctn <- c s; if ctn then f i s
\[ \text{imp-for} (i + 1) u c f \text{ else return s} \}\]

\textbf{declare \textit{imp-for.simps}[code]}

\textbf{lemma \textit{simp}}:
- \( i \geq u \Rightarrow \text{imp-for} i u c f s = \text{return s} \)
- \( i < u \Rightarrow \text{imp-for} i u c f s = \text{do \{ctn \leftarrow c s; if ctn then f i s \Rightarrow \text{imp-for} (i + 1) u c f \text{ else return s}\}} \)

\textit{by (auto simp: \textit{imp-for.simps})}

\textbf{lemma \textit{imp-nfoldli-deforest}[sepref-opt-simps]}:
- \textit{imp-nfoldli \{l..<u\} c = \text{imp-for} l u c}

\textbf{apply (intro ext)}

\textbf{subgoal for f s}
- \textbf{apply (induction u \text{ - i arbitrary}; l u s)}
- \textbf{apply (simp add: \text{upt-cone-Cons}; fail)}
- \textbf{apply (simp add: \text{upt-cone-Cons})}
- \textbf{apply (fo-rule arg-cong)}

\textit{by (auto cong: if-cong)}

\textit{done}

\textbf{partial-function (heap) \textit{imp-for'} :: nat \Rightarrow nat \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow 'a Heap \Rightarrow 'a \Rightarrow 'a Heap where}
- \( \text{imp-for'} i u f s = (\text{if } i \geq u \text{ then return } s \text{ else } f i s \Rightarrow \text{imp-for'} (i + 1) u f) \)

\textbf{declare \textit{imp-for'.simps}[code]}

\textbf{lemma \textit{simp}}:
- \( i \geq u \Rightarrow \text{imp-for'} i u f s = \text{return s} \)
- \( i < u \Rightarrow \text{imp-for'} i u f s = f i s \Rightarrow \text{imp-for'} (i + 1) u f \)

\textit{by (auto simp: \textit{imp-for'.simps})}

\textbf{lemma \textit{imp-for-imp-for'}[sepref-opt-simps]}:
- \textit{imp-for i u (\lambda -. \text{return True}) = \text{imp-for'} i u}

\textbf{apply (intro ext)}

\textbf{subgoal for f s}
- \textbf{apply (induction u \text{ - i arbitrary}; i u s)}
- \textbf{apply (simp; fail)}
- \textbf{apply simp}
- \textbf{apply (fo-rule arg-cong)}

\textit{by auto}

\textit{done}

\textbf{partial-function (heap) \textit{imp-for-down} :: nat \Rightarrow nat \Rightarrow ('a \Rightarrow \text{bool Heap}) \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow 'a Heap \Rightarrow 'a \Rightarrow 'a Heap where}
- \textit{imp-for-down} l i c f s = \text{do \{let } i = i - 1;\c tn \leftarrow c s; \text{ if ctn then do \{}

223
s ← f i s;
if i > l then imp-for-down l i c f s else return s
}
else return s
}
declare imp-for-down.simps[code]

lemma imp-nfoldli-deforest-down[s pref-opt-simps]:
imp-nfoldli (rev [l..<u]) c =
(λf s. if u ≤ l then return s else imp-for-down l u c f s)
proof (intro ext)
fix f s
show imp-nfoldli (rev [l..<u]) c f s =
(if l ≥ u then return s else imp-for-down l u c f s)
proof cases
assume l ≥ u thus ?thesis by auto
next
assume ¬(l ≥ u) hence l < u by auto
thus ?thesis
apply simp
proof (induction u − l arbitrary: u s)
case 0 thus ?case by auto
next
case (Suc u')
from Suc.prems Suc.hyps(2) have [simp]: rev [l..<u] = (u−1)#rev
[l..<u − 1]
apply simp
apply (subst upt-Suc-append[symmetric])
apply auto
done
show ?case using Suc.hyps(1)[of u − 1] Suc.hyps(2) Suc.prems
apply (subst imp-for-down.simps)
apply (cases l ≤ u − Suc 0)
apply (auto simp: Let-def cong: if-cong)
done
qed
qed
qed

context begin

private fun imp-for-down-induction-scheme :: nat ⇒ nat ⇒ unit where
imp-for-down-induction-scheme l i =
let i = i − 1 in
if i > l then
imp-for-down-induction-scheme l i
else ()
)
\textbf{partial-function} (heap) \textit{imp-for-down}': nat \Rightarrow nat \Rightarrow (nat \Rightarrow 'a \Rightarrow 'a \text{Heap}) \Rightarrow 'a \Rightarrow 'a \text{Heap} where
\begin{align*}
\text{imp-for-down}'\, l \, i \, f \, s &= \text{do } \{ \nonumber \\
&\quad \text{let } i = i - 1; \nonumber \\
&\quad s \leftarrow f \, i \, s; \nonumber \\
&\quad \text{if } i > l \, \text{then } \text{imp-for-down}'\, l \, i \, f \, s \, \text{else return } s \} \nonumber
\end{align*}
declare \textit{imp-for-down}.simps[code]

\textbf{lemma} \textit{imp-for-down-no-cond}[sepref-opt-simps]:
\begin{align*}
\text{imp-for-down} \, l \, u \, (\lambda v. \text{return True}) &= \text{imp-for-down}' \, l \, u \nonumber
\end{align*}
apply (induction \textit{l} \, \textit{u} \, \textit{rule}: \textit{imp-for-down-induction-scheme}.induct)
apply (intro ext)
apply (subst \textit{imp-for-down}.simps)
apply (subst \textit{imp-for-down}'.simps)
apply (simp cong: if-cong)
done
end

\textbf{lemma} \textit{imp-for}'.rule:
\begin{align*}
\text{assumes LESS: } l \leq u \nonumber \\
\text{assumes PRE: } P \Rightarrow A \, I \, l \, s \nonumber \\
\text{assumes STEP: } \forall i \, s. [l \leq i; i < u] \Rightarrow <i \, i \, s > \, f \, i \, s \, <i \, (i+1)> \nonumber \\
\text{shows } <P> \Rightarrow \text{imp-for'} \, l \, u \, f \, s \, <i \, u> \nonumber \\
\text{apply (rule Hoare-Triple.cons-pre-rule[OF PRE])} \nonumber \\
\text{using LESS} \nonumber \\
\text{proof (induction arbitrary: } s \text{ rule: inc-induct}) \nonumber \\
\text{case base thus } ?\text{case by sep-auto} \nonumber \\
\text{next} \nonumber \\
\text{case (step } k) \nonumber \\
\text{show } ?\text{case using step.hyps} \nonumber \\
\text{by (sep-auto heap: STEP step.IH)} \nonumber \\
\text{qed} \nonumber
\end{align*}
This lemma is used to manually convert a fold to a loop over indices.

\textbf{lemma} \textit{fold-idx-conv}: fold \, f \, l \, s = fold \, (\lambda i. \, f \, (l!i)) \, [0..<\text{length } l] \, s
\textbf{proof (induction } l \text{ arbitrary: } s \text{ rule: rev-induct}) \nonumber \\
\text{case Nil thus } ?\text{case by simp} \nonumber \\
\text{next} \nonumber \\
\text{case (snoc } x \, l) \nonumber \\
\text{fix } x \, s \nonumber \\
\text{have } \text{fold} \, (\lambda a. \, f \, ((l \, @ \, [x]) \, ! \, a)) \, [0..<\text{length } l] \, s = \text{fold} \, (\lambda a. \, f \, (l \, ! \, a)) \, [0..<\text{length } l] \, s \nonumber \\
\text{by (rule fold-cong) (simp-all add: nth-append)} \nonumber \\
\text{with snoc show } ?\text{case by simp} \nonumber \\
\text{qed} \nonumber
theory Default-Insts
imports Main
begin

instantiation nat :: default begin
  definition default = (0::nat)
  instance ..
end

instantiation int :: default begin
  definition default = (0::int)
  instance ..
end

instantiation bool :: default begin
  definition default = False
  instance ..
end

instantiation prod :: (default, default) default begin
  definition default = (default, default)
  instance ..
end

instantiation list :: (type) default begin
  definition default = []
  instance ..
end

instantiation option :: (type) default begin
  definition default = None
  instance ..
end

instantiation sum :: (default, type) default begin
  definition default = Inl default
  instance ..
end
end

2.3 Ad-Hoc Solutions

theory Sepref-Improper
imports Sepref-Tool Sepref-HOL-Bindings
This theory provides some ad-hoc solutions to practical problems, that, however, still need a more robust/clean solution

### 2.3.1 Pure Higher-Order Functions

Ad-Hoc way to support pure higher-order arguments

**definition** \( \text{pho-apply} :: (a \Rightarrow b) \Rightarrow a \Rightarrow b \) where [code-unfold,simp]: \( \text{pho-apply} \)

\[ f \, x = f \, x \]

**sepref-register** \( \text{pho-apply} \)

**lemmas** \( \text{fold-pho-apply} = \text{pho-apply-def}[\text{symmetric}] \)

**lemma** \( \text{pure-fun-refine}[\text{sepref-fr-rules}]: \) \( \text{hn-refine} \)

\( (\text{hn-val} \ (A \rightarrow B) \ f \, \xi \ \ast \ \text{hn-val} \ A \ x \ \xi) \)

\( (\text{return} \ (\text{pho-apply}\$f\$\xi)) \)

\( (\text{hn-val} \ (A \rightarrow B) \ f \, \xi \ \ast \ \text{hn-val} \ A \ x \ \xi) \)

\( (\text{pure} \ B) \)

\( (\text{RETURN}\$(\text{pho-apply}\$f\$x)) \)

by \( \text{(sep-auto intro}: \text{hn-refineI simp: pure-def hn-ctxt-def dest: fun-relD}) \)

end

**theory** Sepref

**imports**

- Sepref-Tool
- Sepref-HOL-Bindings

- Sepref-Foreach
- Sepref-Intf-Util
- ../Separation-Logic-Imperative-HOL/Examples/Default-Insts
- Sepref-Improper

begin

end
Chapter 3

The Imperative Isabelle Collection Framework

The Imperative Isabelle Collection Framework provides efficient imperative implementations of collection data structures.

3.1 Set Interface

theory IICF-Set
imports ../../../Sepref
begin

3.1.1 Operations

definition [simp]: op-set-is-empty s ≡ s={}
lemma op-set-is-empty-param[param]: (op-set-is-empty,op-set-is-empty)∈⟨A⟩set-rel
→ bool-rel by auto

context
notes [simp] = IS-LEFT-UNIQUE-def
begin

sepref-decl-op set-empty: {} :: ⟨A⟩set-rel .
sepref-decl-op (no-def) set-is-empty: op-set-is-empty :: ⟨A⟩set-rel → bool-rel .
sepref-decl-op set-member: (∈) :: A → ⟨A⟩set-rel → bool-rel where IS-LEFT-UNIQUE A IS-RIGHT-UNIQUE A .
sepref-decl-op set-delete: λx s. s − {x} :: A → ⟨A⟩set-rel → ⟨A⟩set-rel where IS-LEFT-UNIQUE A IS-RIGHT-UNIQUE A .
sepref-decl-op set-union: (∪) :: ⟨A⟩set-rel → ⟨A⟩set-rel → ⟨A⟩set-rel .
sepref-decl-op set-diff: (\(-\)) :: set-rel \to (A) set-rel \to (A) set-rel
where IS-LEFT-UNIQUE A IS-RIGHT-UNIQUE A.

sepref-decl-op set-subseteq: (\(\subseteq\)) :: (A) set-rel \to (A) set-rel \to bool-rel where
IS-LEFT-UNIQUE A IS-RIGHT-UNIQUE A.

sepref-decl-op set-subset: (\(\subset\)) :: (A) set-rel \to (A) set-rel \to bool-rel where IS-LEFT-UNIQUE
A IS-RIGHT-UNIQUE A.

sepref-decl-op set-pick: RES :: [\(\lambda s. \{\} \neq \{\} \}_f (K) set-rel \to K] by auto

end

3.1.2 Patterns

lemma pat-set-def-pat-rules:
{()} \equiv op-set-empty
{\(\epsilon\)} \equiv op-set-member
Set.insert \equiv op-set-insert
{\(\cup\)} \equiv op-set-union
{\(\cap\)} \equiv op-set-inter
{\(-\)} \equiv op-set-diff
{\(\subseteq\)} \equiv op-set-subseteq
{\(\subset\)} \equiv op-set-subset
by (auto intro!: eq-reflection)

lemma pat-set2-def-pat-rules:
(=) \$s\{} \equiv op-set-is-empty$s
(=) \{\}_s \equiv op-set-is-empty$s

(\(-\)) \$s\$(Set.insert\$\_x\$\}\} \equiv op-set-delete\$\_x\$s
SPEC$(\lambda x. (\epsilon) x\}_s \equiv op-set-pick s
RES$s \equiv op-set-pick s
by (auto intro!: eq-reflection)

locale set-custom-empty =
fixes empty and op-custom-empty :: 'a set
assumes op-custom-empty-def: op-custom-empty = op-set-empty
begin
sepref-register op-custom-empty :: 'ax set

lemma fold-custom-empty:
{()} = op-custom-empty
op-set-empty = op-custom-empty
mop-set-empty = RETURN op-custom-empty
unfolding op-custom-empty-def by simp-all
end
end
3.2 Sets by Lists that Own their Elements

theory ICF-List-SetO
imports ../Intf / ICF-Set
begin

Mineral implementation, only supporting a few operations

definition lso-assn A ≡ hr-comp (list-assn A) (br set (λ-. True))
lemmas [fcomp-norm-unfold] = lso-assn-def [symmetric]
lemma lso-is-pure [safe-constraint-rules]: is-pure A ⇒ is-pure (lso-assn A)
  unfolding lso-assn-def by safe-constraint

lemma lso-empty-aref: (uncurry0 (RETURN []), uncurry0 (RETURN op-set-empty))
  ∈ unit-rel →f (br set (λ-. True)) nres-rel
  by (auto simp: in-br-conv intro: frefI nres-relI)

lemma lso-ins-aref: (uncurry (RETURN oo ((#) )), uncurry (RETURN oo op-set-insert))
  ∈ Id ×v br set (λ-. True) →f (br set (λ-. True)) nres-rel
  by (auto simp: in-br-conv intro: frefI nres-relI)

sepref-decl-impl (no-register) lso-empty: hn-Nil[to-hfref] uses lso-empty-aref .
definition [simp]: op-lso-empty ≡ op-set-empty
lemma lso-fold-custom-empty:
  {} = op-lso-empty
  op-set-empty = op-lso-empty
  by auto
lemmas [sepref-fr-rules] = lso-empty-hnr[folded op-lso-empty-def]
sepref-decl-impl lso-insert: hn-Cons[to-hfref] uses lso-ins-aref .

thm hn-Cons[FCOMP lso-ins-aref]

definition [simp]: op-lso-bex P S ≡ ∃x∈S. P x
lemma fold-lso-bex: Bex ≡ λS P. op-lso-bex P S by auto

definition [simp]: mop-lso-bex P S ≡ ASSERT (∀x∈S. ∃y. P x = RETURN y)
  ⇒ RETURN (∃x∈S. P x = RETURN True)
lemma op-mop-lso-bex: RETURN (op-lso-bex P S) = mop-lso-bex (RETURN o P) S by simp

sepref-register op-lso-bex
lemma lso-bex-arity [sepref-monadify-arity]:
  op-lso-bex ≡ λ2P s. SP op-lso-bex$(λ2x. P$x)§s by (auto intro!: eq-reflection ext)
lemma op-lso-bex-monadify[sepref-monadify-comb];
EVAL$( op-lso-bex$(λx. P x)$s)$ ≡ $(⇒) $(EVAL$s)(λx. map-lso-bex$(λx. EVAL$ P x)$s)$ by simp

definition lso-abex P l ≡ nfoldli l (Not) (λx. P x) False
lemma lso-abex-to-set: lso-abex P l ≤ mop-lso-bex P (set l)
proof −
\{ fix b \\
  have nfoldli l (Not) (λx. P x) b ≤ ASSERT (∀ x∈set l. ∃ y. P x = RETURN y) \\
   apply (induction l arbitrary: b) \\
   applyS simp \\
   applyS (clarsimp simp add: pw-le-iff refine-pw-simps; blast) \\
   done \\
\} from this[of False] show ?thesis by (simp add: lso-abex-def)
qed

locale lso-bex-impl-loc =
  fixes Pi and P :: 'a ⇒ bool nres
  fixes li :: 'c list and l :: 'a list
  fixes A :: 'a ⇒ 'c ⇒ assn
  fixes F :: assn
assumes Prl: \( \forall x xi. [x∈set l] \rightarrow hn-refine (F * hn-ctxt A x xi) (Pi xi) (F * hn-ctxt A x xi) \) bool-assn (P x)
begin
  sepref-register l
  sepref-register P

lemma [sepref-comb-rules]:
  assumes Γ \( \Longrightarrow t F' * F * hn-ctxt A x x i \) \\
  assumes x∈set l \\
  shows hn-refine Γ (Pi xi) (F' * F * hn-ctxt A x xi) bool-assn (P$ x)
using hn-refine-frame[OF Prl[OF assms 2]], of Γ F' assim{1} by (simp add: assim-assoc)

schematic-goal lso-bex-impl:
hn-refine (hn-ctxt (list-assn A) l li * F) (?c) (F * hn-ctxt (list-assn A) l li)
bool-assn (lso-abex P l)
  unfolding lso-abex-def[abs-def]
  by sepref
end
concrete-definition lso-bex-impl uses lso-bex-impl-loc.lso-bex-impl

lemma hn-lso-bex[sepref-prep-comb-rule,sepref-comb-rules]:
assumes FR: Γ \( \Longrightarrow t hn-ctxt (iso-assn A) s li * F \)
assumes \( \text{Prl}: \bigwedge x. [x \in s] \implies \text{hn-refine} \ (F \ast \text{hn-ctxt} \ A \ x \ x_i) \ (\Pi x_i) \ (F \ast \text{hn-ctxt} \ A \ x_i) \) \text{bool-assn} \ (P \ x)  

notes [simp def] = mop-lso-bex-def  

shows \( \text{hn-refine} \ \Gamma \ ((\text{iso-bex-impl \ Pi \ li}) \ (F \ast \text{hn-ctxt} \ (\text{iso-assn} \ A) \ s \ li) \) \text{bool-assn} \)  

apply (rule hn-refine-cons-pre[\( \text{OF \ FR} \)])  
apply (clarsimp simp: \( \text{hn-ctxt-def \ lso-assn-def \ hr-comp-def \ in-br-conv \ hn-pre-ex-conv} \))  
apply (rule hn-refine-preI)  
apply (drule mod-starD; clarsimp)  
apply (rule hn-refine-ref[\( \text{OF \ lso-abex-to-set} \)])  
proof –  
fix \ l \ assume [simp]: \( s=\text{set \ l} \)  
from \( \text{Prl} \) have \( \text{Prl}': \bigwedge x. [x \in \text{set \ l}] \implies \text{hn-refine} \ (F \ast \text{hn-ctxt} \ A \ x \ x_i) \ (\Pi x_i) \ (F \ast \text{hn-ctxt} \ A \ x_i) \) \text{bool-assn} \ (P \ x) \)  
by simp  
show \( \text{hn-refine} \ (\text{list-assn} \ A \ l \ li \ast \ F) \ (\text{iso-bex-impl \ Pi \ li}) \ (\exists A. \ F \ast \text{list-assn} \ A \ ba \ li \ast \ (\text{set \ l} = \text{set \ ba})) \) \text{bool-assn} \)  
(\( \text{iso-abex \ P \ l} \))  
apply (rule hn-refine-cons[OF \( \text{OF \ - \ lso-bex-impl.\refine} \))  
apply simp add: \( \text{hn-ctxt-def; \ rule \ entt-refl} \))  
apply1 unfold-locale apply1 (rule \( \text{Prl}' \)) applyS simp  
applyS (sep-auto intro!: \( \text{enttI \ simp: \ hn-ctxt-def} \))  
applyS (rule \( \text{entt-refl} \))  
done  
qed  

end  

### 3.3 Multiset Interface

theory \( IICF-Multiset \)  
imports ../../\Sepref \begin  

#### 3.3.1 Additions to Multiset Theory

**lemma** rel-mset-Plus-gen:  
assumes rel-mset A m m'  
assumes rel-mset A n n'  
shows rel-mset A \( (m+n) \) \( (m'+n') \)  
using assms  
by induction (auto simp: algebra-simps dest: rel-mset-Plus)  

**lemma** rel-mset-single:  
assumes A x y  
shows rel-mset A \( \{\#x\#\} \) \( \{\#y\#\} \)  
unfolding rel-mset-def  

232
apply (rule exI[where \(x=[x]\)])
apply (rule exI[where \(x=[y]\)])
using assms by auto

lemma rel-mset-Minus:
  assumes BIU: \(\text{bi-unique } A\)
  shows \([\text{rel-mset } A \ m \ n; \ A \ x \ y] \implies \text{rel-mset } A (m-\{\#x\}) (n-\{\#y\})\)
  unfolding rel-mset-def
proof clarsimp
  fix \(ml \ nl\)
  assume \(A: A \ x \ y\)
  assume \(R: \text{list-all2 } A \ ml \ nl\)
  show \(\exists ml'. \text{mset } ml' = \text{mset } ml - \{\#x\} \land \exists nl'. \text{mset } nl' = \text{mset } nl - \{\#y\} \land \text{list-all2 } A \ ml' \ nl')\)
  proof (cases \(x \in \text{set } ml\))
  case False
  have \(y \notin \text{set } nl\) using \(A \ R\)
  apply (auto simp: in-set-cone-decomp list-all2-append2 list-all2-Cons2)
  using False BIU [unfolded bi-unique-alt-def]
  apply (auto dest: left-uniqueD)
  done
  with False \(R\) show \(?thesis\) by (auto simp: diff-single-trivial in-multiset-in-set)
  next
  case True
  then obtain \(ml1 \ ml2\) where \(\text{simp: } ml = ml1 \#x \# ml2\) by (auto simp: in-set-cone-decomp)
  then obtain \(nl1 \ nl2\) where \(\text{simp: } nl = nl1 \#y \# nl2\)
    and \(LA: \text{list-all2 } A \ ml1 \ nl1 \text{ list-all2 } A \ ml2 \ nl2\)
    using \(A \ R\)
    apply (auto simp: in-set-cone-decomp list-all2-append1 list-all2-Cons1)
    using BIU [unfolded bi-unique-alt-def]
    apply (auto dest: right-uniqueD)
    done
  have \(\text{mset } (ml1 \#ml2) = \text{mset } ml \# \{\#x\}\)
    \(\text{mset } (nl1 \#nl2) = \text{mset } nl \# \{\#y\}\)
    using \(R\)
    by (auto simp: algebra-simps add-implies-diff union-assoc)
  moreover have \(\text{list-all2 } A \ (ml1 \#ml2) \ (nl1 \#nl2)\)
    by (rule list-all2-append1) fact+
  ultimately show \(?thesis\) by blast
qed

qed

lemma rel-mset-Minus-gen:
  assumes BIU: \(\text{bi-unique } A\)
  assumes \(\text{rel-mset } A \ m \ m'\)
  assumes \(\text{rel-mset } A \ n \ n'\)
  shows \(\text{rel-mset } A \ (m-\n) \ (m'-\n')\)
  using assms(3,2)

233
apply (induction $R = A$ - - rule: rel-mset-induct)
apply (auto dest: rel-mset-Minus[OF BIU] simp: algebra-simps)
done

lemma per-count:
  assumes bi-unique A
  shows rel-fun (rel-mset A) (rel-fun A (=)) count count
apply (intro rel-funI)
unfolding rel-mset-def
apply clarsimp
subgoal for $x$ $y$ $xs$ $ys$
  apply (rotate_tac, induction $xs$ $ys$ rule: list-all2-induct)
  using assms
  by (auto simp: bi-unique-alt-def left-uniqueD right-uniqueD)
done

3.3.2 Parametricity Setup

definition [to-relAPP]: $mset$-rel $A \equiv p2rel$ (rel-mset (rel2p A))

lemma rel2p-mset[rel2p]: rel2p $(\langle A \rangle mset-rel)$ = rel-mset (rel2p A)
  by (simp add: mset-rel-def)

lemma p2re-mset[p2rel]: p2rel (rel-mset A) = $(p2rel A)$ mset-rel
  by (simp add: mset-rel-def)

lemma mset-rel-empty[simp]:
  $(a,\{\#\}) \in \langle A \rangle mset-rel \iff a = \{\#\}$
  $(\{\#\},b) \in \langle A \rangle mset-rel \iff b = \{\#\}$
  by (auto simp: mset-rel-def p2rel-def rel-mset-def)

lemma param-mset-empty[param]: $(\{\#\},\{\#\}) \in \langle A \rangle mset-rel$
  unfolding mset-rel-def
  apply (simp add: p2rel-def)
  by (rule rel-mset-0)

lemma param-mset-Plus[param]: $(\langle+\rangle,\langle+\rangle) \in \langle A \rangle mset-rel \rightarrow \langle A \rangle mset-rel \rightarrow \langle A \rangle mset-rel$
  apply (rule rel2pD)
  apply (simp add: rel2p)
  apply (intro rel-funI)
  by (rule rel-mset-Plus-gen)

lemma param-mset-add[param]: $(add-mset, add-mset) \in A \rightarrow \langle A \rangle mset-rel \rightarrow \langle A \rangle mset-rel$
  apply (rule rel2pD)
apply (simp add: rel2p)
apply (intro rel-funI)
by (rule rel-mset-Plus)

lemma param-mset-minus[param]: [single-valued A; single-valued \((A^{-1})\)]
\(\Rightarrow ((-), (-)) \in \langle A \rangle\ mset-rel \Rightarrow \langle A \rangle\ mset-rel \Rightarrow \langle A \rangle\ mset-rel\)
apply (rule rel2pD)
apply (simp add: rel2p)
apply (intro rel-funI)
apply (rule rel-mset-Minus-gen)
subgoal apply (unfold IS-LEFT-UNIQUE-def[symmetric])
  by (simp add: prop2p bi-unique-alt-def)
apply (simp; fail)
done

lemma param-count[param]: [single-valued A; single-valued \((A^{-1})\)]
\(\Rightarrow (\text{count}, \text{count}) \in \langle A \rangle\ mset-rel \Rightarrow \langle A \rangle\ mset-rel \Rightarrow \langle A \rangle\ mset-rel\)
apply (rule rel2pD)
apply (simp add: prop2p rel2p)
apply (rule pcr-count)
apply (simp add: bi-unique-alt-def)
done

lemma param-set-mset[param]:
shows \((\text{set-mset}, \text{set-mset}) \in \langle A \rangle\ mset-rel \Rightarrow \langle A \rangle\ set-rel\)
apply (rule rel2pD; simp add: rel2p)
by (rule multiset.set-transfer)

definition [simp]: mset-is-empty m \equiv m = \{\#\}

lemma mset-is-empty-param[param]: \((\text{mset-is-empty, mset-is-empty}) \in \langle A \rangle\ mset-rel \Rightarrow \langle A \rangle\ mset-rel \Rightarrow \text{bool-rel}\)
unfolding mset-rel-def mset-is-empty-def[abs-def]
by (auto simp: p2rel-def rel-mset-def intro: nres-relI)

3.3.3 Operations

sepref-decl-op mset-empty: \{\#\} :: \langle A \rangle\ mset-rel .

sepref-decl-op mset-is-empty: \lambda m. m=\{\#\} :: \langle A \rangle\ mset-rel \Rightarrow \text{bool-rel}
unfolding mset-is-empty-def[symmetric]
apply (rule frefI)
by parametricity

sepref-decl-op mset-insert: add-mset :: A \Rightarrow \langle A \rangle\ mset-rel \Rightarrow \langle A \rangle\ mset-rel .
sepref-decl-op  mset-delete: \( \lambda x. m - \{#x#\} :: A \rightarrow (A)\text{mset-rel} \rightarrow (A)\text{mset-rel} \)
where single-valued A single-valued \((A^{-1})\).

sepref-decl-op  mset-plus: \((+)\cdot\cdot\cdot \text{multiset} \Rightarrow - :: (A)\text{mset-rel} \rightarrow (A)\text{mset-rel} \rightarrow (A)\text{mset-rel} \).
sepref-decl-op  mset-minus: \((-)\cdot\cdot\cdot \text{multiset} \Rightarrow - :: (A)\text{mset-rel} \rightarrow (A)\text{mset-rel} \rightarrow (A)\text{mset-rel} \)
where single-valued A single-valued \((A^{-1})\).

sepref-decl-op  mset-contains: \((\in\#) :: A \rightarrow (A)\text{mset-rel} \rightarrow \text{bool} \)
where single-valued A single-valued \((A^{-1})\).

sepref-decl-op  mset-count: \(\lambda x y. \text{count} y x :: A \rightarrow (A)\text{mset-rel} \rightarrow \text{nat-rel} \)
where single-valued A single-valued \((A^{-1})\).

sepref-decl-op  mset-pick: \(\lambda m. \text{SPEC} (\lambda(x,m'). m = \{#x#\} + m') :: [\text{Am. } m\not\notin\{\#\}]f (A)\text{mset-rel} \rightarrow A \times_r (A)\text{mset-rel} \)
unfolding mset-is-empty-def[ symmetric] apply (intro frefl mres-refl)
apply (refine-veq SPEC-refine)
apply1 (rule ccontr; clarsimp)
applyS (metis mset-rel-invL rel2p-def rel2p-mset union-ac(2))
applyS parametricity
done

3.3.4 Patterns

lemma [def-pat-rules]:
\(\{\#\} \equiv \text{op-mset-empty} \)
add-mset \equiv \text{op-mset-insert} \)
\((=) \text{sb}\{\#\} \equiv \text{op-mset-contains}\text{sb} \)
\((=) \text{s}\{\#\}\text{sb} \equiv \text{op-mset-contains}\text{sb} \)
\((+) \text{sa}\text{sb} \equiv \text{op-mset-plus}\text{sa}\text{sb} \)
\((-) \text{sa}\text{sb} \equiv \text{op-mset-minus}\text{sa}\text{sb} \)
by (auto intro!: eq-reflection simp: algebra-simps)

lemma [def-pat-rules]:
\((+) \text{sb}((\text{add-mset}\text{x}\{\#\})) \equiv \text{op-mset-insert}\text{x}\text{sb} \)
\((+) \text{s}(\text{add-mset}\text{x}\{\#\})\text{sb} \equiv \text{op-mset-insert}\text{x}\text{sb} \)
\((-) \text{sb}((\text{add-mset}\text{x}\{\#\})) \equiv \text{op-mset-delete}\text{x}\text{sb} \)
\(<) \text{sb}(\text{count}\text{x}\text{sa}) \equiv \text{op-mset-contains}\text{x}\text{sa} \)
\((\in) \text{x}\text{sa}(\text{set}\text{mset}\text{sa}) \equiv \text{op-mset-contains}\text{x}\text{sa} \)
by (auto intro!: eq-reflection simp: algebra-simps)

locale  mset-custom-empty =

fixes \text{rel empty and op-custom-empty :: 'a multiset}
assumes customize-hir-aux: \(\text{uncurry0} \emptyset, \text{uncurry0} \text{RETURN} \text{op-mset-empty} :: 'a \)
assumes \( \text{op-custom-empty-def} : \text{op-custom-empty} = \text{op-mset-empty} \)

begin
sepref-register \( \text{op-custom-empty} :: \text{'}ax \text{ multiset} \)

lemma fold-custom-empty:
\( \{\#\} = \text{op-custom-empty} \)
\( \text{op-mset-empty} = \text{op-custom-empty} \)
\( \text{mop-mset-empty} = \text{RETURN op-custom-empty} \)

unfolding \( \text{op-custom-empty-def} \) by simp-all

end

end

3.4 Priority Bag Interface

theory IICF-Prio-Bag
imports IICF-Multiset
begin

3.4.1 Operations

We prove quite general parametricity lemmas, but restrict them to relations below identity when we register the operations.
This restriction, although not strictly necessary, makes usage of the tool much simpler, as we do not need to handle different prio-functions for abstract and concrete types.

custom context
fixes prio:: \( \text{'}a \Rightarrow \text{'}b \text{ linorder} \)
begin

definition mop-prio-pop-min \( b \equiv \text{ASSERT} (b \neq \{\#\}) \Rightarrow \text{SPEC} (\lambda (v, b^\prime). \)
\( v \in \# \ b \)
\( \land \ b^\prime \neq b - \{\#v\#\} \)
\( \land \ (\forall v^\prime \in \text{set-mset} b. \text{prio } v \leq \text{prio } v^\prime) \)

definition mop-prio-peek-min \( b \equiv \text{ASSERT} (b \neq \{\#\}) \Rightarrow \text{SPEC} (\lambda v. \)
\( v \in \# \ b \)
\( \land \ (\forall v^\prime \in \text{set-mset} b. \text{prio } v \leq \text{prio } v^\prime) \)
end

lemma param-mop-prio-pop-min[param]:
assumes \( \text{param} : (\text{prio}^\prime, \text{prio}) \in A \rightarrow B \)
assumes \( \text{param} : ((\leq),(\leq)) \in B \rightarrow B \rightarrow \text{bool-rel} \)
show \( (\text{mop-prio-pop-min } \text{prio}^\prime,\text{mop-prio-pop-min } \text{prio}) \in (A)\text{mset-rel} \rightarrow (A \times_f (A)\text{mset-rel})\text{nres-rel} \)
unfolding mop-prio-pop-min-def[abs-def]
apply (clarsimp simp: mop-prio-pop-min-def nres-rel-def pw-le-iff refine-pw-simps)
apply (safe; simp)
proof goal-cases
  case (1 m n x)
    assume (m,n)\in(A)mset-rel
    and x\in#m
    and P': \forall x'\in set-mset m. prio' x \leq prio' x'
    hence R: \text{rel-mset (rel2p A) m n by (simp add: mset-rel-def p2rel-def)}
    from multi-member-split[OF \text{OF} \langle x\in #m \rangle] obtain m'where
by auto
    from msed-rel-invL[OF R\ simplified] obtain n'y where
      \text{simp}: n = n' + \#y# and \text{param, simp}: (x,y)\in A and R': (m',n')\in(A)mset-rel
    by (auto simp: rel2p-def mset-rel-def p2rel-def)
    have \forall y'\in set-mset n. prio y \leq prio y'
      proof
        fix y' assume y'\in set-mset n
        then obtain x' where \text{param}: (x',y')\in A and x'\in set-mset m
        using R
        by (metis insert-DiffM msed-rel-invR rel2pD union-single-eq-member)
        with P' have prio' x \leq prio' x' by blast
        moreover have (prio' x \leq prio' x', prio y \leq prio y') \in bool-rel
        by parametricity
        ultimately show prio y \leq prio y' by simp
      qed
    thus
      \exists a. (x, a) \in A \land (m - \#x#, n - \#a#) \in (A)mset-rel \land a \in# n \land
      (\forall v'\in set-mset n. prio a \leq prio v')
    using R' by (auto intro!: exI[where x=n'] exI[where x=y])
    qed

lemma param-mop-prio-peek-min[.param]:
  assumes [param]: (prio',prio) \in A \rightarrow B
  assumes [param]: ((\leq),(\leq)) \in B \rightarrow B \rightarrow bool-rel
  shows (mop-prio-peek-min prio',mop-prio-peek-min prio) \in (A)mset-rel \rightarrow (A)nres-rel
unfolding mop-prio-peek-min-def[abs-def]
apply (clarsimp simp: mop-prio-pop-min-def nres-rel-def pw-le-iff refine-pw-simps)
apply (safe; simp?)
proof -
  fix m n x
  assume (m,n)\in(A)mset-rel
  and x\in#m
  and P': \forall x'\in set-mset m. prio' x \leq prio' x'
  hence R: \text{rel-mset (rel2p A) m n by (simp add: mset-rel-def p2rel-def)}

238
from multi-member-split[\(\{x \in \#m\}\)] obtain \(m\)' where \([\text{simp}]: m = m' + \{\#x\}\)
by auto

from mased-rel-invL[(\(\{x \in \#m\}\)] obtain \(n\)' \(y\) where
\([\text{simp}]: n = n' + \{\#y\}\] and \([\text{param, simp}]: (x,y) \in A \text{ and } R': (m',n') \in \langle A \rangle \text{mset-rel}
by (auto simp: rel2p-def mset-rel-def p2rel-def)

have \(\forall y' \in \text{set-mset } n. \text{ prio } y \leq \text{ prio } y'\)
proof
fix \(y'\) assume \(y' \in \text{set-mset } n\)
then obtain \(x'\) where \([\text{param}]: (x',y') \in A \text{ and } x' \in \text{set-mset } m\)
using \(R\)
by (metis mased-rel-invR mset-contains-eq rel2pD union-mset-add-mset-left union-single-eq-member)
with \(P'\) have \(\text{prio } x \leq \text{ prio } x'\) by blast
moreover have \((\text{prio } x \leq \text{ prio } x', \text{ prio } y \leq \text{ prio } y') \in \text{bool-rel}\)
by parametricity
ultimately show \(\text{prio } y \leq \text{ prio } y'\) by simp
qed
thus \(\exists y. (x, y) \in A \land y \in \# n \land (\forall v' \in \text{set-mset } n. \text{ prio } y \leq \text{ prio } v')\)
using \(R'\) by (auto intro: exI [where \(x=y\)])
qed

context fixes \(\text{prio :: } 'a \Rightarrow 'b::linorder \text{ and } A :: (\'a \times 'a) \text{ set begin}
sepref-decl-op (no-def, no-mop) prio-pop-min:
\(\text{PR-CONST} (\text{mop-prio-pop-min prio }) :: \langle A \rangle \text{mset-rel} \rightarrow_f \langle A \times, (A) \text{mset-rel} \rangle \text{nres-rel}\)
where \(\text{IS-BELOW-ID } A\)
proof goal-cases
  case 1
  hence \([\text{param}]: (\text{prio, prio}) \in A \rightarrow Id\)
  by (auto simp: IS-BELOW-ID-def)
  show \(?case\)
  apply (rule fref-ncI)
  apply parametricity
  by auto
qed

sepref-decl-op (no-def, no-mop) prio-peek-min:
\(\text{PR-CONST} (\text{mop-prio-peek-min prio }) :: \langle A \rangle \text{mset-rel} \rightarrow_f \langle A \text{mset-rel} \rangle \text{nres-rel}\)
where \(\text{IS-BELOW-ID } A\)
proof goal-cases
  case 1
  hence \([\text{param}]: (\text{prio, prio}) \in A \rightarrow Id\)
  by (auto simp: IS-BELOW-ID-def)
  show \(?case\)
  apply (rule fref-ncI)

apply parametricity
by auto
qed
end

3.4.2 Patterns

lemma [def-pat-rules]:
mop-prio-pop-min$prio ≡ UNPROTECT (mop-prio-pop-min prio)
mop-prio-peek-min$prio ≡ UNPROTECT (mop-prio-peek-min prio)
by auto
end

3.5 Multisets by Lists

theory IICF-List-Mset
imports ../Intf/IICF-Multiset
begin

3.5.1 Abstract Operations

definition list-mset-rel ≡ br mset (λ· True)

lemma lms-empty-aref: ([], op-mset-empty) ∈ list-mset-rel
unfolding list-mset-rel-def by (auto simp: in-br-conv)

lemma lms-is-empty-aref: (is-Nil, op-mset-is-empty) ∈ list-mset-rel → bool-rel
unfolding list-mset-rel-def by (auto simp: in-br-conv split: list.splits)

lemma lms-insert-aref: ((#), op-mset-insert) ∈ Id → list-mset-rel → list-mset-rel
unfolding list-mset-rel-def by (auto simp: in-br-conv)

lemma lms-union-aref: (@), op-mset-plus) ∈ list-mset-rel → list-mset-rel → list-mset-rel
unfolding list-mset-rel-def by (auto simp: in-br-conv)

lemma lms-pick-aref: (λx#l ⇒ RETURN (x,l), mop-mset-pick) ∈ list-mset-rel → (Id ×r list-mset-rel)nres-rel
unfolding list-mset-rel-def mop-mset-pick-alt[abs-def]
apply1 (refine-vcg nres-reI fun-reI)
apply1 (clarsimp simp: in-br-conv neq-Nil-conv)
apply1 (refine-vcg RETURN-SPEC-refine)
applyS (clarsimp simp: in-br-conv algebra-simps)
donedefinition list-contains x l ≡ list-ex ((=) x) l

240
lemma lms-contains-aref: (list-contains, op-mset-contains) ∈ Id → list-mset-rel → bool-rel
  unfolding list-mset-rel-def list-contains-def[abs-def]
  by (auto simp: in-br-conv list-ex-iff in-multiset-in-set)

fun list-remove1 :: 'a ⇒ 'a list ⇒ 'a list
  where
    list-remove1 x [] = []
    | list-remove1 x (y#ys) = (if x=y then ys else y#list-remove1 x ys)

lemma mset-list-remove1[simp]: mset (list-remove1 x l) = mset l − {#x#}
  apply (induction l)
  applyS simp
  by (clarsimp simp: algebra-simps)

lemma lms-remove-aref: (list-remove1, op-mset-delete) ∈ Id → list-mset-rel → list-mset-rel
  unfolding list-mset-rel-def by (auto simp: in-br-conv)

fun list-count :: 'a ⇒ 'a list ⇒ nat
  where
    list-count - [] = 0
    | list-count x (y#ys) = (if x=y then 1 + list-count x ys else list-count x ys)

lemma mset-list-count[simp]: list-count x ys = count (mset ys) x
  by (induction ys) auto

lemma lms-count-aref: (list-count, op-mset-count) ∈ Id → list-mset-rel → nat-rel
  unfolding list-mset-rel-def by (auto simp: in-br-conv)

definition list-remove-all :: 'a list ⇒ 'a list ⇒ 'a list
  where
    list-remove-all xs ys ≡ fold list-remove1 ys xs

lemma list-remove-all-mset[simp]: mset (list-remove-all xs ys) = mset xs − mset ys
  unfolding list-remove-all-def
  by (induction ys arbitrary: xs) (auto simp: algebra-simps)

lemma lms-minus-aref: (list-remove-all, op-mset-minus) ∈ list-mset-rel → list-mset-rel → list-mset-rel
  unfolding list-mset-rel-def by (auto simp: in-br-conv)

### 3.5.2 Declaration of Implementations

definition list-mset-assn A ≡ pure (list-mset-rel O (the-pure A)mset-rel)

declare list-mset-assn-def[symmetric,fcomp-norm-unfold]

lemma [safe-constraint-rules]: is-pure (list-mset-assn A) unfolding list-mset-assn-def
  by simp

definition simp: op-list-mset-empty ≡ op-mset-empty

lemma lms-fold-custom-empty:
{#} = op-list-mset-empty

op-mset-empty = op-list-mset-empty

by auto

sepref-register op-list-mset-empty


— Some extra work is required for nondeterministic ops

lemma lms-pick-aref':
(λx#1 ⇒ return (x,l), mop-mset-pick) ∈ (pure list-mset-rel) → prod-assn
id-assn (pure list-mset-rel)

apply (simp only: prod-assn-pure-conv)

apply sepref-to-hoare

apply (sep-auto simp: refine-pw-simps list-mset-rel-def in-br-conv algebra-simps

eintro del: exI)

done

sepref-decl-impl (ismop) lms-pick: lms-pick-aref' .


end

theory IICF-List-MsetO

imports ..../Intf/IICF-Multiset

begin

definition lmso-assn A ≡ hr-comp (list-assn A) (br mset (λ-. True))

lemmas [fcomp-norm-unfold] = lmso-assn-def[symmetric]

lemma lmso-is-pure[safe-constraint-rules]: is-pure A ⇒ is-pure (lmso-assn A)

unfolding lmso-assn-def by safe-constraint

lemma lmso-empty-aref: (uncurry0 (RETURN []), uncurry0 (RETURN op-mset-empty))
∈ unit-rel → f (br mset (λ-. True)) nres-rel

by (auto intro!: frefl nres-relI simp: in-br-conv)

lemma lmso-is-empty-aref: (RETURN o List.null, RETURN o op-mset-is-empty)

242
∈ br mset (λ-. True) → _ f⟨bool-rel⟩ nres-rel
by (auto intro!: frefI nres-relI simp: in-br-conv List.null-def split: list.split)

lemma lmso-insert-aref: (uncurry (RETURN oo (#)), uncurry (RETURN oo op-mset-insert)) ∈ [(Id ×v br mset (λ-. True)) → _ f ⟨br mset (λ-. True)⟩ nres-rel]
by (auto intro!: frefI nres-relI simp: in-br-conv List.null-def split: list.split)

definition [simp]: hd-tl l ≡ (hd l, tl l)

lemma hd-tl-opt[sepref-opt-simps]: hd-tl l = (case l of (x#xs) ⇒ (x,xs) | - ⇒ CODE-ABORT (λ-. (hd l, tl l)))
by (auto split: list.split)

lemma lmso-pick-aref: (RETURN o hd-tl,op-mset-pick) ∈ [λm. m≠{#}]f br mset (λ-. (hd l, tl l)) → ⟨Id ×v br mset (λ-. True)⟩ nres-rel
by (auto intro!: frefI nres-relI simp: in-br-conv pw-le-iff refine-pw-simps neq-Nil-conv algebra-simps)

lemma hd-tl-hnr: (return o hd-tl,RETURN o hd-tl) ∈ [λl. ¬is-Nil l]a (list-assn A)d → prod-assn A (list-assn A)
apply sepref-to-hoare
subgoal for l li by (cases l; cases li; sep-auto)
done


definition [simp]: op-lmso-empty ≡ op-mset-empty
sepref-register op-lmso-empty
lemma lmso-fold-custom-empty:
{#} = op-lmso-empty
op-mset-empty = op-lmso-empty
mop-mset-empty = RETURN op-lmso-empty
by auto

lemma list-null-hnr: (return o List.null,RETURN o List.null) ∈ (list-assn A)k
→a bool-assn
apply sepref-to-hoare
subgoal for l li by (cases l; cases li; sep-auto simp: List.null-def)
done


context notes \( \text{[simp]} = \text{in-br-conv and [split]} = \text{list.splits} \) begin

Dummy lemma, to exploit `sepref-decl-impl` automation without parametricity stuff.

**Private lemma** \( \text{op-mset-pick-dummy-param: (op-mset-pick, op-mset-pick) ∈ Id → } (\text{Id} \text{nres-rel}) \)

by (auto intro!: frefl nres-rell)

**sepref-decl-impl** \( \text{lmso-pick: hd-tl-hnr[FCOMP lmso-pick-aref]} \) uses \( \text{op-mset-pick-dummy-param} \)

by simp

end

theory IICF-List

imports 
..../Sepref
  List-Index.List-Index

begin

**lemma** \( \text{param-index[param]:} \)

\[ [\text{single-valued } A; \text{single-valued } (A^{-1})] \implies (\text{index}, \text{index}) ∈ (A)\text{list-rel} \rightarrow A \rightarrow \text{nat-rel} \]

unfolding \( \text{index-def[abs-def]} \) \( \text{find-index-def} \)

apply \((\text{subgoal-tac (((=), (=)) ∈ A \rightarrow A \rightarrow \text{bool-rel}}))\)

apply parametricity

by (simp add: pres-eq-iff-svb)

3.5.3 Swap two elements of a list, by index

**definition** \( \text{swap } l \ i \ j \equiv l[i := lj, j := li] \)

**lemma** \( \text{swap-nth[simp]:} \)

\[ [i < \text{length } l; j < \text{length } l; k < \text{length } l] \implies \]

\( \text{swap } l \ i \ j \) \( k = \)

\( \text{if } k = i \text{ then } l j \)

\( \text{else if } k = j \text{ then } l i \)

\( \text{else } l k \)

)\n
unfolding \( \text{swap-def} \)

by auto

**lemma** \( \text{swap-set[simp]:} \)

\[ [i < \text{length } l; j < \text{length } l] \implies \text{set } (\text{swap } l \ i \ j) = \text{set } l \]

unfolding \( \text{swap-def} \)

by auto

**lemma** \( \text{swap-multiset[simp]:} \)

\[ [i < \text{length } l; j < \text{length } l] \implies \text{mset } (\text{swap } l \ i \ j) = \text{mset } l \]

unfolding \( \text{swap-def} \)

by (auto simp: mset-swap)
lemma swap-length[simp]: length (swap l i j) = length l
  unfolding swap-def
  by auto

lemma swap-same[simp]: swap l i i = l
  unfolding swap-def by auto

lemma distinct-swap[simp]:
  \[ i < \text{length } l; j < \text{length } l \implies \text{distinct } (\text{swap } l i j) = \text{distinct } l \]
  unfolding swap-def
  by auto

lemma map-swap: \[ i < \text{length } l; j < \text{length } l \]
  \implies map f (swap l i j) = swap (map f l) i j
  unfolding swap-def
  by (auto simp add: map-update)

lemma swap-param[param]: \[ i < \text{length } l; j < \text{length } l; (l',l)\in(\mathcal{A}\text{-list-rel}); (i',i)\in\text{nat-rel};
  (j',j)\in\text{nat-rel} \]
  \implies (swap l' i' j', swap l i j)\in(\mathcal{A}\text{-list-rel})
  unfolding swap-def
  by parametricity

lemma swap-param-fref: (uncurry2 swap, uncurry2 swap) \in
  \[ λ((l,i,j). i < \text{length } l \land j < \text{length } l)\}_f (\mathcal{A}\text{-list-rel} \timesr \text{nat-rel}) \timesr \text{nat-rel} \rightarrow (\mathcal{A}\text{-list-rel})
  apply rule apply clarsimp
  unfolding swap-def
  apply parametricity
  by simp-all

lemma param-list-null[param]: (List.null,List.null) \in (\mathcal{A}\text{-list-rel} \rightarrow \text{bool-rel})
  proof
  have 1: List.null = (λ[]. True | - ⇒ False)
    apply (rule ext) subgoal for l by (cases l) (auto simp: List.null-def)
  done
  show ?thesis unfolding I by parametricity
  qed

3.5.4 Operations

sepref-decl-op list-empty: [] :: (\mathcal{A}\text{-list-rel} .
context notes [simp] = eq-nil-null begin
  sepref-decl-op list-is-empty: λl. l=[] :: (\mathcal{A}\text{-list-rel} → \text{bool-rel} .
end

sepref-decl-op list-replicate: replicate :: nat-rel → A → (\mathcal{A}\text{-list-rel} .
definition op-list-copy :: 'a list ⇒ 'a list where [simp]: op-list-copy l ≡ l

sepref-decl-op (no-def) list-copy: op-list-copy :: (\mathcal{A}\text{-list-rel} → (\mathcal{A}\text{-list-rel} .
sepref-decl-op list-prepend: (©) :: A → (A)list-rel → (A)list-rel .
sepref-decl-op list-append: λxs x. xs@x[z] :: (A)list-rel → A → (A)list-rel .
sepref-decl-op list-concat: (@) :: (A)list-rel → (A)list-rel → (A)list-rel .
sepref-decl-op list-length: length :: (A)list-rel → nat-rel .
sepref-decl-op list-get: nth :: [λ(l,i). i<length l]f (A)list-rel ×r nat-rel → A .
sepref-decl-op list-set: list-update :: [λ((l,i),j). i<length l]f ((A)list-rel ×r nat-rel) ×r A → (A)list-rel .

countext notes [simp] = eq-Nil-null begin
  sepref-decl-op list-hd: hd :: [λl. l≠[]]f (A)list-rel → A .
  sepref-decl-op list-tl: tl :: [λl. l≠[]]f (A)list-rel → (A)list-rel .
  sepref-decl-op list-last: last :: [λl. l≠[]]f (A)list-rel → A .
  sepref-decl-op list-butlast: butlast :: [λl. l≠[]]f (A)list-rel → (A)list-rel .
end
sepref-decl-op list-contains: λx l. x∈set l :: A → (A)list-rel → bool-rel
  where single-valued A single-valued (A⁻¹) .
sepref-decl-op list-swap: swap :: [λ((l,i),j). i<length l ∧ j<length l]f ((A)list-rel ×r nat-rel) ×r nat-rel → (A)list-rel .
sepref-decl-op list-rotate1: rotate1 :: (A)list-rel → (A)list-rel .
sepref-decl-op list-rotate: rev :: (A)list-rel → (A)list-rel .
sepref-decl-op list-index: index :: (A)list-rel → A → nat-rel
  where single-valued A single-valued (A⁻¹) .

3.5.5 Patterns

lemma [def-pat-rules]:
  [] ≡ op-list-empty
  (=) $x$[] ≡ op-list-is-empty$x$
  (=) $x$l ≡ op-list-is-empty$x$
  replicate$n$v ≡ op-list-replicate$n$v
  Cons$x$x ≡ op-list-prepend$x$x
  (@) $x$x(Cons$x[])$ ≡ op-list-append$x$x
  (@) $x$ys ≡ op-list-concat$x$ys
  op-list-concat$x$x(Cons$x[])$ ≡ op-list-append$x$x
  length$x$x ≡ op-list-length$x$x
  nth$I$i ≡ op-list-get$I$i
  list-update$I$i$x$ ≡ op-list-set$I$i$x$
  hd$I$ ≡ op-list-hd$I$
  hd$I$ ≡ op-list-hd$I$
  tl$I$ ≡ op-list-tl$I$
  tl$I$ ≡ op-list-tl$I$
  last$I$ ≡ op-list-last$I$
  butlast$I$ ≡ op-list-butlast$I$
  (©) $x$(set$I$) ≡ op-list-contains$x$I$
  swap$I$i ≡ op-list-swap$I$i
  rotate1$I$ ≡ op-list-rotate$I$
  rev$I$ ≡ op-list-rev$I$
  index$I$x ≡ op-list-index$I$x
  by (auto intro!: eq-reflection)

Standard preconditions are preserved by list-relation. These lemmas are
used for simplification of preconditions after composition.

\textbf{lemma} list-rel-pres-neq-nil[fcomp-prenorm-simps]: \((x', x) \in (A) \leadsto x' \neq [] \iff x \neq []\) by auto

\textbf{lemma} list-rel-pres-length[fcomp-prenorm-simps]: \((x', x) \in (A) \leadsto \text{length } x' = \text{length } x\) by (rule list-rel-imp-same-length)

\textbf{locale} list-custom-empty =
\begin{align*}
\text{fixes } & \text{rel empty and op-custom-empty :: 'a list} \\
\text{assumes customize-hnr-aux: } & (\text{uncurry0 empty, uncurry0 (RETURN (op-list-empty::'a list)))} \in \text{unit-assn}^{k} \implies \text{rel} \\
\text{assumes op-custom-empty-def: op-custom-empty = op-list-empty} \\
\text{begin } & \text{sepref-register op-custom-empty-def: 'c list} \\
\text{lemma} & \text{fold-custom-empty:} \\
& [] = \text{op-custom-empty} \\
& \text{op-list-empty = op-custom-empty} \\
& \text{mop-list-empty = RETURN op-custom-empty} \\
& \text{unfolding op-custom-empty-def by simp-all}
\end{align*}

\textbf{lemmas} custom-hnr[sepref-fr-rules] = customize-hnr-aux[folded op-custom-empty-def]

\textbf{end}

\textbf{lemma} gen-mop-list-swap: mop-list-swap l i j = do 
\begin{align*}
x_i & \leftarrow \text{mop-list-get } l \ i \\
x_j & \leftarrow \text{mop-list-get } l \ j \\
l & \leftarrow \text{mop-list-set } l \ i \ x_j \\
l & \leftarrow \text{mop-list-set } l \ j \ x_i \\
& \text{RETURN } l
\end{align*}

\text{unfolding mop-list-swap-def by (auto simp: pw-eq-iff refine-pw-simps swap-def)}

\textbf{end}

\section{3.6 Heap Implementation On Lists}

\textbf{theory} IICF-Abs-Heap
\textbf{imports}
\begin{itemize}
\item HOL\textendash Library.Multiset
\item ../../../Sepref.List-Index.List-Index
\item ../../../Intf/IICF-List
\item ../../../Intf/IICF-Prio-Bag
\end{itemize}

\textbf{begin}

We define Min-Heaps, which implement multisets of prioritized values. The
operations are: empty heap, emptiness check, insert an element, remove a minimum priority element.

3.6.1 Basic Definitions

type-synonym 'a heap = 'a list

locale heapstruct =
  fixes prio :: 'a ⇒ 'b:linorder
begin
  definition valid :: 'a heap ⇒ nat ⇒ bool
    where valid h i ≡ i>0 ∧ i≤length h

abbreviation α :: 'a heap ⇒ 'a multiset
  where α ≡ mset

lemma valid-empty[simp]: ¬valid [] i by (auto simp: valid-def)
lemma valid0[simp]: ¬valid h 0 by (auto simp: valid-def)
lemma valid-glen[simp]: i>length h ⇒ ¬valid h i by (auto simp: valid-def)
lemma valid-len[simp]: h ≠ [] ⇒ valid h (length h) by (auto simp: valid-def)
lemma validI: 0<i ⇒ i≤length h ⇒ valid h i
  by (auto simp: valid-def)

definition val-of :: 'a heap ⇒ nat ⇒ 'a
  where val-of l i ≡ l!(i−1)

abbreviation prio-of :: 'a heap ⇒ nat ⇒ 'b
  where prio-of l i ≡ prio (val-of l i)

Navigating the tree

definition parent :: nat ⇒ nat
  where parent i ≡ i div 2

definition left :: nat ⇒ nat
  where left i ≡ 2*i

definition right :: nat ⇒ nat
  where right i ≡ 2*i + 1

abbreviation has-parent h i ≡ valid h (parent i)
abbreviation has-left h i ≡ valid h (left i)
abbreviation has-right h i ≡ valid h (right i)

abbreviation vparent h i == val-of h (parent i)
abbreviation vleft h i == val-of h (left i)
abbreviation vright h i == val-of h (right i)

abbreviation pparent h i == prio-of h (parent i)
abbreviation pleft h i == prio-of h (left i)
abbreviation pright h i == prio-of h (right i)

lemma parent-left-id[simp]: parent (left i) = i
  unfolding parent-def left-def
by auto

lemma parent-right-id[simp]: parent (right i) = i
  unfolding parent-def right-def
  by auto

lemma child-of-parentD:
  has-parent l i \implies left (parent i) = i \lor right (parent i) = i
  unfolding parent-def left-def right-def valid-def
  by auto

lemma rc-imp-lc: [ valid h i; has-right h i ] \implies has-left h i
  by (auto simp: valid-def left-def right-def)

lemma plr-corner-cases[simp]:
  assumes 0 < i
  shows i \not= parent i
  i \not= left i
  i \not= right i
  parent i \not= i
  left i \not= i
  right i \not= i
  using assms
  by (auto simp: parent-def left-def right-def)

lemma i-eq-parent-conv[simp]: i = parent i \iff i = 0
  by (auto simp: parent-def)

Heap Property

The heap property states, that every node’s priority is greater or equal to its parent’s priority

definition heap-invar :: 'a heap \Rightarrow bool
  where heap-invar l
  \equiv \forall i. valid l i \rightarrow has-parent l i \rightarrow pparent l i \leq prio-of l i

definition heap-rel1 \equiv br \alpha heap-invar

lemma heap-invar-empty[simp]: heap-invar []
  by (auto simp: heap-invar-def)

function heap-induction-scheme :: nat \Rightarrow unit
  where heap-induction-scheme i = ( 
    if i > 1 then heap-induction-scheme (parent i) else ()
  )
  by pat-completeness auto

termination
apply (relation less-than)
apply (auto simp: parent-def)
done

lemma
heap-parent-le: \[\text{heap-invar } l \text{; valid } l \text{; has-parent } l \text{ i} \]
\[\implies \text{parent } l \text{ i } \leq \text{prio-of } l \text{ i} \]
unfolding heap-invar-def
by auto

lemma heap-min-prop:
assumes H: heap-invar h
assumes V: valid h i
shows prio-of h (Suc 0) \leq prio-of h i
proof (cases i > 1)
case False with V show ?thesis
  by (auto simp: valid-def intro: Suc-lessI)
next
case True
from V have i \leq length h valid h (Suc 0) by (auto simp: valid-def)
with True show ?thesis
  apply (induction i rule: heap-induction-scheme.induct)
  apply (rename-tac i)
  apply (case-tac parent i = Suc 0)
  apply (rule order-trans[rotated])
  apply (rule heap-parent-le[OF H])
  apply (auto simp: valid-def parent-def)
done
qed

Obviously, the heap property can also be stated in terms of children, i.e.,
each node’s priority is smaller or equal to it’s children’s priority.

definition children-ge h p i \equiv
  (has-left h i \rightarrow p \leq pleft h i)
\land (has-right h i \rightarrow p \leq pright h i)

definition heap-invar' h \equiv \forall i. \text{valid } h \text{ i } \rightarrow \text{children-ge } h \text{ (prio-of } h \text{ i) } i

lemma heap-eq-heap':
shows heap-invar h \longleftrightarrow heap-invar' h
unfolding heap-invar-def
unfolding heap-invar'-def children-ge-def
apply rule

250
apply auto []
apply clarsimp
apply (frule child-of-parentD)
apply auto []
done

3.6.2 Basic Operations

The basic operations are the only operations that directly modify the underlying data structure.

Val-Of

abbreviation (input) val-of-pre l i ≡ valid l i
definition val-of-op :: 'a heap ⇒ nat ⇒ 'a nres
  where val-of-op l i ≡ ASSERT (i>0) ⇒ mop-list-get l (i-1)
lemma val-of-correct[refine-vcg]:
  val-of-pre l i ⇒ val-of-op l i ≤ SPEC (λr. r = val-of l i)
unfolding val-of-op-def val-of-def valid-def
by refine-vcg auto

abbreviation (input) prio-of-pre ≡ val-of-pre
definition prio-of-op l i ≡ do {v ← val-of-op l i; RETURN (prio v)}
lemma prio-of-op-correct[refine-vcg]:
  prio-of-pre l i ⇒ prio-of-op l i ≤ SPEC (λr. r = prio-of l i)
unfolding prio-of-op-def
apply refine-vcg by simp

Update

abbreviation update-pre h i v ≡ valid h i
definition update :: 'a heap ⇒ nat ⇒ 'a ⇒ 'a heap
  where update h i v ≡ h[i-1 := v]
definition update-op :: 'a heap ⇒ nat ⇒ 'a ⇒ 'a heap nres
  where update-op h i v ≡ ASSERT (i>0) ⇒ mop-list-set h (i-1) v
lemma update-correct[refine-vcg]:
  update-pre h i v ⇒ update-op h i v ≤ SPEC(λr. r = update h i v)
unfolding update-op-def update-def valid-def by refine-vcg auto

lemma update-valid[simp]: valid (update h i v) j ←→ valid h j
by (auto simp: update-def valid-def)

lemma val-of-update[simp]: [update-pre h i v; valid h j] ⇒ val-of (update h i v) j = (if i=j then v else val-of h j)
unfolding update-def val-of-def
by (auto simp: nth-list-update valid-def)

lemma length-update[simp]: length (update l i v) = length l

251
Exchange

Exchange two elements

definition exch :: 'a heap ⇒ nat ⇒ nat ⇒ 'a heap where
  exch l i j ≡ swap l (i - 1) (j - 1)
abbreviation exch-pre l i j ≡ valid l i ∧ valid l j

definition exch-op :: 'a list ⇒ nat ⇒ nat ⇒ 'a list nres
where  exch-op l i j ≡ do
  ASSERT (i>0 ∧ j>0);
  l ← mop-list-swap l (i - 1) (j - 1);
  RETURN l

lemma exch-op-alt: exch-op l i j = do
  vi ← val-of-op l i;
  vj ← val-of-op l j;
  l ← update-op l i vj;
  l ← update-op l j vi;
  RETURN l
by (auto simp: exch-op-def swap-def val-of-op-def update-op-def
    pw-eq-iff refine-pw-simps)

lemma exch-op-correct[refine-vcg]:
  exch-pre l i j ⇒ exch-op l i j ≤ SPEC (λr. r = exch l i j)
unfolding exch-op-def
apply refine-vcg
apply (auto simp: exch-def valid-def)
done

lemma valid-exch[simp]: valid (exch l i j) k = valid l k
unfolding exch-def by (auto simp: valid-def)

lemma val-of-exch[simp]: [valid l i; valid l j; valid l k] ⇒
  val-of (exch l i j) k = ( if k=i then val-of l j
                             else if k=j then val-of l i
                             else val-of l k )
unfolding exch-def val-of-def valid-def
by (auto)

lemma exch-eq[simp]: exch h i i = h
by (auto simp: exch-def)

lemma α-exch[simp]: [valid l i; valid l j]
  ⇒ α (exch l i j) = α l
unfolding exch-def valid-def
by (auto)

lemma length-exch[simp]: length (exch l i j) = length l
by (auto simp: exch-def)

Butlast
Remove last element

abbreviation butlast-pre l ≡ l ≠ []
definition butlast-op :: 'a heap ⇒ 'a heap nres
where butlast-op l ≡ mop-list-butlast l
lemma butlast-op-correct[refine-vcg]:
  butlast-pre l ⇒ butlast-op l ≤ SPEC (λr. r = butlast l)
unfolding butlast-op-def by (refine-vcg; auto)

lemma valid-butlast-conv[simp]: valid (butlast h) i ←→ valid h i ∧ i < length h
by (auto simp: valid-def)

lemma valid-butlast: valid (butlast h) i ⇒ valid h i
by (cases h rule: rev-cases) (auto simp: valid-def)

lemma val-of-butlast'[simp]: [ valid h i; i < length h ]
⇒ val-of (butlast h) i = val-of h i
by (auto simp: valid-def val-of-def nth-append)

lemma α-butlast[simp]: [ length h ≠ 0 ]
⇒ α (butlast h) = α h - {# val-of h (length h)#}
apply (cases h rule: rev-cases)
apply (auto simp: val-of-def)
done

lemma heap-invar-butlast[simp]: heap-invar h ⇒ heap-invar (butlast h)
apply (cases h = [])
apply simp
apply (auto simp: heap-invar-def dest: valid-butlast)
done

Append

definition append-op :: 'a heap ⇒ 'a ⇒ 'a heap nres
where append-op l v ≡ mop-list-append l v
lemma append-op-correct[refine-vcg]:
  append-op l v ≤ SPEC (λr. r = [v])
unfolding `append-op-def` by (refine-vcg; auto)

**Lemma** `valid-append[simp]`: `valid (l@[v]) i ↔ valid l i ∨ i = length l + 1`
by (auto simp: `valid-def`)

**Lemma** `val-of-append[simp]`: `valid (l@[v]) i ⟷ val-of (l@[v]) i = (if valid l i then val-of l i else v)`
unfolding `valid-def` `val-of-def` by (auto simp: `nth-append`)

**Lemma** `α-append[simp]`: `α (l@[v]) = α l + {#v#}`
by (auto simp:)

### 3.6.3 Auxiliary operations

The auxiliary operations do not have a corresponding abstract operation, but are to restore the heap property after modification.

**Swim**

This invariant expresses that the heap has a single defect, which can be repaired by swimming up

**Definition** `swim-invar` :: `'a heap ⇒ nat ⇒ bool`
where `swim-invar h i ≡`
  `valid h i ∧ (∀ j. valid h j ∧ has-parent h j ∧ j ≠ i → pparent h j ≤ prio-of h j)
  ∧ (has-parent h i → (∀ j. valid h j ∧ has-parent h j ∧ parent j = i
  → pparent h i ≤ prio-of h j))`

Move up an element that is too small, until it fits

**Definition** `swim-op` :: `'a heap ⇒ nat ⇒ 'a heap nres` where
`swim-op h i ≡ do`
  `RECT (λswim (h,i). do {
    ASSERT (valid h i ∧ swim-invar h i);
    if has-parent h i then do {
      ppi ← prio-of-op h (parent i);
      pi ← prio-of-op h i;
      if (¬ppi ≤ pi) then do {
        h ← exch-op h i (parent i);
        swim (h, parent i)
      } else
        RETURN h
    } else
    RETURN h
  }) (h,i)
lemma swim-invar-valid: \( \text{swim-invar} \ h \ i \implies \text{valid} \ h \ i \)
unfolding \( \text{swim-invar-def} \) by simp

lemma swim-invar-exit1: \( \neg \text{has-parent} \ h \ i \implies \text{swim-invar} \ h \ i \implies \text{heap-invar} \ h \)
unfolding heap-invar-def \( \text{swim-invar-def} \) by auto

lemma swim-invar-exit2: \( \text{pparent} \ h \ i \leq \text{prio-of} \ h \ i \implies \text{swim-invar} \ h \ i \implies \text{heap-invar} \ h \)
unfolding heap-invar-def \( \text{swim-invar-def} \) by auto

lemma swim-invar-pres:
assumes HPI: \( \text{has-parent} \ h \ i \)
assumes VIOLATED: \( \text{pparent} \ h \ i \ > \text{prio-of} \ h \ i \)
and INV: \( \text{swim-invar} \ h \ i \)
defines \( h' \equiv \text{exch} \ h \ i \ (\text{parent} \ i) \)
shows \( \text{swim-invar} \ h' \ (\text{parent} \ i) \)
unfolding \( \text{swim-invar-def} \)
apply safe
apply (simp add: \( h'\)-def HPI)

using HPI VIOLATED INV
unfolding \( \text{swim-invar-def} \ h'\)-def
apply auto []

using HPI VIOLATED INV
unfolding \( \text{swim-invar-def} \ h'\)-def
apply auto
by (metis order-trans)

lemma swim-invar-decr:
assumes INV: \( \text{heap-invar} \ h \)
assumes V: \( \text{valid} \ h \ i \)
assumes DECR: \( \text{prio} \ v \leq \text{prio-of} \ h \ i \)
shows \( \text{swim-invar} \ (\text{update} \ h \ i \ v) \ i \)
using INV V DECR
apply (auto simp: \( \text{swim-invar-def} \) heap-invar-def intro: dual-order.trans)
done

lemma swim-op-correct[refine-vcg]:
\([\text{swim-invar} \ h \ i] \implies \text{swim-op} \ h \ i \leq \text{SPEC} \ (\lambda h'. \alpha \ h' = \alpha \ h \land \text{heap-invar} \ h' \land \text{length} \ h' = \text{length} \ h)\)
unfolding \( \text{swim-op-def} \)
using [[goals-limit = 1]]
apply (refine-vcg RECT-rule[where
  pre=\( \lambda (hh,i). \text{swim-invar} \ hh \ i \land \alpha \ hh = \alpha \ h \)\]

255
\[\text{\texttt{Sink}}\]

Move down an element that is too big, until it fits in

\textbf{definition} sink-op :: 'a heap ⇒ nat ⇒ 'a heap nres where

\begin{verbatim}
  sink-op h i ≡ do { RECT (λsink (h,i). do {
    ASSERT (valid h i);
    if has-right h i then do {
      ASSERT (has-left h i);
      lp ← prio-of-op h (left i);
      rp ← prio-of-op h (right i);
      p ← prio-of-op h i;
      if (lp < p ∧ rp ≥ lp) then do {
        h ← exch-op h i (left i);
        sink (h,left i)
      } else if (rp<lp ∧ rp < p) then do {
        h ← exch-op h i (right i);
        sink (h,right i)
      } else
        RETURN h
    } else if (has-left h i) then do {
      lp ← prio-of-op h (left i);
      p ← prio-of-op h i;
      if (lp < p) then do {
        h ← exch-op h i (left i);
      }
    }
  }
}
\end{verbatim}
\[
\begin{align*}
sink (h, \text{left } i) \\
\} \text{ else } \\
\quad \text{RETURN } h
\end{align*}
\]
\[
\begin{align*}
\} \text{ else } \\
\quad \text{RETURN } h \\
\} (h, i)
\end{align*}
\]

This invariant expresses that the heap has a single defect, which can be repaired by sinking

**definition** sink-invar \ l i \equiv

\[
\begin{align*}
\text{valid } l i \\
\land (\forall j. \text{valid } l j \land j \neq i \rightarrow \text{children-ge } l (\text{prio-of } l j) j) \\
\land (\text{has-parent } l i \rightarrow \text{children-ge } l (\text{parent } l i) i)
\end{align*}
\]

**lemma** sink-invar-valid: sink-invar \ l i \implies \text{valid } l i

unfolding sink-invar-def by auto

**lemma** sink-invar-exit: \[[\text{sink-invar } l i; \text{children-ge } l (\text{prio-of } l i) i]\]\n
\[
\implies \text{heap-invar'} \ l
\]

unfolding heap-invar'-def sink-invar-def by auto

**lemma** sink-aux1: \((2i \leq \text{length } h) \implies \neg\text{has-left } h i \land \neg\text{has-right } h i

unfolding valid-def left-def right-def by auto

**lemma** sink-invar-pres1:

assumes sink-invar \ h i

assumes has-left \ h i has-right \ h i

assumes prio-of \ h i \geq \text{pleft } h i

assumes pleft \ h i \geq \text{pright } h i

shows sink-invar (exch \ h i (right \ i)) (right \ i)

using assms

unfolding sink-invar-def

apply auto

apply (auto simp: children-ge-def)

done

**lemma** sink-invar-pres2:

assumes sink-invar \ h i

assumes has-left \ h i has-right \ h i

assumes prio-of \ h i \geq \text{pleft } h i

assumes pleft \ h i \leq \text{pright } h i

shows sink-invar (exch \ h i (left \ i)) (left \ i)

using assms

unfolding sink-invar-def

apply auto

apply (auto simp: children-ge-def)
lemma sink-invar-pres3:
assumes sink-invar h i
assumes has-left h i has-right h i
assumes prio-of h i \geq pright h i
assumes pleft h i \leq pright h i
shows sink-invar (exch h i (left i)) (left i)
using assms
unfolding sink-invar-def
apply auto
apply \((auto\ simp:\ children-ge-def)\)
done

lemma sink-invar-pres4:
assumes sink-invar h i
assumes has-left h i has-right h i
assumes prio-of h i \geq pright h i
assumes pleft h i \geq pright h i
shows sink-invar (exch h i (right i)) (right i)
using assms
unfolding sink-invar-def
apply auto
apply \((auto\ simp:\ children-ge-def)\)
done

lemma sink-invar-pres5:
assumes sink-invar h i
assumes has-left h i \neg has-right h i
assumes prio-of h i \geq pleft h i
shows sink-invar (exch h i (left i)) (left i)
using assms
unfolding sink-invar-def
apply auto
apply \((auto\ simp:\ children-ge-def)\)
done

lemmas sink-invar-pres =
sink-invar-pres1
sink-invar-pres2
sink-invar-pres3
sink-invar-pres4
sink-invar-pres5

lemma sink-invar-incr:
assumes INV: heap-invar h
assumes V: valid h i
assumes INCR: prio v \geq prio-of h i
shows sink-invar (update h i v) i
using INV V INCR
apply (auto simp: sink-invar-def)
apply (auto simp: children-ge-def heap-invar-def intro: order-trans)
apply (frule spec[where x=left i])
apply auto
apply (frule spec[where x=right i])
apply auto
done

lemma sink-op-correct[refine-vcg]:
[sink-invar h i] ==> sink-op h i \leq SPEC (\alpha h' = \alpha h \land heap-invar h' \land length h' = length h)
unfolding sink-op-def heap-eq-heap'
using [goals-limit = 1]
apply (refine-vcg RECT-rule[where
pre=\lambda(hh,i). sink-invar hh i \land \alpha hh = \alpha h \land length hh = length h and
V = measure (\lambda(l,i). length l - i)
])
apply (auto)
apply (auto)
apply (auto)
apply (auto)
apply (auto simp: sink-invar-valid)
apply (auto simp: valid-def left-def right-def)
apply rprems
apply (auto intro: sink-invar-pres)
apply (auto simp: valid-def left-def right-def)
apply rprems
apply (auto intro: sink-invar-pres)
apply (auto simp: valid-def left-def right-def)
apply (auto)
apply clarsimp
apply (rule sink-invar-exit, assumption)
apply (auto simp: children-ge-def)
apply (auto)
apply rprems
apply (auto intro: sink-invar-pres)
lemma sink-op-swim-rule:
swim-invar h i ⟷ sink-op h i ≤ SPEC (λh'. h'=h)
apply (frule swim-invar-valid)
unfolding sink-op-def
apply (subst RECT-unfold, refine-mono)
apply (fold sink-op-def)
apply refine-vcg
apply (simp-all)
apply (auto simp add: valid-def left-def right-def dest: swim-invar-valid)
apply (auto simp: swim-invar-def)
apply (auto simp: swim-invar-def)
apply (auto simp: swim-invar-def)
apply (auto simp: swim-invar-def)
done

definition sink-op-opt
— Sink operation as presented in Sedgewick et al. Algs4 reference implementation

where
sink-op-opt h k ≡ RECT (λD (h,k). do {
  ASSERT (k>0 ∧ k≤length h);
  let len = length h;
  if (2*k ≤ len) then do {
    let j = 2*k;
    pj ← prio-of-op h j;
    j ← (pj < len then do {
        psj ← prio-of-op h (Suc j);
        j ← (j < len then do {
            pj ← prio-of-op h (Suc j);
if \( pj > psj \) then RETURN \((j+1)\) else RETURN \(j\);

\(pj \leftarrow \text{prio-of-op } h \ j\);
\(pk \leftarrow \text{prio-of-op } h \ k\);
if \((pk > pj)\) then do 
\{ 
\(h \leftarrow \text{exch-op } h \ k \ j\); 
\(D \ (h,j)\) 
\} else
\(\text{RETURN } h\) 
\} else RETURN \(h\)

\}

\text{lemma } \text{sink-op-opt-eq}: \text{sink-op-opt } h \ k = \text{sink-op } h \ k
\\text{unfolding } \text{sink-op-opt-def } \text{sink-op-def}
\text{apply } (\text{fo-rule } \text{arg-cong } \text{fun-cong})+
\text{apply } (\text{intro } \text{ext})
\text{unfolding } \text{sink-op-def[ symmetric]}
\text{apply } (\text{simp } \text{cong: if-cong split del: if-split add: Let-def})
\text{apply } (\text{auto simp: valid-def left-def right-def prio-of-op-def val-of-op-def}
\text{ val-of-def less-imp-diff-less } \text{ASSERT-same-eq-cone nz-le-conv-less})
\text{done}

\text{Repair}

Repair a local defect in the heap. This can be done by swimming and sinking. Note that, depending on the defect, only one of the operations will change the heap. Moreover, note that we do not need repair to implement the heap operations. However, it is required for heapmaps.

\text{definition } \text{repair-op } h \ i \equiv \text{do } 
\{ 
\(h \leftarrow \text{sink-op } h \ i\);
\(h \leftarrow \text{swim-op } h \ i\);
\(\text{RETURN } h\)
\} 

\text{lemma } \text{update-sink-swim-cases}:
\text{assumes } \text{heap-invar } h
\text{assumes } \text{valid } h \ i
\text{obtains } \text{swim-invar } (\text{update } h \ i \ v) \ i \ | \ \text{sink-invar } (\text{update } h \ i \ v) \ i
\text{apply } (\text{cases rule: linear[of prio-v prio-of-h-i, THEN disjE]})
\text{apply } (\text{blast dest: swim-invar-decr[OF assms]})
\text{apply } (\text{blast dest: sink-invar-incr[OF assms]})
\text{done}

\text{lemma } \text{heap-invar-imp-swim-invar}: [\text{heap-invar } h; \text{valid } h \ i] \implies \text{swim-invar } h \ i
\text{unfolding } \text{heap-invar-def } \text{swim-invar-def}

261
by (auto intro: order-trans)

lemma repair-correct[refine-vcg]:
  assumes heap-invar h and valid h i
  shows repair-op (update h i v) i \leq SPEC (\lambda h'.
  heap-invar h' \land \alpha h' = \alpha (update h i v) \land length h' = length h)
  apply (rule update-sink-swim-cases[of h i v, OF assms])
  unfolding repair-op-def
  apply (refine-vcg sink-op-swim-rule)
  apply auto [4]
  apply (refine-vcg)
  using assms (2)
  apply (auto intro: heap-invar-imp-swim-invar simp: valid-def) []
  apply auto [3]
  done

3.6.4 Operations

Empty

abbreviation (input) empty :: \'a heap — The empty heap
  where empty \equiv []

definition empty-op :: \'a heap nres
  where empty-op \equiv mop-list-empty

lemma empty-op-correct[refine-vcg]:
  empty-op \leq SPEC (\lambda r. \alpha r = \{\#\} \land heap-invar r)
  unfolding empty-op-def
  apply refine-vcg by auto

Emptiness check

definition is-empty-op :: \'a heap \Rightarrow bool nres — Check for emptiness
  where is-empty-op h \equiv do {ASSERT (heap-invar h); let l=length h; RETURN (l=0)}

lemma is-empty-op-correct[refine-vcg]:
  heap-invar h \implies is-empty-op h \leq SPEC (\lambda r. r\leftarrow\alpha h = \{\#\})
  unfolding is-empty-op-def
  apply refine-vcg by auto

Insert

definition insert-op :: \'a \Rightarrow \'a heap \Rightarrow \'a heap nres — Insert element
  where insert-op v h \equiv do {
  ASSERT (heap-invar h);
  h \leftarrow append-op h v;
  let l = length h;
  h \leftarrow swim-op h l;
  RETURN h
}

262
lemma swim-invar-insert: heap-invar \ l \Longrightarrow \ swim-invar ((\ell@[x]) (\text{Suc} (\text{length} \ l)))
unfolding swim-invar-def heap-invar-def valid-def parent-def val-of-def
by (fastforce simp: nth-append)

lemma (insert-op,\ RETURN oo \ op-mset-insert) \in \ Id \rightarrow \ heap-rel1 \rightarrow \ (heap-rel1)nres-rel
unfolding insert-op-def [abs-def] heap-rel1-def o-def
by refine-vcg (auto simp: swim-invar-insert in-br-conv)

lemma insert-op-correct:
heap-invar \ h \Longrightarrow \ insert-op \ v \ h \leq \ SPEC (\lambda h'. \ heap-invar \ h' \land \alpha \ h' = \alpha \ h + \{\#v\})
unfolding insert-op-def
by (refine-vcg) (auto simp: swim-invar-insert)
lemmas [refine-vcg] = insert-op-correct

Pop minimum element
definition pop-min-op :: 'a heap \Rightarrow ('a \times 'a heap) nres where
pop-min-op \ h \equiv \ do \{
ASSERT (heap-invar \ h);
ASSERT (valid \ h 1);
m \leftarrow \ val-of-op \ h \ 1;
let \ l = \ length \ h;
h \leftarrow \ exch-op \ h \ 1 \ l;
h \leftarrow \ butlast-op \ h;

if \ (l\neq 1) \ then \ do \{
\ h \leftarrow \ sink-op \ h \ 1;
\ RETURN \ (m, h)
\} \ else \ RETURN \ (m, h)
\}

lemma left-not-one[simp]: left \ j \neq \ Suc \ 0
by (auto simp: left-def)

lemma right-one-conv[simp]: right \ j = \ Suc \ 0 \iff \ j=0
by (auto simp: right-def)

lemma parent-one-conv[simp]: parent (Suc \ 0) = 0
by (auto simp: parent-def)

lemma sink-invar-init:
assumes \ I: heap-invar \ h
assumes \ NE: \ length \ h > 1
shows \ sink-invar \ (butlast \ (exch \ h \ (Suc \ 0) \ (\text{length} \ h))) \ (Suc \ 0)
proof --
from \ NE \ have \ V: \ valid \ h \ (Suc \ 0) \ valid \ h \ (\text{length} \ h)
apply –
apply (auto simp: valid-def neg-Nil-conv) []
by (cases h) (auto simp: valid-def)

show ?thesis using I
unfolding heap-eq-heap' heap-invar'-def sink-invar-def
apply (intro impI conjI allI)
using NE apply (auto simp: V valid-butlast-conv) []
apply (auto simp add: children-ge-def V NE valid-butlast-conv) []
apply (auto simp add: children-ge-def V NE valid-butlast-conv) []
done
qed

lemma in-set-conv-val: v ∈ set h ←→ (∃ i. valid h i ∧ v = val-of h i)
apply (rule iffI)
apply (clarsimp simp add: valid-def val-of-def in-set-conv-nth)
apply (rule-tac x = Suc i in exI; auto)
apply (clarsimp simp add: valid-def val-of-def in-set-conv-nth)
apply (rule-tac x = i − Suc 0 in exI; auto)
done

lemma pop-min-op-correct:
assumes heap-invar h α h ≠ {} shows pop-min-op h ≤ SPEC (λ(v, h'). heap-invar h' ∧ v ∈ # h ∧ α h' = α h − {} # v #) ∧ (∀ v' ∈ set-mset (α h). prio v ≤ prio v')
proof –
note [simp del] = length-greater-0-conv
note LG = length-greater-0-conv[symmetric]

from assms show ?thesis
unfolding pop-min-op-def
apply refine-vcg
apply (simp-all add: sink-invar-init LG)
apply (auto simp: valid-def) []
apply (cases h; auto simp: val-of-def) []
apply (auto simp: in-set-conv-val simp: heap-min-prop) []
apply auto []
apply (cases h; auto simp: val-of-def) []
apply auto []
apply (cases h; auto simp: val-of-def) []
done
qed

lemmas [refine-vcg] = pop-min-op-correct
Peek minimum element

definition peek-min-op :: 'a heap ⇒ 'a nres where
peek-min-op h ≡ do 
  ASSERT (heap-invar h);
  ASSERT (valid h 1);
  val-of-op h 1

lemma peek-min-op-correct:
  assumes heap-invar h α h ≠ {#}
  shows peek-min-op h ≤ SPEC (λv.
  v ∈# α h ∧ (∀v′∈set-mset (α h). prio v ≤ prio v′))

unfolding peek-min-op-def
apply refine-vcg
using assms
apply clarsimp-all
apply (auto simp: valid-def)
apply (auto simp: in-set-conv-val simp: heap-min-prop)
done

lemmas peek-min-op-correct[refine-vcg] = peek-min-op-correct

3.6.5 Operations as Relator-Style Refinement

lemma empty-op-refine: (empty-op,RETURN op-mset-empty)∈⟨heap-rel1⟩nres-rel
apply (rule nres-relI)
apply (rule order-trans)
apply (rule empty-op-correct)
apply (auto simp: heap-rel1-def br-def pw-le-iff refine-pw-simps)
done

lemma is-empty-op-refine: (is-empty-op,RETURN o op-mset-is-empty)∈heap-rel1
→ ⟨bool-rel⟩nres-rel
apply (intro nres-relI fun-relI; simp)
apply refine-vcg
apply (auto simp: heap-rel1-def br-def)
done

lemma insert-op-refine: (insert-op,RETURN oo op-mset-insert)∈Id → heap-rel1
→ ⟨heap-rel1⟩nres-rel
apply (intro nres-relI fun-relI; simp)
apply (refine-vcg RETURN-as-SPEC-refine)
apply (auto simp: heap-rel1-def br-def pw-le-iff refine-pw-simps)
done

lemma pop-min-op-refine:
  (pop-min-op, PR-CONST (mop-prio-pop-min prio))∈heap-rel1 → ⟨Id ×r heap-rel1⟩nres-rel
apply \ (\text{intro \ fun-relI \ nres-relI})
\textbf{unfolding} mop-prio-pop-min-def \ PR-CONST-def
apply \ (\text{refine-vcg \ SPEC-refine})
apply \ (\text{auto \ simp: heap-rel1-def \ br-def})
done

\textbf{lemma \ peek-min-op-refine:}
(peek-min-op, PR-CONST (mop-prio-peek-min \ prio)) \in \ heap-rel1 \Rightarrow \langle Id \rangle nres-rel
apply \ (\text{intro \ fun-relI \ nres-relI})
\textbf{unfolding} mop-prio-peek-min-def \ PR-CONST-def
apply \ (\text{refine-vcg \ RES-refine})
apply \ (\text{auto \ simp: heap-rel1-def \ br-def})
done

end
end
theory IICF-HOL-List
imports ../Intf/IICF-List
begin

c\textbf{ontext \ begin}

\textbf{private \ lemma \ id-take-nth-drop-rl:}
\textbf{assumes} \ i < \text{length \ l}
\textbf{assumes} \ \forall \ l1 \ x \ l2. \ [l1@x#l2; i = \text{length \ l1}] \Rightarrow \ P \ (l1@x#l2)
\textbf{shows} \ P \ l
apply \ (\text{rule \ assms} (2))
apply \ (\text{subst \ id-take-nth-drop} [\text{OF \ assms} (1)])
apply \ (\text{simp \ add: \ assms} (1))
done

\textbf{private \ lemma \ list-set-entails-aux:}
\textbf{shows} \ list-assn \ A \ l \ li * A \ x \ xi = \Rightarrow \ A \ list-assn \ A \ (l[i := x]) \ (li[i := xi]) * \text{true}
apply \ (\text{rule \ entails-preI})
apply \ (\text{clarsimp})
apply \ (\text{cases \ i < \text{length \ l}; \ cases \ i < \text{length \ li}; \ (sep-auto \ dest!: \ list-assn-aux-eqlen-simp; \ fail)?})
apply \ (\text{erule \ id-take-nth-drop-rl})
apply \ (\text{erule \ id-take-nth-drop-rl})
apply \ (\text{sep-auto \ simp \ add: \ list-update-append})
done

\textbf{private \ lemma \ list-set-hd-tl-aux:}
a \neq \ [] \Rightarrow \ list-assn \ R \ a \ c \Rightarrow \ A \ R \ (hd \ a) \ (hd \ c) * \text{true}

266
\[ a \neq [] \implies \text{list-assn} \ R \ a \ c \implies_A \text{list-assn} \ R \ (\text{tl} \ a) \ (\text{tl} \ c) \ast \text{true} \]

by (cases c; cases a; sep-auto; fail)+

**private lemma** list-set-last-butlast-aux:
\[ a \neq [] \implies \text{list-assn} \ R \ a \ c \implies_A \text{list-assn} \ R \ (\text{last} \ a) \ (\text{last} \ c) \ast \text{true} \]
\[ a \neq [] \implies \text{list-assn} \ R \ a \ c \implies_A \text{list-assn} \ R \ (\text{butlast} \ a) \ (\text{butlast} \ c) \ast \text{true} \]

by (cases c rule: rev-cases; cases a rule: rev-cases; sep-auto; fail)+

**private lemma** swap-decomp-simp[simp]:
\[ \text{swap} \ ((l @ x \# c21 \# @ xa \# l2a) \ (\text{Suc} \ (\text{length} \ l1 + \text{length} \ c21'))) = l1 @ xa \# c21 \# @ x \# l2a \]
\[ \text{swap} \ ((l @ x \# c21 \# @ xa \# l2a) \ (\text{Suc} \ (\text{length} \ l1 + \text{length} \ c21'))) \ (\text{length} \ l1) = l1 @ xa \# c21 \# @ x \# l2a \]

by (auto simp: swap-def list-update-append nth-append)

**private lemma** list-swap-aux: \([i < \text{length} \ l; j < \text{length} \ l] \implies \text{list-assn} \ A \ l \ li \implies_A \text{list-assn} \ A \ (\text{swap} \ l \ i \ j) \ (\text{swap} \ li \ i \ j) \ast \text{true} \]

apply (subst list-assn-aux-len; clarsimp)
apply (cases i=j; (sep-auto; fail)?)
apply (rule id-take-nth-drop-rl[where \ l=l \ and \ i=i]; simp)
apply (rule id-take-nth-drop-rl[where \ l=l \ and \ i=j]; simp)
apply (erule list-match-lel-lel; simp)
apply (split-list-according li l; sep-auto)
apply (split-list-according li l; sep-auto)
done

**private lemma** list-rotate1-aux: \(\text{list-assn} \ A \ a \ c \implies_A \text{list-assn} \ A \ (\text{rotate1} \ a) \ (\text{rotate1} \ c) \ast \text{true} \]

by (cases a; cases c; sep-auto)

**private lemma** list-rev-aux: \(\text{list-assn} \ A \ a \ c \implies_A \text{list-assn} \ A \ (\text{rev} \ a) \ (\text{rev} \ c) \ast \text{true} \]

apply (subst list-assn-aux-len; clarsimp)
apply (induction rule: list-induct2)
apply sep-auto
apply sep-auto
apply (erule ent-frame-fwd, frame-inference)
apply sep-auto
done

lemma mod-starE:
assumes \ h |= A \ast B
obtains \ h1 \ h2 \ where \ h1|=A \ h2|=B
using assms by (auto simp: mod-star-conv)

**private lemma** CONSTRAINT-is-pureE:
assumes \ CONSTRAINT \is-pure \ A
obtains \ R \ where \ A=pure \ R
using assms by (auto simp: is-pure-conv)

267
private method solve-dbg =
  ( (elim CONSTRAINT-is-pureE; (simp only: list-assn-pure-conv the-pure-pure))
    simp: pure-def hn-ctxt-def invalid-assn-def list-assn-aux-eqlen-simp
    intro!: hn-refineI [THEN hn-refine-preI] hpreI
    elim!: mod-starE
    intro: list-set-entails-aux list-set-hd-tl-aux list-set-last-buttlast-aux
    list-swap-aux list-rotate1-aux list-rev-aux
  ;
  ((rule entails-preI; sep-auto simp: list-assn-aux-eqlen-simp | (parametricity;
    simp: fail))))

private method solve = solve-dbg; fail

lemma HOL-list-empty-hnr-aux: (uncurry op-list-empty) (RETURN op-list-empty) ∈ unit-assn A → (list-assn A) by solve
lemma HOL-list-is-empty-hnr[sepref-fr-rules]: (return op-list-is-empty, RETURN op-list-is-empty) ∈ (list-assn A)k → boolean-assn by solve
lemma HOL-list-prepend-hnr[sepref-fr-rules]: (uncurry (return op-list-prepend), uncurry (RETURN op-list-prepend)) ∈ A* (list-assn A) → (list-assn A) by solve
lemma HOL-list-append-hnr[sepref-fr-rules]: (uncurry (return op-list-append), uncurry (RETURN op-list-append)) ∈ (list-assn A) → (list-assn A) by solve
lemma HOL-list-concat-hnr[sepref-fr-rules]: (uncurry (return op-list-concat), uncurry (RETURN op-list-concat)) ∈ (list-assn A) → (list-assn A) by solve
lemma HOL-list-length-hnr[sepref-fr-rules]: (return op-list-length, RETURN op-list-length) ∈ (list-assn A)k → nat-assn by solve
lemma HOL-list-set-hnr[sepref-fr-rules]: (uncurry op-list-set, uncurry2 (RETURN op-list-set)) ∈ (list-assn A) → (list-assn A) by solve
lemma HOL-list-hd-hnr[sepref-fr-rules]: (return op-list-hd, RETURN op-list-hd) ∈ [λy. y ≠ []] (list-assn R) → R by solve
lemma HOL-list-tl-hnr[sepref-fr-rules]: (return op-list-tl, RETURN op-list-tl) ∈ [λy. y ≠ []] (list-assn A) → list-assn A by solve
lemma HOL-list-last-hnr[sepref-fr-rules]: (return op-list-last, RETURN op-list-last) ∈ [λy. y ≠ []] (list-assn A) → R by solve
lemma HOL-list-buttlast-hnr[sepref-fr-rules]: (return op-list-buttlast, RETURN op-list-buttlast) ∈ [λy. y ≠ []] (list-assn A) → list-assn A by solve
lemma HOL-list-swap-hnr[sepref-fr-rules]: (uncurry2 (return op-list-swap), uncurry2 (RETURN op-list-swap)) ∈ [λ(a, b), ba < length a ∧ ba < length a] (list-assn A) → (list-assn A) by solve
lemma HOL-list-rotate1-hnr[sepref-fr-rules]: (return op-list-rotate1, RETURN op-list-rotate1) ∈ (list-assn A) → list-assn A by solve
lemma \( \text{HOL-list-replicate-hnr} \subseteq (\text{list-assn} A)^d \rightarrow_a \text{list-assn} A \) by solve

lemma \( \text{HOL-list-rev-hnr} \subseteq (\text{list-assn} A)^d \rightarrow_a \text{list-assn} A \) by solve

lemma \( \text{HOL-list-get-hnr} \subseteq (\text{list-assn} A)^d \rightarrow_a \text{list-assn} A \) by solve

private lemma \( \text{mk-mop-rl1-np} \).

private lemma \( \text{mk-mop-rl2-np} \).

private lemma \( \text{mk-mop-rl1-np} \).

private lemma \( \text{mk-mop-rl2-np} \).

lemma \( \text{HOL-list-contains-hnr} \subseteq (\text{list-assn} A)^d \rightarrow_a \text{bool-assn} \).

apply solve-dbq
apply (erule bool-by-paramE[where \( a = \in \) set]) apply parametricity
apply (erule bool-by-paramE'[where \( a = \in \) set]) apply parametricity

lemmas \( \text{HOL-list-empty-hnr-mop} = \text{HOL-list-empty-hnr-aux}[\text{FCOMP} \text{mk-mop-rl0-np}[\text{OF}\ \text{map-list-empty-alt}]] \).

lemmas \( \text{HOL-list-is-empty-hnr-mop} = \text{HOL-list-is-empty-hnr}[\text{FCOMP} \text{mk-mop-rl1-np}[\text{OF}\ \text{map-list-is-empty-alt}]] \).

lemmas \( \text{HOL-list-prepend-hnr-mop} = \text{HOL-list-prepend-hnr}[\text{FCOMP} \text{mk-mop-rl2-np}[\text{OF}\ \text{map-list-prepend-alt}]] \).

lemmas \( \text{HOL-list-append-hnr-mop} = \text{HOL-list-append-hnr}[\text{FCOMP} \text{mk-mop-rl2-np}[\text{OF}\ \text{map-list-append-alt}]] \).

lemmas \( \text{HOL-list-concat-hnr-mop} = \text{HOL-list-concat-hnr}[\text{FCOMP} \text{mk-mop-rl2-np}[\text{OF}\ \text{map-list-concat-alt}]] \).

lemmas \( \text{HOL-list-length-hnr-mop} = \text{HOL-list-length-hnr}[\text{FCOMP} \text{mk-mop-rl1-np}[\text{OF}\ \text{map-list-length-alt}]] \).

lemmas \( \text{HOL-list-set-hnr-mop} = \text{HOL-list-set-hnr}[\text{FCOMP} \text{mk-mop-rl3}[\text{OF}\ \text{map-list-set-alt}]] \).

lemmas \( \text{HOL-list-hd-hnr-mop} = \text{HOL-list-hd-hnr}[\text{FCOMP} \text{mk-mop-rl1}[\text{OF}\ \text{map-list-hd-alt}]] \).

lemmas \( \text{HOL-list-tl-hnr-mop} = \text{HOL-list-tl-hnr}[\text{FCOMP} \text{mk-mop-rl1}[\text{OF}\ \text{map-list-tl-alt}]] \).

lemmas \( \text{HOL-list-last-hnr-mop} = \text{HOL-list-last-hnr}[\text{FCOMP} \text{mk-mop-rl1}[\text{OF}\ \text{map-list-last-alt}]] \).

lemmas \( \text{HOL-list-butlast-hnr-mop} = \text{HOL-list-butlast-hnr}[\text{FCOMP} \text{mk-mop-rl1}[\text{OF}\ \text{map-list-butlast-alt}]] \).

lemmas \( \text{HOL-list-swap-hnr-mop} = \text{HOL-list-swap-hnr}[\text{FCOMP} \text{mk-mop-rl3}[\text{OF}\ \text{map-list-swap-alt}]] \).

lemmas \( \text{HOL-list-rotate1-hnr-mop} = \text{HOL-list-rotate1-hnr}[\text{FCOMP} \text{mk-mop-rl3}[\text{OF}\ \text{map-list-rotate1-alt}]] \).

269
lemmas HOL-list-rev-hnr-mop[sepref-fr-rules] = HOL-list-rev-hnr[FCOMP mk-mop-rl1-np[OF mop-list-rev-alt]]
lemmas HOL-list-replicate-hnr-mop[sepref-fr-rules] = HOL-list-replicate-hnr[FCOMP mk-mop-rl2-np[OF mop-list-replicate-alt]]
lemmas HOL-list-get-hnr-mop[sepref-fr-rules] = HOL-list-get-hnr[FCOMP mk-mop-rl2[OF mop-list-get-alt]]
lemmas HOL-list-contains-hnr-mop[sepref-fr-rules] = HOL-list-contains-hnr[FCOMP mk-mop-rl2-np[OF mop-list-contains-alt]]

lemmas HOL-list-empty-hnr = HOL-list-empty-hnr-aux HOL-list-empty-hnr-mop

end

definition [simp]: op-HOL-list-empty ≡ op-list-empty
interpretation HOL-list: list-custom-empty list-assn A return [] op-HOL-list-empty
  apply unfold-locales
  apply (sep-auto intro!: hrefI hn-refineI)
  by simp

schematic-goal
notes [sepref-fr-rules] = HOL-list-empty-hnr
shows
hn-refine (emp) (?c::'?c Heap) ?Γ' ?R (do {
  x ← RETURN [1,2,3::nat];
  let x2 = op-list-append x 5;
  ASSERT (length x = 4);
  let x = op-list-swap x 1 2;
  x ← mop-list-swap x 1 2;
  RETURN (x@x)
})
by sepref

end
theory IICF-Array-List
imports
  ../Intf/IICF-List
  Separation-Logic-Imperative-HOL.Array-Blit
begin

type-synonym 'a array-list = 'a Heap.array × nat

definition is-array-list l ≡ λ(a,n). ∃ A l'. a › a l' * ↑(n ≤ length l' ∧ l = take n l' ∧ length l'>0)

lemma is-array-list-prec[safe-constraint-rules]: precise is-array-list
  unfolding is-array-list-def[abs-def]
  apply(rule preciseI)

270
apply(simp split: prod.splits)
using preciseD snga-prec by fastforce

definition initial-capacity ≡ 16::nat
definition minimum-capacity ≡ 16::nat

definition arl-empty ≡ do {
  a ← Array.new initial-capacity default;
  return (a,0)
}

definition arl-empty-sz init-cap ≡ do {
  a ← Array.new (max init-cap minimum-capacity) default;
  return (a,0)
}

definition arl-append ≡ \(\lambda\)(a,n) x. do {
  len ← Array.len a;
  if n < len then do {
    a ← Array.upd n x a;
    return (a,n+1)
  } else do { 
    let newcap = 2 * len;
    a ← array-grow a newcap default;
    a ← Array.upd n x a;
    return (a,n+1)
  }
}

definition arl-copy ≡ \(\lambda\)(a,n). do {
  a ← array-copy a;
  return (a,n)
}

definition arl-length :: 'a::heap array-list ⇒ nat Heap where
  arl-length ≡ \(\lambda\)(a,n). return (n)

definition arl-is-empty :: 'a::heap array-list ⇒ bool Heap where
  arl-is-empty ≡ \(\lambda\)(a,n). return (n=0)

definition arl-last :: 'a::heap array-list ⇒ 'a Heap where
  arl-last ≡ \(\lambda\)(a,n). do {
    Array.nth a (n - 1)
  }

definition arl-butlast :: 'a::heap array-list ⇒ 'a array-list Heap where
  arl-butlast ≡ \(\lambda\)(a,n). do {
    let n = n - 1;
len ← Array.len a;
if (n*4 < len ∧ n*2≥minimum-capacity) then do {
    a ← array-shrink a (n*2);
    return (a,n)
} else
    return (a,n)

**definition** arl-get :: 'a::heap array-list ⇒ nat ⇒ 'a Heap
arl-get ≡ λ(a,n) i. Array.nth a i

**definition** arl-set :: 'a::heap array-list ⇒ nat ⇒ 'a⇒ 'a array-list Heap
arl-set ≡ λ(a,n) x. do { a ← Array.upd i x a; return (a,n) }

**lemma** arl-empty-rule[sep-heap-rules]: < emp > arl-empty <is-array-list []>
    by (sep-auto simp: arl-empty-def is-array-list-def initial-capacity-def)

**lemma** arl-empty-sz-rule[sep-heap-rules]: < emp > arl-empty-sz N <is-array-list []>
    by (sep-auto simp: arl-empty-sz-def is-array-list-def minimum-capacity-def)

**lemma** arl-copy-rule[sep-heap-rules]: <is-array-list l a > arl-copy a <λr. is-array-list l a * is-array-list l r>
    by (sep-auto simp: arl-copy-def is-array-list-def)

**lemma** arl-append-rule[sep-heap-rules]:
    < is-array-list l a >
    arl-append a x
    <λa. is-array-list (l@[x]) a >t
    by (sep-auto
        simp: arl-append-def is-array-list-def take-update-last neq-Nil-conv
        split: prod.splits nat.split)

**lemma** arl-length-rule[sep-heap-rules]:
    <is-array-list l a >
    arl-length a
    <λr. is-array-list l a * ↑(r=length l)>t
    by (sep-auto simp: arl-length-def is-array-list-def)

**lemma** arl-is-empty-rule[sep-heap-rules]:
    <is-array-list l a >
    arl-is-empty a
    <λr. is-array-list l a * ↑(r→(l=[]))>
    by (sep-auto simp: arl-is-empty-def is-array-list-def)

**lemma** arl-last-rule[sep-heap-rules]:
    l≠[] ⇒
    <is-array-list l a >
arl-last a
<λ r. is-array-list l a * ↑(r=last l)>
by (sep-auto simp: arl-last-def is-array-list-def last-take-nth-conv)

lemma arl-butlast-rule[sep-heap-rules]:
l≠[] ⟹
<is-array-list l a>
arl-butlast a
<is-array-list (butlast l)>t
proof –
assume [simp]: l≠[]

have [simp]: ∀ x. min (x−Suc 0) ((x−Suc 0)*2) = x−Suc 0 by auto

show ?thesis
by (sep-auto
  split: prod.splits
  simp: arl-butlast-def is-array-list-def butlast-take minimum-capacity-def)
qed

lemma arl-get-rule[sep-heap-rules]:
i<length l ⟹
<is-array-list l a>
arl-get a i
<λ r. is-array-list l a * ↑(r=!i)>
by (sep-auto simp: arl-get-def is-array-list-def split: prod.split)

lemma arl-set-rule[sep-heap-rules]:
i<length l ⟹
<is-array-list l a>
arl-set a i x
<is-array-list (l[i:=x])>
by (sep-auto simp: arl-set-def is-array-list-def split: prod.split)

definition arl-assn A ≡ hr-comp is-array-list ((the-pure A)|list-rel)
lemmas [safe-constraint-rules] = CN-FALSEI[of is-pure arl-assn A for A]

lemma arl-assn-comp: is-pure A ⟹ hr-comp (arl-assn A) ((B)|list-rel) = arl-assn
(hr-comp A B)
  unfolding arl-assn-def
  by (auto simp: hr-comp-the-pure hr-comp-assoc list-rel-compp)

lemma arl-assn-comp’: hr-comp (arl-assn id-assn) ((B)|list-rel) = arl-assn (pure B)
  by (simp add: arl-assn-comp)

context
\textbf{notes} \([\text{intro}] = \text{h\textsuperscript{ref}I \, hn-refineI [THEN \, hn-refine-preI]}]$
\textbf{notes} \([\text{simp}] = \text{pure-def \, hn-ctxt-def \, invalid-assn-def}$

\texttt{begin}

\textbf{lemma} \texttt{arl-empty-hnr-aux}: \((\text{uncurry0} \, \text{arl-empty}, \text{uncurry0} \, (\text{RETURN} \, \text{op-list-empty})) \in \text{unit-assn}^k \rightarrow_a \text{is-array-list} \)
\texttt{by sep-auto}
\texttt{sepref-decl-impl (no-register) \text{arl-empty}: \text{arl-empty-hnr-aux} \)}.

\textbf{lemma} \texttt{arl-empty-sz-hnr-aux}: \((\text{uncurry0} \, (\text{arl-empty-sz} \, N), \text{uncurry0} \, (\text{RETURN} \, \text{op-list-empty})) \in \text{unit-assn}^k \rightarrow_a \text{is-array-list} \)
\texttt{by sep-auto}
\texttt{sepref-decl-impl (no-register) \text{arl-empty-sz}: \text{arl-empty-sz-hnr-aux} \)}.

\textbf{definition} \texttt{op-arl-empty} $\equiv \text{op-list-empty}$
\textbf{definition} \texttt{op-arl-empty-sz} \((N :: \text{nat}) \equiv \text{op-list-empty} $
\textbf{lemma} \texttt{arl-copy-hnr-aux}: \((\text{arl-copy}, \text{RETURN} \, \text{o \, op-list-copy}) \in \text{is-array-list}^k \rightarrow_a \text{is-array-list} \)
\texttt{by sep-auto}
\texttt{sepref-decl-impl \text{arl-copy}: \text{arl-copy-hnr-aux} \)}.

\textbf{lemma} \texttt{arl-append-hnr-aux}: \((\text{uncurry} \, \text{arl-append}, \text{uncurry} \, (\text{RETURN} \, \text{oo \, op-list-append})) \in (\text{is-array-list}^d \ast_a \text{id-assn}^k) \rightarrow_a \text{is-array-list} \)
\texttt{by sep-auto}
\texttt{sepref-decl-impl \text{arl-append}: \text{arl-append-hnr-aux} \)}.

\textbf{lemma} \texttt{arl-length-hnr-aux}: \((\text{arl-length}, \text{RETURN} \, \text{o \, op-list-length}) \in \text{is-array-list}^k \rightarrow_a \text{nat-assn} \)
\texttt{by sep-auto}
\texttt{sepref-decl-impl \text{arl-length}: \text{arl-length-hnr-aux} \)}.

\textbf{lemma} \texttt{arl-is-empty-hnr-aux}: \((\text{arl-is-empty}, \text{RETURN} \, \text{o \, op-list-is-empty}) \in \text{is-array-list}^k \rightarrow_a \text{bool-assn} \)
\texttt{by sep-auto}
\texttt{sepref-decl-impl \text{arl-is-empty}: \text{arl-is-empty-hnr-aux} \)}.

\textbf{lemma} \texttt{arl-last-hnr-aux}: \((\text{arl-last}, \text{RETURN} \, \text{o \, op-list-last}) \in [\text{pre-list-last}]_a \text{is-array-list}^k \rightarrow \text{id-assn} \)
\texttt{by sep-auto}
\texttt{sepref-decl-impl \text{arl-last}: \text{arl-last-hnr-aux} \)}.

\textbf{lemma} \texttt{arl-butlast-hnr-aux}: \((\text{arl-butlast}, \text{RETURN} \, \text{o \, op-list-butlast}) \in [\text{pre-list-butlast}]_a \text{is-array-list}^d \rightarrow \text{is-array-list} \)
\texttt{by sep-auto}
\texttt{sepref-decl-impl \text{arl-butlast}: \text{arl-butlast-hnr-aux} \)}.

\textbf{lemma} \texttt{arl-get-hnr-aux}: \((\text{uncurry} \, \text{arl-get}, \text{uncurry} \, (\text{RETURN} \, \text{oo \, op-list-get})) \in \)}

274
\[ \lambda (l, i), i < \text{length} \ l \] \to \text{id-assn} \\
\text{by sep-auto} \\
\text{sepref-decl-impl} \ arl-get : \ arl-get-hnr-aux . \\

\text{lemma} \ arl-set-hnr-aux : (\text{uncurry2} \ arl-set, \text{uncurry2} (\text{RETURN} \ \text{o00} \ \text{op-list-set})) \\
\in [\lambda ((l, i), -), i < \text{length} \ l] \to \text{is-array-list} \\
\text{by sep-auto} \\
\text{sepref-decl-impl} \ arl-set : \ arl-set-hnr-aux . \\

\text{sepref-definition} \ arl-swap \ is \ \text{uncurry2} \ \text{mop-list-swap} :: ((\text{arl-assn} \ \text{id-assn})^d \ *_a \ \text{nat-assn}^k *_a \ \text{id-assn}^k) \rightarrow \text{arl-assn} \ \text{id-assn} \\
\text{unfolding} \ \text{gen-mop-list-swap}[\text{abs-def}] \\
\text{by sepref} \\
\text{sepref-decl-impl} \ (\text{ismop}) \ arl-swap : \ arl-swap.\text{refine} . \\
\text{end} \\

\text{interpretation} \ arl : \ \text{list-custom-empty} \ arl-assn \ A \ \text{arl-empty} \ \text{op-arl-empty} \\
\text{apply} \ \text{unfold-locales} \\
\text{apply} \ (\text{rule} \ \text{arl-empty-hnr}) \\
\text{by} \ (\text{auto simp: op-arl-empty-def}) \\

\text{lemma} [\text{def-pat-rules}] : \ \text{op-arl-empty-sz} N \equiv \text{UNPROTECT} \ (\text{op-arl-empty-sz} N) \\
\text{by simp} \\
\text{interpretation} \ \text{arl-sz} : \ \text{list-custom-empty} \ arl-assn \ A \ \text{arl-empty-sz} N \ \text{PR-CONST} \\
(\text{op-arl-empty-sz} N) \\
\text{apply} \ \text{unfold-locales} \\
\text{apply} \ (\text{rule} \ \text{arl-empty-sz-hnr}) \\
\text{by} \ (\text{auto simp: op-arl-empty-sz-def}) \\
\text{end} \\

\textbf{3.7 Implementation of Heaps with Arrays} \\

\textbf{3.7.1 Setup of the Sepref-Tool} \\
\textbf{context} \\
\text{fixes} \ \text{prio} :: 'a::{heap,default} \Rightarrow 'b::linorder \\

\textbf{275}
interpretation heapstruct prio 

definition heap-rel A ≡ hr-comp (hr-comp (arl-assn id-assn) heap-rel1) ((the-pure A) mset-rel)
end

locale heapstruct-impl =
  fixes prio :: 'a::{heap, default} ⇒ 'b::linorder
begin
  sublocale heapstruct prio 

definition rel ≡ arl-assn id-assn

abbreviation sepref-register

lemma [sepref-import-param]: (prio, prio) ∈ Id → Id by simp

lemma [sepref-import-param]:
  ((≤), (≤)::'b ⇒ _) ∈ Id → Id → bool-rel
  ((<), (<)::'b ⇒ _) ∈ Id → Id → bool-rel
  by simp-all

sepref-register
  update-op
  val-of-op
  PR-CONST prio-of-op
  exch-op
  valid
  length::'a list ⇒ -
  append-op
  butlast-op

  PR-CONST sink-op
  PR-CONST swim-op
  PR-CONST repair-op

lemma [def-pat-rules]:
  heapstruct.prio-of-op$prio ≡ PR-CONST prio-of-op
  heapstruct.sink-op$prio ≡ PR-CONST sink-op
  heapstruct.swim-op$prio ≡ PR-CONST swim-op
  heapstruct.repair-op$prio ≡ PR-CONST repair-op
  by simp-all
end

context
  fixes prio :: 'a::{heap, default} ⇒ 'b::linorder
3.7.2 Synthesis of operations

Note that we have to repeat some boilerplate per operation. It is future work to add more automation here.

\[ \text{sepref-definition } \text{update-impl} \text{ is uncurry2 } \text{update-op} :: \text{rel}^d \ast_a \text{nat-assn}^k \ast_a \text{id-assn}^k \rightarrow_a \text{rel} \]

unfolding \text{update-op-def[abs-def]}

by sepref

lemmas [sepref-fr-rules] = update-impl.refine

\[ \text{sepref-definition } \text{val-of-impl} \text{ is uncurry } \text{val-of-op} :: \text{rel}^k \ast_a \text{nat-assn}^k \rightarrow_a \text{id-assn} \]

unfolding \text{val-of-op-def[abs-def]}

by sepref

lemmas [sepref-fr-rules] = val-of-impl.refine

\[ \text{sepref-definition } \text{exch-impl} \text{ is uncurry2 } \text{exch-op} :: \text{rel}^d \ast_a \text{nat-assn}^k \ast_a \text{nat-assn}^k \rightarrow_a \text{rel} \]

unfolding \text{exch-op-def[abs-def]}

by sepref

lemmas [sepref-fr-rules] = exch-impl.refine

\[ \text{sepref-definition } \text{valid-impl} \text{ is uncurry } \text{(RETURN oo valid)} :: \text{rel}^k \ast_a \text{nat-assn}^k \rightarrow_a \text{bool-assn} \]

unfolding \text{valid-def[abs-def]}

by sepref

lemmas [sepref-fr-rules] = valid-impl.refine

\[ \text{sepref-definition } \text{prio-impl} \text{ is uncurry } \text{(PR-CONST prio-op)} :: \text{rel}^k \ast_a \text{nat-assn}^k \rightarrow_a \text{id-assn} \]

unfolding \text{prio-of-op-def[abs-def]} \text{ PR-CONST-def}

by sepref

lemmas [sepref-fr-rules] = prio-impl.refine

\[ \text{sepref-definition } \text{swim-impl} \text{ is uncurry } \text{(PR-CONST swim-op)} :: \text{rel}^d \ast_a \text{nat-assn}^k \rightarrow_a \text{rel} \]

unfolding \text{swim-op-def[abs-def]} \text{ parent-def PR-CONST-def}

by sepref

lemmas [sepref-fr-rules] = swim-impl.refine

\[ \text{sepref-definition } \text{sink-impl} \text{ is uncurry } \text{(PR-CONST sink-op)} :: \text{rel}^d \ast_a \text{nat-assn}^k \rightarrow_a \text{rel} \]

unfolding \text{sink-op-opt-def[abs-def]} \text{ sink-op-opt-eq[symmetric,abs-def]} \text{ PR-CONST-def}

by sepref

lemmas [sepref-fr-rules] = sink-impl.refine
lemmas \([f\text{comp-norm-unfold}] = \text{heap-rel-def}[\text{symmetric}]\)

sepref-definition empty-impl is uncurry0 empty-op :: unit-assn \(k \rightarrow_a \text{rel}\)

unfolding empty-op-def arl.fold-custom-empty

by sepref

sepref-decl-impl (no-register) heap-empty: empty-impl.refine[FCOMP empty-op-refine]

.

sepref-definition is-empty-impl is is-empty-op :: \(\text{rel}^k \rightarrow_a \text{bool-assn}\)

unfolding is-empty-op-def[abs-def]

by sepref

sepref-decl-impl heap-is-empty: is-empty-impl.refine[FCOMP is-empty-op-refine]

.

sepref-definition insert-impl is uncurry insert-op :: id-assn \(k \ast \text{a} \rightarrow_a \text{rel}\)

unfolding insert-op-def[abs-def] append-op-def

by sepref

sepref-decl-impl heap-insert: insert-impl.refine[FCOMP insert-op-refine]

.

sepref-definition pop-min-impl is pop-min-op :: \(\text{rel}^d \rightarrow_a \text{prod-assn id-assn rel}\)

unfolding pop-min-op-def[abs-def] butlast-op-def

by sepref

sepref-decl-impl (no-mop) heap-pop-min: pop-min-impl.refine[FCOMP pop-min-op-refine]

.

sepref-definition peek-min-impl is peek-min-op :: \(\text{rel}^k \rightarrow_a \text{id-assn}\)

unfolding peek-min-op-def[abs-def]

by sepref

sepref-decl-impl (no-mop) heap-peek-min: peek-min-impl.refine[FCOMP peek-min-op-refine]

.

end

definition [simp]: heap-custom-empty \(\equiv\) op-mset-empty

interpretation heap: mset-custom-empty

heap-rel prio A empty-impl heap-custom-empty for prio A

apply unfold-locales

apply (rule heap-empty-hnr)

by simp

3.7.3 Regression Test

export-code empty-impl is-empty-impl insert-impl pop-min-impl peek-min-impl

checking SML

278
definition sort-by-prio prio l ≡ do 
  q ← nfoldli l (λ-. True) (λx q. mop-mset-insert x q) heap-custom-empty;
  (l,q) ← WHILET (λ(l,q). ¬op-mset-is-empty q) (λ(l,q). do 
    (x,q) ← mop-prio-pop-min prio q;
    RETURN (l@x],q)
  }) (op-art-empty,q);
  RETURN l
}

context fixes prio: 'a::{default,heap} ⇒ 'b::linorder begin
sepref-definition sort-impl is
  sort-by-prio prio :: (list-assn (id-assn::'a::{default,heap} ⇒ -))→→ arl-assn
id-assn
  unfolding sort-by-prio-def[abs-def]
  by sepref
end
definition sort-impl-nat ≡ sort-impl (id::nat⇒nat)

export-code sort-impl checking SML

ML :
  @{code sort-impl-nat} (map @{code nat-of-integer} [4,1,7,2,3,9,8,62]) ()

hide-const sort-impl sort-impl-nat
hide-fact sort-impl-def sort-impl-nat-def sort-impl.refine

end

3.8 Map Interface

theory IICF-Map
imports ../../Sepref
begin

3.8.1 Parametricity for Maps
definition [to-relAPP]: map-rel K V ≡ (K → (V)option-rel)
  ∩ \{ (mi,m). dom mi ⊆ Domain K ∧ dom m ⊆ Range K \}

lemma bi-total-map-rel-eq:
  [IS-RIGHT-TOTAL K; IS-LEFT-TOTAL K] \implies (K,V)map-rel = K → (V)option-rel
  unfolding map-rel-def IS-RIGHT-TOTAL-def IS-LEFT-TOTAL-def
  by (auto dest: fun-relD)

lemma map-rel-Id[simp]: (Id,Id)map-rel = Id
  unfolding map-rel-def by auto

279
lemma map-rel-empty1-simp[simp]:
\((\text{Map.empty}, m) \in (K, V)\) map-rel \(\iff\) \(m = \text{Map.empty}\)
apply (auto simp: map-rel-def)
by (meson RangeE domIff option-rel-simp(1) subsetCE tagged-fun-relD-none)

lemma map-rel-empty2-simp[simp]:
\((m, \text{Map.empty}) \in (K, V)\) map-rel \(\iff\) \(m = \text{Map.empty}\)
apply (auto simp: map-rel-def)
by (meson Domain_cases domIff fun-relD2 option-rel-simp(2) subset-eq)

lemma map-rel-obtain1:
assumes 1: \((m, n) \in (K, V)\) map-rel
assumes 2: \(n l = \text{Some } w\)
obtains \(k v\) where \(m k = \text{Some } v\) \((k, l) \in K\) \((v, w) \in V\)
using 1 unfolding map-rel-def
proof clarsimp
assume \(R: (m, n) \in K \to \{V\}\) option-rel
assume \(\text{dom } n \subseteq \text{Range } K\)
with 2 obtain \(k\) where \((k, l) \in K\) by auto
moreover from fun-relD[OF \(R\) this] have \((m k, n l) \in \{V\}\) option-rel .
with 2 obtain \(v\) where \(m k = \text{Some } v\) \((v, w) \in V\) by (cases \(m k\); auto)
ultimately show thesis by \(\neg\) (rule that)
qed

lemma map-rel-obtain2:
assumes 1: \((m, n) \in (K, V)\) map-rel
assumes 2: \(m k = \text{Some } v\)
obtains \(l w\) where \(n l = \text{Some } w\) \((k, l) \in K\) \((v, w) \in V\)
using 1 unfolding map-rel-def
proof clarsimp
assume \(R: (m, n) \in K \to \{V\}\) option-rel
assume \(\text{dom } m \subseteq \text{Domain } K\)
with 2 obtain \(l\) where \((k, l) \in K\) by auto
moreover from fun-relD[OF \(R\) this] have \((m k, n l) \in \{V\}\) option-rel .
with 2 obtain \(w\) where \(n l = \text{Some } w\) \((v, w) \in V\) by (cases \(n l\); auto)
ultimately show thesis by \(\neg\) (rule that)
qed

lemma param-dom[param]: \((\text{dom}, \text{dom}) \in (K, V)\) map-rel \(\to\) \((K)\) set-rel
apply (clarsimp simp: set-rel-def; safe)
apply (erule (1) map-rel-obtain2; auto)
apply (erule (1) map-rel-obtain1; auto)
done

3.8.2 Interface Type
sepref-decl-intf \(\langle k, v\rangle\) i-map is \(k \to v\)
lemma [synth-rules]: \[ \text{INTF-OF-REL} \ K \text{ TYPE('k) ; INTF-OF-REL} \ V \text{ TYPE('v)} \]
\[
\Rightarrow \text{INTF-OF-REL} \ (\langle \text{K, V} \rangle \text{ map-rel}) \text{ TYPE}((\text{'k, 'v}) \text{ i-map}) \text{ by simp}
\]

3.8.3 Operations

\textbf{sepref-decl-op map-empty: Map.empty :: \langle \text{K, V} \rangle \text{ map-rel .}}

\textbf{sepref-decl-op map-is-empty: (=) Map.empty :: \langle \text{K, V} \rangle \text{ map-rel} \rightarrow \text{bool-rel}}

\hspace{1em} \text{apply (rule fref-ncI)}

\hspace{1em} \text{apply parametricity}

\hspace{1em} \text{apply (rule fun-relI; auto)}

\hspace{1em} \text{done}

\textbf{sepref-decl-op map-update: \lambda \text{k v m. m(\text{k} \mapsto \text{v}) :: \text{K} \rightarrow \text{V} \rightarrow \langle \text{K, V} \rangle \text{ map-rel} \rightarrow \langle \text{K, V} \rangle \text{ map-rel}}

\hspace{1em} \text{where single-valued K single-valued (K^{-1})}

\hspace{1em} \text{apply (rule fref-ncI)}

\hspace{1em} \text{apply parametricity}

\hspace{1em} \text{unfolding map-rel-def}

\hspace{1em} \text{apply (intro fun-relI)}

\hspace{1em} \text{apply (elim IntE; rule IntI)}

\hspace{1em} \text{apply (intro fun-relI)}

\hspace{1em} \text{apply parametricity}

\hspace{1em} \text{apply (simp add: pres-eq-iff-svb)}

\hspace{1em} \text{apply auto}

\hspace{1em} \text{done}

\textbf{sepref-decl-op map-delete: \lambda \text{k m. fun-upd m k None :: \text{K} \rightarrow \langle \text{K, V} \rangle \text{ map-rel} \rightarrow \langle \text{K, V} \rangle \text{ map-rel}}

\hspace{1em} \text{where single-valued K single-valued (K^{-1})}

\hspace{1em} \text{apply (rule fref-ncI)}

\hspace{1em} \text{apply parametricity}

\hspace{1em} \text{unfolding map-rel-def}

\hspace{1em} \text{apply (intro fun-relI)}

\hspace{1em} \text{apply (elim IntE; rule IntI)}

\hspace{1em} \text{apply (intro fun-relI)}

\hspace{1em} \text{apply parametricity}

\hspace{1em} \text{apply (simp add: pres-eq-iff-svb)}

\hspace{1em} \text{apply auto}

\hspace{1em} \text{done}

\textbf{sepref-decl-op map-lookup: \lambda \text{k (m::'k \mapsto 'v). m k :: \text{K} \rightarrow \langle \text{K, V} \rangle \text{ map-rel} \rightarrow \langle \text{V} \rangle \text{ option-rel}}

\hspace{1em} \text{apply (rule fref-ncI)}

\hspace{1em} \text{apply parametricity}

\hspace{1em} \text{unfolding map-rel-def}

\hspace{1em} \text{apply (intro fun-relI)}

\hspace{1em} \text{apply (elim IntE)}

\hspace{1em} \text{apply (elim IntE)}

281
apply parametricity
done

lemma in-dom-alt: \( k \in \text{dom } m \iff \neg \text{is-None } (m k) \) by (auto split: option.split)

sepref-decl-op map-contains-key: \( \lambda k. k \in \text{dom } m \iff K \to \langle K, V \rangle \text{map-rel } \to \text{bool-rel} \)
unfolding in-dom-alt
apply (rule fref-ncI)
apply parametricity
unfolding map-rel-def
apply (elim IntE)
apply parametricity
done

3.8.4 Patterns

lemma pat-map-empty[pat-rules]: \( \lambda_2 \cdot \text{None } \equiv \text{op-map-empty} \) by simp

lemma pat-map-is-empty[pat-rules]:
(\(=\)) \( m \cdot (\lambda_2 \cdot \text{None}) \equiv \text{op-map-is-empty } m \)
(\(=\)) \( (\lambda_2 \cdot \text{None}) m \equiv \text{op-map-is-empty } m \)
(\(=\)) \( (\text{dom } m) \cdot \{\} \equiv \text{op-map-is-empty } m \)
(\(=\)) \( \{\} \cdot (\text{dom } m) \equiv \text{op-map-is-empty } m \)
unfolding atomize-eq
by (auto dest: sym)

lemma pat-map-update[pat-rules]:
fun-upd\( m \cdot k \cdot (\text{Some } v) \equiv \text{op-map-update } ^\prime k ^\prime v ^\prime m \)
by simp

lemma pat-map-lookup[pat-rules]:
m\( k \equiv \text{op-map-lookup } ^\prime k ^\prime m \)
by simp

lemma op-map-delete-pat[pat-rules]:
(\(\mid\)) \( m \cdot (\text{uminus } (\text{insert } k \cdot \{\})) \equiv \text{op-map-delete } ^\prime k ^\prime m \)
fun-upd\( m \cdot k \cdot \text{None } \equiv \text{op-map-delete } ^\prime k ^\prime m \)
by (simp-all add: map-upd-eq-restrict)

lemma op-map-contains-key[pat-rules]:
(\(\in\)) \( k \cdot (\text{dom } m) \equiv \text{op-map-contains-key } ^\prime k ^\prime m \)
Not\((\in )\) \( (m k) \cdot \text{None } \equiv \text{op-map-contains-key } ^\prime k ^\prime m \)
by (auto intro!: eq-reflection)

3.8.5 Parametricity

locale map-custom-empty =
  fixes op-custom-empty :: 'k \to 'v
  assumes op-custom-empty-def: op-custom-empty = op-map-empty
begin
sepref-register op-custom-empty :: ('kx,'vx) i-map

282
lemma fold-custom-empty:
Map.empty = op-custom-empty
op-map-empty = op-custom-empty
map-map-empty = RETURN op-custom-empty

unfolding op-custom-empty-def by simp-all
end

end

3.9 Priority Maps

theory IICF-Prio-Map
imports IICF-Map
begin

This interface inherits from maps, and adds some operations

lemma uncurry-fun-rel-conv:
(uncurry f, uncurry g) ∈ A ×, B → R ←→ (f,g)∈A→B→R
by (auto simp: uncurry-def dest!: fun-relD intro: prod-relI)

lemma uncurry0-fun-rel-conv:
(uncurry0 f, uncurry0 g) ∈ unit-rel → R ←→ (f,g)∈R
by (auto dest!: fun-relD)

lemma RETURN-rel-conv0: (RETURN f, RETURN g)∈(A)nres-rel ←→ (f,g)∈A
by (auto simp: nres-rel-def)

lemma RETURN-rel-conv1: (RETURN o f, RETURN o g)∈A → ⟨B)nres-rel
←→ (f,g)∈A→B
by (auto simp: nres-rel-def dest!: fun-relD)

lemma RETURN-rel-conv2: (RETURN oo f, RETURN oo g)∈A → B → ⟨R)nres-rel
←→ (f,g)∈A→B→R
by (auto simp: nres-rel-def dest!: fun-relD)

lemma RETURN-rel-conv3: (RETURN ooo f, RETURN ooo g)∈A→B→C → ⟨R)nres-rel
←→ (f,g)∈A→B→C→R
by (auto simp: nres-rel-def dest!: fun-relD)

lemmas fref2param-unfold =
uncurry-fun-rel-conv uncurry0-fun-rel-conv
RETURN-rel-conv0 RETURN-rel-conv1 RETURN-rel-conv2 RETURN-rel-conv3

lemmas param-op-map-update[param] = op-map-update.fref[THEN fref-neD,
unfolded fref2param-unfold]
lemmas param-op-map-delete[param] = op-map-delete.fref[THEN fref-ncD, unfolded fref2param-unfold]
lemmas param-op-map-is-empty[param] = op-map-is-empty.fref[THEN fref-ncD, unfolded fref2param-unfold]

3.9.1 Additional Operations

sepref-decl-op map-update-new: op-map-update :: \[λ((k,v),m). k\notin dom m\] \(K \times V\ ×_r (K, V)\) map-rel
where single-valued K single-valued \((K^{-1})\).

sepref-decl-op map-update-ex: op-map-update :: \[λ((k,v),m). k\in dom m\] \(K \times V\ ×_r (K, V)\) map-rel
where single-valued K single-valued \((K^{-1})\).

sepref-decl-op map-delete-ex: op-map-delete :: \[λ(k,m)\] \(K \times (V::(v\times′ v)\ set))\) map-rel
where single-valued K single-valued \((K^{-1})\) IS-BELOW-ID V

proof goal-cases
  case 1
  have \([param]::(\leq,\leq)\) in \(I d_\rightarrow I d_\rightarrow boolean-rel\) by \(simp\)
  from 1 show ?case
    apply (parametricity add: param-and-cong1)
    apply (auto simp: IS-BELOW-ID-def map-rel-def dest!: fun-relD)
  done
  qed

sepref-decl-op pm-decrease-key: op-map-update
:: \[\lambda((k,v),m). k\in dom m \land prio v \leq prio (the (m k))\] \(K \times V\ ×_r (K, V)\) map-rel
→ \(K,(V::(v\times′ v)\ set))\) map-rel
where single-valued K single-valued \((K^{-1})\) IS-BELOW-ID V

proof goal-cases
  case 1
  have \([param]::(\leq,\leq)\) in \(I d_\rightarrow I d_\rightarrow boolean-rel\) by \(simp\)
  from 1 show ?case
    apply (parametricity add: param-and-cong1)
    apply (auto simp: IS-BELOW-ID-def map-rel-def dest!: fun-relD)
  done
  qed

284
lemma IS-BELOW-ID-D: \((a,b) \in R \rightarrow IS-BELOW-ID \ R \rightarrow a=b\) by (auto simp: IS-BELOW-ID-def)

sepref-decl-op pm-peek-min: \(\lambda m. SPEC (\lambda(k,v).
\ m \ k = Some \ v \land (\forall k' v'. m \ k' = Some \ v' \rightarrow prio v \leq prio v'))\)
\:: [Not o op-map-is-empty] \((K,V)\maprel \rightarrow K \times,(V::(\{v\times{v'}\} set))\times,(K,V)\maprel\)
where single-valued K single-valued \((K^{-1}) IS-BELOW-ID V\)
apply (rule frefl)
apply (intro nres-reff)
apply (clarsimp simp: pw-le-iff refine-pw-simps simp del)
apply (rule map-rel-obtain1, assumption, assumption)
apply1 (intro exI conjI allI impI; assumption?)

proof –
fix x y k' v' b w
assume \((x,y) \in (K, V)\maprel y k' = Some \ v'\)
then obtain k v where \(\{k,k'\} \in K (v,v') \in V x k = Some \ v\)
by (rule map-rel-obtain1)

assume IS-BELOW-ID V \((b, w) \in V\)
with \((v,v') \in V\) have [simp]: \(b=w v=v'\) by (auto simp: IS-BELOW-ID-def)

assume \(\forall k' v', x k' = Some \ v' \rightarrow prio b \leq prio v'\)
with \((x k = Some \ v)\) show prio w \(\leq prio v'\)
by auto

qed

sepref-decl-op pm-pop-min: \(\lambda m. SPEC (\lambda((k,v),m').
\ m \ k = Some \ v \land m' = op-map-delete \ k m
\land (\forall k' v'. m \ k' = Some \ v' \rightarrow prio v \leq prio v')\)
\:: [Not o op-map-is-empty] \((K,V)\maprel \rightarrow (K \times,V::\{v\times{v'}\} set)\times,(K,V)\maprel\)
where single-valued K single-valued \((K^{-1}) IS-BELOW-ID V\)
apply (rule frefl)
apply (intro nres-reff)
apply (clarsimp simp: pw-le-iff refine-pw-simps simp del: op-map-delete-def)
apply (rule map-rel-obtain2, assumption, assumption)
applyS parametricity

proof –
fix x y k' v' b w
assume \((x,y) \in (K, V)\maprel y k' = Some \ v'\)
then obtain k v where \(\{k,k'\} \in K (v,v') \in V x k = Some \ v\)
by (rule map-rel-obtain1)

assume IS-BELOW-ID V \((b, w) \in V\)
with \((v,v') \in V\) have [simp]: \(b=w v=v'\) by (auto simp: IS-BELOW-ID-def)

assume \(\forall k' v', x k' = Some \ v' \rightarrow prio b \leq prio v'\)
with \((x k = Some \ v)\) show prio w \(\leq prio v'\)


285
by auto
qed
end

end

3.10 Priority Maps implemented with List and Map

theory IICF-Abs-Heapmap
imports IICF-Abs-Heap HOL-Library.Rewrite../../Intf/IICF-Prio-Map
begin

type-synonym ('k,'v) ahm = 'k list × ('k ⇒ 'v)

3.10.1 Basic Setup

First, we define a mapping to list-based heaps

definition hmr-α :: ('k,'v) ahm ⇒ 'v heap where
  hmr-α ≡ λ(pq,m). map (the o m) pq

definition hmr-invar ≡ λ(pq,m). distinct pq ∧ dom m = set pq

definition hmr-rel ≡ br hmr-α hmr-invar

lemmas hmr-rel-defs = hmr-rel-def br-def hmr-α-def hmr-invar-def

lemma hmr-empty-invar[simp]: hmr-invar ([],Map.empty)
  by (auto simp: hmr-invar-def)

locale hmstruct = h: heapstruct prio for prio :: 'v ⇒ 'b::linorder
begin

Next, we define a mapping to priority maps.

definition heapmap-α :: ('k,'v) ahm ⇒ ('k ⇒ 'v) where
  heapmap-α ≡ λ(pq,m). m

definition heapmap-invar :: ('k,'v) ahm ⇒ bool where
  heapmap-invar ≡ λhm. hmr-invar hm ∧ h.heap-invar (hmr-α hm)

definition heapmap-rel ≡ br heapmap-α heapmap-invar

lemmas heapmap-rel-defs = heapmap-rel-def br-def heapmap-α-def heapmap-invar-def

lemma [refine-dref-RELATES]: RELATES hmr-rel by (simp add: RELATES-def)
lemma h-heap-invar\[simp\]: heapmap-invar hm \Rightarrow h.heap-invar (hmr-\alpha hm)
by (simp add: heapmap-invar-def)

lemma hmr-invarI[simp]: heapmap-invar hm \Rightarrow hmr-invar hm
unfolding heapmap-invar-def by blast

lemma set-hmr-\alpha[simp]: hmr-invar hm \Rightarrow set (hmr-\alpha hm) = ran (heapmap-\alpha hm)
apply (clarsimp simp: hmr-\alpha-def hmr-invar-def heapmap-\alpha-def)
apply force
done

lemma in-h-hmr-\alpha-conv[simp]: hmr-invar hm \Rightarrow x \in\# h.(\alpha (hmr-\alpha hm)) \longleftrightarrow x \in ran (heapmap-\alpha hm)
by (force simp: hmr-\alpha-def hmr-invar-def heapmap-\alpha-def in-multiset-in-set ran-is-image)

3.10.2 Basic Operations

In this section, we define the basic operations on heapmaps, and their relations to heaps and maps.

Length

Length of the list that represents the heap

definition hm-length :: (\'k,\'v) ahm \Rightarrow nat where
hm-length \equiv \lambda (pq,\cdot). length pq

lemma hm-length-refine: (hm-length, length) \in hmr-rel \Rightarrow nat-rel
apply (intros fun-refl)
unfolding hm-length-def
by (auto simp: hmr-rel-defs)

lemma hm-length-hmr-\alpha[simp]: length (hmr-\alpha hm) = hm-length hm
by (auto simp: hm-length-def hmr-\alpha-def split: prod.splits)

lemmas [refine] = hm-length-refine[param-fo]

Valid

Check whether index is valid

definition hm-valid hm \equiv i > 0 \land i \leq hm-length hm
lemma \textit{hm-valid-refine}: \((\textit{hm-valid}, \textit{h.valid}) : \text{hmr-rel} \rightarrow \text{nat-rel} \rightarrow \text{bool-rel})
apply (intro fun-refl)
unfolding \textit{hm-valid-def} \textit{h.valid-def}
by (parametricity add: \textit{hm-length-refine})

lemma \textit{hm-valid-hmr-\(\alpha\)[simp]}: \textit{h.valid} \((\textit{hmr-\(\alpha\)} \textit{hm}) = \text{hm-valid} \textit{hm}
by (intro ext) (auto simp: \textit{h.valid-def} \textit{hm-valid-def})

\textbf{Key-Of}

definition \textit{hm-key-of} :: \((\text{'k}, \text{'v}) \text{ahm} \Rightarrow \text{nat} \Rightarrow \text{'k})\)
where
\textit{hm-key-of} \equiv \lambda \((\text{pq}, \text{m})\) \text{i}. \text{pq}!(\text{i} - 1)
definition \textit{hm-key-of-op} :: \((\text{'k}, \text{'v}) \text{ahm} \Rightarrow \text{nat} \Rightarrow \text{'k nres})\)
where
\textit{hm-key-of-op} \equiv \lambda \((\text{pq}, \text{m})\) \text{i}. \text{ASSERT} (\text{i} > 0) \Rightarrow \text{mop-list-get} \text{pq} (\text{i} - 1)

lemma \textit{hm-key-of-op-unfold}:
shows \textit{hm-key-of-op} \textit{hm} \text{i} = \text{ASSERT} (\text{hm-valid} \textit{hm} \text{i}) \Rightarrow \text{RETURN} (\text{hm-key-of} \textit{hm} \text{i})
unfolding \textit{hm-valid-def} \textit{hm-length-def} \textit{hm-key-of-def} \textit{hm-key-of-op-def}
by (auto split: prod.splits simp: pw-eq-iff refine-pw-simps)

lemma \textit{val-of-hmr-\(\alpha\)[simp]}: \text{hm-valid} \textit{hm} \text{i} = \Rightarrow \textit{h.val-of} \((\text{hmr-\(\alpha\)} \textit{hm}) \text{i})
= \text{the} \((\text{heapmap-\(\alpha\)} \textit{hm} \text{(hm-key-of} \textit{hm} \text{i}))
by (auto simp: \textit{hmr-\(\alpha\)-def} \textit{h.valid-def} \textit{heapmap-\(\alpha\)-def} \textit{hm-key-of-def} \textit{hm-valid-def}
\textit{hm-length-def}
split: prod.splits)

lemma \textit{hm-\(\alpha\)-key-ex[simp]}:
\[ [\text{hmr-invar} \textit{hm}; \text{hm-valid} \textit{hm} \text{i}] \Rightarrow (\text{heapmap-\(\alpha\)} \textit{hm} \text{(hm-key-of} \textit{hm} \text{i}) \neq \text{None}) \]
unfolding \textit{heapmap-invar-def} \textit{hm-key-of-op-def} \textit{hm-valid-def} \textit{heapmap-\(\alpha\)-def}
\textit{hm-key-of-def} \textit{hm-length-def}
by (auto split: prod.splits)

\textbf{Lookup}

abbreviation (input) \textit{hm-lookup} where \textit{hm-lookup} \equiv \text{heapmap-\(\alpha\)}
definition \textit{hm-the-lookup-op} \textit{hm} \text{k} \equiv
\text{ASSERT} (\text{heapmap-\(\alpha\)} \textit{hm} \text{k} \neq \text{None} \land \text{hmr-invar} \textit{hm})
\Rightarrow \text{RETURN} (\text{the} \((\text{heapmap-\(\alpha\)} \textit{hm} \text{k}))

\textbf{Exchange}

Exchange two indices
definition \textit{hm-exch-op} \equiv \lambda \((\text{pq}, \text{m})\) \text{i} \text{j}. \text{do} \{ \text{ASSERT} \ (\text{hm-valid} \ (\text{pq}, \text{m}) \text{i}); \text{ASSERT} \ (\text{hm-valid} \ (\text{pq}, \text{m}) \text{j}); \}

288
assert (hmr-invar (pq,m));
    pq ← map-list-swap pq (i - 1) (j - 1);
    return (pq,m)
}

lemma hm-exch-op-invar: hm-exch-op hm i j ≤ₙ SPEC hmr-invar
    unfolding hm-exch-op-def h.exch-op-def h.val-of-op-def h.update-op-def
    apply simp
    apply refine-vcg
    apply (auto simp: hm-valid-def map-swap hm-length-def hmr-rel-defs)
    done

lemma hm-exch-op-refine: (hm-exch-op,h.exch-op) ∈ hmr-rel → nat-rel → nat-rel → ⟨hmr-rel⟩ nres-rel
    apply (intro fun-relI nres-relI)
    unfolding hm-exch-op-def h.exch-op-def h.val-of-op-def h.update-op-def
    apply simp
    apply refine-vcg
    apply (auto simp: hm-valid-def map-swap hm-length-def hmr-rel-defs)
    done

lemmas hm-exch-op-refine׳[refine] = hm-exch-op-refine[param-fo, THEN nres-relD]

definition hm-exch :: ('k,'v) ahm ⇒ nat ⇒ nat ⇒ ('k,'v) ahm
    where hm-exch ≡ λ(pq,m) i j. (swap pq (i-1) (j-1),m)

lemma hm-exch-op-α-correct: hm-exch-op hm i j ≤ₙ SPEC (λhm'.
    hm-valid hm i ∧ hm-valid hm j ∧ hm'≡hm-exch hm i j)
    unfolding hm-exch-op-def
    apply refine-vcg
    apply (vc-solve simp: hm-valid-def hm-length-def heapmap-α-def solve: asm-rl)
    apply (auto simp add: hm-key-of-def hm-exch-def split: prod.splits)
    done

lemma hm-exch-α[simp]: heapmap-α (hm-exch hm i j) = (heapmap-α hm)
    by (auto simp: heapmap-α-def hm-exch-def split: prod.splits)

lemma hm-exch-valid[simp]: hm-valid (hm-exch hm i j) = hm-valid hm
    by (intro ext) (auto simp: hm-valid-def hm-length-def hm-exch-def split: prod.splits)

lemma hm-exch-length[simp]: hm-length (hm-exch hm i j) = hm-length hm
    by (auto simp: hm-length-def hm-exch-def split: prod.splits)

lemma hm-exch-same[simp]: hm-exch hm i i = hm
    by (auto simp: hm-exch-def split: prod.splits)

lemma hm-key-of-exch-conv[simp]:
\[ \text{hm-valid hm i; hm-valid hm j; hm-valid hm k} \implies \]
\[ \text{hm-key-of (hm-exch hm i j) k} = ( \]
\begin{align*}
\text{if } k &= i \text{ then hm-key-of hm j} \\
\text{else if } k &= j \text{ then hm-key-of hm i} \\
\text{else hm-key-of hm k}
\end{align*}
\)

\text{unfolding hm-exch-def hm-valid-def hm-length-def hm-key-of-def}

\text{by (auto split: prod.splits)}

\text{lemma hm-key-of-exch-matching[simp]:}
\[ \text{hm-valid hm i; hm-valid hm j} \implies \text{hm-key-of (hm-exch hm i j) i} = \text{hm-key-of hm j} \]
\[ \text{hm-valid hm i; hm-valid hm j} \implies \text{hm-key-of (hm-exch hm i j) j} = \text{hm-key-of hm i} \]
\text{by simp-all}

\text{Index}

\text{Obtaining the index of a key}

\text{definition hm-index} \equiv \lambda (pq, m) \ k. \ \text{index pq k + 1}

\text{lemma hm-index-valid[simp]:} \ [\text{hm-invar hm}; \ \text{heapmap-α hm k} \neq \text{None}] \implies \text{hm-valid hm (hm-index hm k)}

\text{by (force simp: hm-valid-def heapmap-α-def hm-invar-def hm-index-def hm-length-def Suc-le-eq)}

\text{lemma hm-index-key-of[simp]:} \ [\text{hm-invar hm}; \ \text{heapmap-α hm k} \neq \text{None}] \implies \text{hm-key-of hm (hm-index hm k)} = k

\text{by (force simp: hm-valid-def heapmap-α-def hm-invar-def hm-index-def hm-length-def hm-key-of-def Suc-le-eq)}

\text{definition hm-index-op} \equiv \lambda (pq, m) \ k.
\text{do} \{ \]
\text{ASSERT (hm-invar (pq, m) \land heapmap-α (pq, m) k \neq \text{None});}
\text{i ← mop-list-index pq k;}
\text{RETURN (i+1)}
\}

\text{lemma hm-index-op-correct:}
\text{assumes hm-invar hm}
\text{assumes heapmap-α hm k \neq \text{None}}
\text{shows hm-index-op hm k} \leq \text{SPEC (λx. \text{r= hm-index hm k)}}
\text{using assms unfolding hm-index-op-def}
\text{apply refine-vcg}
\text{apply (auto simp: heapmap-α-def hm-invar-def hm-index-def index-nth-id)}
\text{done}

\text{lemmas [refine-vcg] = hm-index-op-correct}
Update

Updating the heap at an index

definition hm-update-op :: (′k,′v) ahm ⇒ nat ⇒ (′k,′v) ahm nres where
hm-update-op ≡ λ(pq,m) i v. do {
  ASSERT (hm-valid (pq,m) i ∧ hmr-invar (pq,m));
  k ← mop-list-get pq (i - 1);
  RETURN (pq, m(k ↦→ v))
}

lemma hm-update-op-invar: hm-update-op hm k v ≤₉ SPEC hmr-invar
unfolding hm-update-op-def h.update-op-def
apply refine-vcg
by (auto simp: hmr-rel-defs map-distinct-upd-conv hm-valid-def hm-length-def)

lemma hm-update-op-refine: (hm-update-op, h.update-op) ∈ hmr-rel → nat-rel
→ Id → (hm-rel)nres-rel
apply (intro fun-relI nres-relI)
unfolding hm-update-op-def h.update-op-def mop-list-get-alt mop-list-set-alt
apply refine-vcg
apply (auto simp: hmr-rel-defs map-distinct-upd-conv hm-valid-def hm-length-def)
done

lemmas [refine] = hm-update-op-refine[param-fo, THEN nres-relID]

lemma hm-update-op-α-correct:
assumes hmr-invar hm
assumes heapmap-α hm k ≠ None
shows hm-update-op hm (hm-index hm k) v ≤₉ SPEC (λhm′. heapmap-α
hm′ = (heapmap-α hm)(k↦v))
using assms
unfolding hm-update-op-def
apply refine-vcg
apply (force simp: heapmap-rel-defs hmr-rel-defs hm-index-def)
done

Butlast

Remove last element

definition hm-butlast-op :: (′k,′v) ahm ⇒ (′k,′v) ahm nres where
hm-butlast-op ≡ λ(pq,m). do {
  ASSERT (hm-invar (pq,m));
  k ← mop-list-get pq (length pq - 1);
  pq ← mop-list-butlast pq;
  let m = m(k:=None);
  RETURN (pq,m)
}

lemma hm-butlast-op-refine: (hm-butlast-op, h.butlast-op) ∈ hmr-rel → (hm-rel)nres-rel
supply simp del = map-upd-eq-restrict
apply (intro fun-refl nres-refl)
unfolding hm-butlast-op-def h.butlast-op-def
apply simp
apply refine-vcg
apply (clarsimp-all simp: hm-butlast-op-def h)

butlast-op-def
apply simp
apply refine-vcg
apply (clarsimp-all simp: hmr-rel-defs map-butlast distinct-butlast)
apply (auto simp: neq-Nil-rev-cone) []
done

lemmas [refine] = hm-butlast-op-refine[param-fo, THEN nres-relD]

lemma hm-butlast-op-α-correct: hm-butlast-op hm ≤ₜ SPEC (λhm'. heapmap-α hm' = (heapmap-α hm)(hm-key-of hm (hm-length hm)) := None ))
unfolding hm-butlast-op-def
apply refine-vcg
apply (auto simp: heapmap-α-def hm-key-of-def hm-length-def)
done

Append

Append new element at end of heap

definition hm-append-op :: (k,v) ahm ⇒ k ⇒ v ⇒ (k,v) ahm nres
where hm-append-op ≡ λ(pq,m) k v. do
  ASSERT (k ∉ dom m);
  ASSERT (hmr-invar (pq,m));
  pq ← mop-list-append pq k;
  let m = m (k ↦→ v);
  RETURN (pq,m)

lemma hm-append-op-invar: hm-append-op hm k v ≤ₜ SPEC hmr-invar
unfolding hm-append-op-def h.append-op-def
apply refine-vcg
unfolding heapmap-α-def hmr-rel-defs
apply (auto simp: )
done

lemma hm-append-op-refine: [] heapmap-α hm k = None; (hm,h)∈hmr-rel []
⇒ (hm-append-op hm k v, h.append-op h v) ∈ ⟨hmr-rel⟩nres-rel
apply (intro fun-refl nres-refl)
unfolding hm-append-op-def h.append-op-def
apply refine-vcg
unfolding heapmap-α-def hmr-rel-defs
apply (auto simp: )
done

lemmas hm-append-op-refine"[refine] = hm-append-op-refine[param-fo, THEN nres-relD]
lemma hm-append-op-α-correct:
    hm-append-op hm k v ≤ₚ SPEC (λhm'. heapmap-α hm' = (heapmap-α hm)
    (k ↦ v))
    unfolding hm-append-op-def
    apply refine-vcg
    by (auto simp: heapmap-α-def)

3.10.3 Auxiliary Operations

Auxiliary operations on heapmaps, which are derived from the basic operations, but do not correspond to operations of the priority map interface

We start with some setup

lemma heapmap-hmr-relI: (hm,h) ∈ heapmap-rel ⇒ (hm,hmr-α hm) ∈ hmr-rel
    by (auto simp: heapmap-rel-defs hmr-rel-defs)

lemma heapmap-hmr-relI': heapmap-invar hm ⇒ (hm,hmr-α hm) ∈ hmr-rel
    by (auto simp: heapmap-rel-defs hmr-rel-defs)

The basic principle how we prove correctness of our operations: Invariant preservation is shown by relating the operations to operations on heaps. Then, only correctness on the abstraction remains to be shown, assuming the operation does not fail.

lemma heapmap-nres-relI'":
    assumes hm ≤⇓ hmr-rel h'
    assumes h' ≤ SPEC (h.heap-invar)
    assumes hm ≤ₚ SPEC (λhm'. RETURN (heapmap-α hm') ≤ h)
    shows hm ≤⇓ heapmap-rel h
    using assms
    unfolding heapmap-rel-defs hmr-rel-def
    by (auto simp: pw-le-iff pw-leof-iff refine-pw-simps)

lemma heapmap-nres-relI'":
    assumes hm ≤⇓ hmr-rel h'
    assumes h' ≤ SPEC Φ
    assumes h', Φ h' ⇒ h.heap-invar h'
    assumes hm ≤ₚ SPEC (λhm'. RETURN (heapmap-α hm') ≤ h)
    shows hm ≤⇓ heapmap-rel h
    apply (rule heapmap-nres-relI')
    apply fact
    apply (rule order-trans, fact)
    apply (clarsimp; fact)
    apply fact
done
Val-of

Indexing into the heap

definition hm-val-of-op :: (′k,′v) ahm ⇒ nat ⇒ 'v nres where
hm-val-of-op ≡ λhm i. do {
  k ← hm-key-of-op hm i;
  v ← hm-the-lookup-op hm k;
  RETURN v
}

lemma hm-val-of-op-refine: (hm-val-of-op,h.val-of-op) ∈ (hmr-rel → nat-rel → ⟨Id⟩nres-rel)
apply (intro fun-relI nres-relI)
unfolding hm-val-of-op-def h.val-of-op-def
hm-key-of-op-def hm-key-of-def hm-valid-def hm-length-def
hm-the-lookup-op-def
apply clarsimp
apply (rule refine-IdD)
apply refine-vcg
apply (auto simp: hmr-rel-defs heapmap-α-def)
done

lemmas [refine] = hm-val-of-op-refine[param-fo, THEN nres-relD]

Prio-of

Priority of key

definition hm-prio-of-op h i ≡ do {v ← hm-val-of-op h i; RETURN (prio v)}

lemma hm-prio-of-op-refine: (hm-prio-of-op, h.prio-of-op) ∈ hmr-rel → nat-rel → ⟨Id⟩nres-rel
apply (intro fun-relI nres-relI)
unfolding hm-prio-of-op-def h.prio-of-op-def
apply refine-rcg
by auto

lemmas hm-prio-of-op-refine[param-fo, THEN nres-relD]

Swim

definition hm-swim-op :: (′k,′v) ahm ⇒ nat ⇒ (′k,′v) ahm nres where
hm-swim-op h i ≡ do {
  RECT (λswim (h,i). do {
    ASSERT (hm-valid h i ∧ h.swim-invar (hmr-α h) i);
    if hm-valid h (h.parent i) then do {
      ppi ← hm-prio-of-op h (h.parent i);
      pi ← hm-prio-of-op h i;
      if (¬ppi ≤ pi) then do {
        RETURN v
      }
    }
  })
}
\[ h \leftarrow \text{hm-exch-op } h \, i \, (h.\text{parent } i); \]
\[ \text{swim } (h, \, h.\text{parent } i) \]
\} else
\[ \text{RETURN } h \]
\} else
\[ \text{RETURN } h \]
\} (h,i)
\}

\textbf{lemma} \text{hm-swim-op-refine: } (\text{hm-swim-op}, \, h.\text{swim-op}) \in \text{hm-rel} \rightarrow \text{nat-rel} \rightarrow \langle \text{hm-rel}\rangle \, \text{nres-rel}

\begin{itemize}
  \item \text{apply} (\text{intro fun-relI } nres-relI)
  \item \text{unfolding} \text{hm-swim-op-def } h.\text{swim-op-def}
  \item \text{apply} \text{refine-rcg}
  \item \text{apply} \text{refine-dref-type}
  \item \text{apply} (\text{clarsimp-all simp: } \text{hm-valid-refine[param-fo, THEN } \text{IdD}])
  \item \text{apply} (\text{simp add: } \text{hm-rel-def in-br-conv})
\end{itemize}
\text{done}

\textbf{lemmas} \text{hm-swim-op-refine'}[\text{refine} ] = \text{hm-swim-op-refine}[\text{param-fo, THEN } \text{nres-relD}]

\textbf{lemma} \text{hm-swim-op-nofail-imp-valid: }
\text{nfail } (\text{hm-swim-op } h \text{m } i) \implies \text{hm-valid } h \text{m } i \land \text{h.swim-invar (hm-\alpha } h \text{m) } i

\begin{itemize}
  \item \text{unfolding} \text{hm-swim-op-def}
  \item \text{apply} (\text{subst (asm) RECT-unfold, refine-mono})
  \item \text{by} (\text{auto simp: refine-pw-simps})
\end{itemize}

\textbf{lemma} \text{hm-swim-op-\alpha-correct: } \text{hm-swim-op } h \text{m } i \leq_n \text{SPEC } (\lambda h \text{m}' \cdot \text{heapmap-\alpha } h \text{m}' = \text{heapmap-\alpha } h \text{m})

\begin{itemize}
  \item \text{apply} (\text{rule leof-add-nofailI})
  \item \text{apply} (\text{drule } \text{hm-swim-op-nofail-imp-valid})
  \item \text{unfolding} \text{hm-swim-op-def}
  \item \text{apply} (\text{rule RECT-rule-leof[where P=\lambda h \text{m}',i). hm-valid h \text{m}' i \land heapmap-\alpha h \text{m}' = heapmap-\alpha h \text{m} and } V = \text{inv-image less-than snd})
\end{itemize}

\begin{itemize}
  \item \text{apply} \text{simp}
  \item \text{apply} \text{simp}
  \item \text{unfolding} \text{hm-prio-of-op-def } h \text{m-val-of-op-def}
  \item \text{hm-exch-op-def } h \text{m-key-of-op-def } h \text{m-the-lookup-op-def}
  \item \text{apply} (\text{refine-rcg})
  \item \text{apply} (\text{vc-solve simp add: } \text{hm-valid-def } h \text{m-length-def})
  \item \text{apply} \text{rprems}
  \item \text{apply} (\text{vc-solve simp: } \text{heapmap-\alpha-def } h.\text{parent-def})
\end{itemize}
\text{done}
Sink

**definition** \texttt{hm-sink-op} \\
**where** \\
\texttt{hm-sink-op} \( h \ k \equiv \text{RECT} (\lambda D (h,k). \text{do} \{\)} \\
\text{ASSERT \((k>0 \ \wedge \ k \leq \text{hm-length} \ h)\);} \\
\text{let \( \text{len} = \text{hm-length} \ h \);} \\
\text{if \((2\times k \leq \text{len})\) then \{\)} \\
\text{let \( j = 2\times k \);} \\
\text{\( \text{pj} \leftarrow \text{hm-prio-of-op} \ h \ j \);} \\
\text{if \( j < \text{len} \) then \{\)} \\
\text{\( \text{psj} \leftarrow \text{hm-prio-of-op} \ h \ (\text{Sac} \ j) \);} \\
\text{if \( \text{pj} > \text{psj} \) then RETURN \((j+1)\) else RETURN \(j\);\) } \\
\text{\} else RETURN \(j\);\) \\
\text{\}} \text{\text{do\}}} \text{\{\) \\
\text{\( \text{pj} \leftarrow \text{hm-prio-of-op} \ h \ j \);} \\
\text{\( \text{pk} \leftarrow \text{hm-prio-of-op} \ h \ k \);} \\
\text{if \( \text{pk} > \text{pj} \) then \{\)} \\
\text{\( \text{h} \leftarrow \text{hm-exch-op} \ h \ k \ j \);} \\
\text{\( D (h,j) \);} \\
\text{\} else \text{\{\) \\
\text{\( \text{RETURN} \ h \);} \\
\text{\}\} else RETURN \( h \);} \\
\text{\}}\) \((h,k)\) \\

**lemma** \texttt{hm-sink-op-refine}: \((\text{hm-sink-op}, h.\text{sink-op}) \in \text{hmr-rel} \rightarrow \text{nat-rel} \rightarrow \langle \text{hmr-rel} \rangle \text{nres-rel}\) \\
\text{apply \((\text{intro fun-relI nres-relI})\)} \\
\text{unfolding \texttt{hm-sink-op-def} h.\text{sink-op-opt-eq}[\text{symmetric}] h.\text{sink-op-opt-def}} \\
\text{apply \texttt{refine-rcg}} \\
\text{apply \texttt{refine-dref-type}} \\
\text{unfolding \texttt{hmr-rel-def} heapmap-rel-def} \\
\text{apply \((\text{clarsing-all simp: in-br-cone})\)} \\
\text{done}\)

**lemmas** \texttt{hm-sink-op-refine'}[\texttt{refine}] = \texttt{hm-sink-op-refine'}[\texttt{param-fo, THEN nres-relD}] \\

**lemma** \texttt{hm-sink-op-nofail-imp-valid}: \texttt{nofail (hm-sink-op hm i) \implies hm-valid hm i} \\
\text{unfolding \texttt{hm-sink-op-def}} \\
\text{apply \((\text{subst (asm) RECT-unfold, refine-mono})\)} \\
\text{by \((\text{auto simp: refine-pw-simps hm-valid-def})\)} \\

**lemma** \texttt{hm-sink-op-α-correct}: \texttt{hm-sink-op hm i \leq α SPEC (λhm'. heapmap-α hm' = heapmap-α hm)} \\
\text{apply \((\text{rule leof-add-nofail})\)} \\
\text{apply \((\text{drule hm-sink-op-nofail-imp-valid})\)} \\

296
unfolding \( \text{hm-sink-op-def} \)
apply (rule RECT-rule-leaf[where
pre=\( \lambda (hm',i). \text{hm-valid } hm' \land \text{heapmap-\( \alpha \) } hm' = \text{heapmap-\( \alpha \) } hm \land \text{hm-length } hm' = \text{hm-length } hm \)
and \( V = \text{measure } (\lambda (hm',i). \text{hm-length } hm' - i) \))
apply simp
apply simp

unfolding \( \text{hm-prio-of-op-def} \) \( \text{hm-val-of-op-def} \) \( \text{hm-exch-op-def} \)
\( \text{hm-key-of-op-def} \) \( \text{hm-the-lookup-op-def} \)
apply (refine-vcg)
apply (vc-solve simp add: \( \text{hm-valid-def} \) \( \text{hm-length-def} \))
apply rprems
apply (vc-solve simp: \( \text{heapmap-\( \alpha \) -def} \) \( h \).parent-def split: prod.splits)
apply (auto)
done

**Repair**

definition \( \text{hm-repair-op} hm i \equiv \{ \)
\( \text{hm} \leftarrow \text{hm-sink-op} hm i; \)
\( \text{hm} \leftarrow \text{hm-swim-op} hm i; \)
RETURN \( \text{hm} \)
\}

lemma \( \text{hm-repair-op-refine}: (\text{hm-repair-op}, h.\text{repair-op}) \in \text{hmr-rel} \to \text{nat-rel} \to (\text{hmr-rel} \to \text{nres-rel}) \)
apply (intro fun-relI nres-relI)
unfolding \( \text{hm-repair-op-def} \) h.\text{repair-op-def}
by refine-vcg

lemmas \( \text{hm-repair-op-refine}[^{\text{refine}}] = \text{hm-repair-op-refine}[^{\text{param-fo, THEN nres-relD}}] \)

lemma \( \text{hm-repair-op-\( \alpha \) -correct}: \text{hm-repair-op} hm i \leq_n \text{SPEC } (\lambda hm'. \text{heapmap-\( \alpha \) } hm' = \text{heapmap-\( \alpha \) } hm) \)
unfolding \( \text{hm-repair-op-def} \)
apply (refine-vcg
hm-swim-op-\( \alpha \) -correct[^{\text{THEN leof-trans}}]
hm-sink-op-\( \alpha \) -correct[^{\text{THEN leof-trans}}])
by auto

**3.10.4 Operations**

In this section, we define the operations that implement the priority-map interface
Empty

definition hm-empty-op :: ('k,'v) ahm nres
  where hm-empty-op ≡ RETURN ([],Map.empty)

lemma hm-empty-aref; (hm-empty-op,RETURN op-map-empty) ∈ (heapmap-rel)nres-rel

unfolding hm-empty-op-def
by (auto simp: heapmap-rel-defs hmr-rel-defs intro: nres-reI)

Insert

definition hm-insert-op :: ('k ⇒ 'v ⇒ ('k,'v) ahm ⇒ ('k,'v) ahm nres
  where hm-insert-op ≡ λk v h. do 
    ASSERT (h.heap-invar (hmr-α h)); 
    h ← hm-append-op h k v; 
    let l = hm-length h; 
    h ← hm-swim-op h l; 
    RETURN h
  }

lemma hm-insert-op-refine: [ heapmap-α hm k = None; (hm,h)∈hmr-rel ] =⇒
  hm-insert-op k v hm ≤ ⇓ hmr-rel (h.insert-op v h)
unfolding hm-insert-op-def h.insert-op-def
apply refine-rcg
by (auto simp: hmr-rel-def br-def)

lemma hm-insert-op-aref:
(hm-insert-op, mop-map-update-new) ∈ Id → Id → heapmap-rel → (heapmap-rel)nres-rel
apply (intro fun-reI nres-reI)
unfolding mop-map-update-new-alt
apply (rule ASSERT-refine-right)
apply (rule heapmap-nres-reI'[OF hm-insert-op-refine h.insert-op-correct])
apply (unfold heapmap-rel-def in-br-conv; clarsimp)
apply (erule heapmap-hmr-relI)
apply (unfold heapmap-rel-def in-br-conv; clarsimp)
apply (unfold heapmap-rel-def in-br-conv; clarsimp)
unfolding hm-insert-op-def
apply (refine-vcg
  hm-append-op-α-correct[THEN leaf-trans]
  hm-swim-op-α-correct[THEN leaf-trans])
apply (unfold heapmap-rel-def in-br-conv; clarsimp)
done

Is-Empty

lemma hmr-α-empty-iff[simp]:
  hmr-invar hm ⇒ hmr-α hm = [] =⇒ heapmap-α hm = Map.empty
by (auto

298
simp: hmr-α-def heapmap-invar-def heapmap-α-def hmr-invar-def
split: prod.split)

definition hm-is-empty-op :: ('k,'v) ahm ⇒ bool nres where
hm-is-empty-op ≡ λhm. do 
  ASSERT (hmr-invar hm);
  let l = hm-length hm;
  RETURN (l=0)

lemma hm-is-empty-op-refine: (hm-is-empty-op, h.is-empty-op) ∈ hmr-rel → ∨<bool-rel>nres-rel
apply (intro fun-relI nres-relI)
unfolding hm-is-empty-op-def h.is-empty-op-def
apply refine-rcq
apply (auto simp: hmr-rel-defs)
done

lemma hm-is-empty-op-aref: (hm-is-empty-op, RETURN o op-map-is-empty) ∈ heapmap-rel → ∨<bool-rel>nres-rel
apply (intro fun-relI nres-relI)
unfolding hm-is-empty-op-def
apply refine-vcg
apply simp-all
done

Lookup

definition hm-lookup-op : 'k ⇒ ('k,'v) ahm ⇒ 'v option nres
where hm-lookup-op ≡ λk hm. ASSERT (heapmap-invar hm) ⇒ RETURN (hm-lookup hm k)

lemma hm-lookup-op-aref: (hm-lookup-op,RETURN o op-map-lookup) ∈ Id → heapmap-rel → ∨<Id> nres-rel
apply (intro fun-relI nres-relI)
unfolding hm-lookup-op-def heapmap-rel-def in-br-conv
apply refine-vcg
apply simp-all
done

Contains-Key

definition hm-contains-key-op ≡ λk (pq,m). ASSERT (heapmap-invar (pq,m)) ⇒ RETURN (k∈dom m)
lemma hm-contains-key-op-aref: (hm-contains-key-op,RETURN oo op-map-contains-key) ∈ Id → heapmap-rel → ∨<bool-rel>nres-rel
apply (intro fun-relI nres-relI)
unfolding hm-contains-key-op-def heapmap-rel-defs

299
apply refine-vcg
by (auto)

Decrease-Key

definition hm-decrease-key-op ≡ λk v hm. do {
  ASSERT (heapmap-invar hm);
  ASSERT (heapmap-α hm k ≠ None ∧ prio v ≤ prio (the (heapmap-α hm k)));
  i ← hm-index-op hm k;
  hm ← hm-update-op hm i v;
  hm-swim-op hm i
}

definition (in heapstruct) decrease-key-op i v h ≡ do {
  ASSERT (valid h i ∧ prio v ≤ prio-of h i);
  h ← update-op h i v;
  swim-op h i
}

lemma (in heapstruct) decrease-key-op-invar:
[ heap-invar h; valid h i; prio v ≤ prio-of h i ] =⇒ decrease-key-op i v h ≤

SPEC heap-invar

unfolding decrease-key-op-def
apply refine-vcg
by (auto simp: swim-invar-decr)

lemma index-op-inline-refine:
assumes heapmap-invar hm
assumes heapmap-α hm k ≠ None
assumes f (hm-index hm k) ≤ m
shows do {i ← hm-index-op hm k; f i} ≤ m
using hm-index-op-correct[of hm k] assms
by (auto simp: pw-le-iff refine-pw-simps)

lemma hm-decrease-key-op-refine:
[(hm,h)∈hmr-rel; (hm,m)∈heapmap-rel; m k = Some v']
⇒ hm-decrease-key-op k v hm ≤↓hmr-rel (h.decrease-key-op (hm-index hm k) v h)

unfolding hm-decrease-key-op-def h.decrease-key-op-def

apply (refine-rcg index-op-inline-refine)
unfolding hmr-rel-def heapmap-rel-def in-br-conv
apply (clarsimp-all)
done

lemma hm-index-op-inline-leaf:
assumes f (hm-index hm k) ≤ n m
shows \( \{ i \leftarrow \text{hm-index-op } \text{hm } k; \{ f i \} \leq_n m \} \)

using \( \text{hm-index-op-correct[of } \text{hm } k \} \) assms unfolding \( \text{hm-index-op-def} \)
by (auto simp: pw-le-iff pw-leof-iff refine-pw-simps split: prod.splits)

lemma \( \text{hm-decrease-key-op-} \alpha \text{-correct} \):
\[
\text{heapmap-invar } \text{hm} \Longrightarrow \text{hm-decrease-key-op } k \text{ } v \text{ } \text{hm} \leq_n \text{SPEC } (\lambda \text{hm}'. \text{heapmap-} \alpha \text{ } \text{hm}': = \text{heapmap-} \alpha \text{ } \text{hm}((k\rightarrow v)))
\]
unfolding \( \text{hm-decrease-key-op-def} \)
apply (refine-vcg
\[
\text{hm-update-op-} \alpha \text{-correct[THEN leaf-trans]}\n\text{hm-swim-op-} \alpha \text{-correct[THEN leaf-trans]}\n\text{hm-index-op-inline-leaf}
\]
)
apply simp-all done

lemma \( \text{hm-decrease-key-op-aref} \):
\[
(\text{hm-decrease-key-op, PR-CONST } (\text{mop-pm-decrease-key } \text{prio})) \in \text{Id} \rightarrow \text{Id} \rightarrow \text{heapmap-rel} \rightarrow (\text{heapmap-rel})\text{nres-rel}
\]
unfolding \( \text{PR-CONST-def} \)
apply (intro fun-relI nres-relI)
apply (frule heapmap-hmr-relI)
unfolding \( \text{mop-pm-decrease-key-alt} \)
apply (rule ASSERT-refine-right; clarsimp)
apply (rule heapmap-nres-relI)
apply (rule \text{hm-decrease-key-op-refine}; assumption)
unfolding \( \text{heapmap-rel-def hmr-rel-def in-br-conv} \)
apply (rule \text{h.decrease-key-op-invar; simp; fail})
apply (refine-vcg \text{hm-decrease-key-op-} \alpha \text{-correct[THEN leaf-trans]; simp; fail})
done

Increase-Key

definition \( \text{hm-increase-key-op} \equiv \lambda k \text{ } v \text{ } \text{hm}. \) do {\n\[
\text{ASSERT (heapmap-invar } \text{hm});\n\text{ASSERT (heapmap-} \alpha \text{ } \text{hm } k = \text{None } \land \text{prio } v \geq \text{prio } (\text{the (heapmap-} \alpha \text{ } \text{hm } k));\n\]
\[
i \leftarrow \text{hm-index-op } \text{hm } k;\nnm \leftarrow \text{hm-update-op } \text{hm } i \text{ } v;\nnm \text{-sink-op } \text{hm } i\n\]
}
definition \( \text{(in heapstruc}t) \text{increase-key-op } i \text{ } v \text{ } h \equiv \) do {\n\[
\text{ASSERT (valid } h \text{ } i \land \text{prio } v \geq \text{prio-of } h \text{ } i);\nn \leftarrow \text{update-op } h \text{ } i \text{ } v;\nsink-op h \text{ } i\n\]
}
lemma \( \text{(in heapstruc}t) \text{increase-key-op-invar} ;\)
heap-invar h; valid h i; prio v ≥ prio-of h i] \implies increase-key-op i v h ≤

**SPEC heap-invar**

unfolding increase-key-op-def
apply refine-evg
by (auto simp; sink-invar-incr)

**lemma hm-increase-key-op-refine:**

\[(hm,h)\in hmr-rel; (hm,m)\in heapmap-rel; m k = Some v']

\implies hm\text{-increase-key-op} k v hm ≤ \u hmr-rel (h\text{-increase-key-op} (hm-index hm k) v h)

unfolding hm-increase-key-op-def h.increase-key-op-def
apply (refine-rcg index-op-inline-refine)
unfolding hmr-rel-def heapmap-rel-def in-br-conv
apply (clarsimp-all)
done

**lemma hm-increase-key-op-α-correct:**

heapmap-invar hm \implies hm\text{-increase-key-op} k v hm ≤_n SPEC (λhm'. heapmap-α hm' = heapmap-α hm(k→v))

unfolding hm-increase-key-op-def
apply (refine-evg

hm-update-op-α-correct[THEN leaf-trans]

hm-sink-op-α-correct[THEN leaf-trans]

hm-index-op-inline-leof)
apply simp-all
done

**lemma hm-increase-key-op-aref:**

(hm\text{-increase-key-op}, PR-CONST (mop-pm-increase-key prio)) ∈ Id \to Id \to
heapmap-rel \to (heapmap-rel)nres-rel

unfolding PR-CONST-def
apply (intro fun-refl nres-refl)
apply (frule heapmap-hmr-reflI)
unfolding mop-pm-increase-key-alt
apply (rule ASSERT-refine-right; clarsimp)
apply (rule heapmap-nres-refl')
apply (rule hm\text{-increase-key-op-refine}; assumption)
unfolding heapmap-rel-def hmr-rel-def in-br-conv
apply (rule h.increase-key-op-invar; simp; fail)
apply (refine-evg hm\text{-increase-key-op-α-correct}[THEN leaf-trans]; simp)
done

**Change-Key**

**definition** hm-change-key-op ≡ \lambda k v hm. do {
ASSERT (heapmap-invar hm);
ASSERT (heapmap-α hm k ≠ None);
i ← hm-index-op hm k;

302
hm ← hm-update-op hm i v;
hm-repair-op hm i
}

**definition (in heapstruct)** change-key-op i v h ≡ do {
  ASSERT (valid h i);
  h ← update-op h i v;
  repair-op h i
}

**lemma (in heapstruct)** change-key-op-invar: 

\[ [\text{heap-invar } h; \text{valid } h \ i] \implies \text{change-key-op } i \ v \ h \leq \text{SPEC heap-invar} \]

unfolding change-key-op-def
apply (refine-vcg)
apply hypsubst
apply refine-vcg
by (auto simp: sink-invar-incr)

**lemma hm-change-key-op-refine:**

\[ [(hm,h) \in hmr-rel; (hm,m) \in \text{heapmap-rel}; m \ k = \text{Some } v'] \implies hm-change-key-op k v hm \leq hmr-rel (h.change-key-op (hm-index hm k) v h) \]

unfolding hm-change-key-op-def h.change-key-op-def
apply (refine-reg index-op-inline-refine)
unfolding hr-mp-rel-def heapmap-rel-def in-br-conv
apply (clarsimp-all)
done

**lemma hm-change-key-op-α-correct:**

\[ \text{heapmap-invar } hm \implies hm-change-key-op k v hm \leq_n \text{SPEC } (\lambda hm'. \text{heapmap-α} h m' = \text{heapmap-α } hm(k \mapsto v)) \]

unfolding hm-change-key-op-def
apply (refine-vcg
hm-update-op-α-correct[THEN leof-trans]
hm-repair-op-α-correct[THEN leof-trans]
hm-index-op-inline-leaf)
unfolding heapmap-rel-def in-br-conv
apply simp
apply simp
done

**lemma hm-change-key-op-aref:**

\[ (hm-change-key-op, \text{map-map-update-ex}) \in \text{Id} \rightarrow \text{Id} \rightarrow \text{heapmap-rel} \rightarrow (\text{heapmap-rel})nres-rel \]

apply (intro fun-refl nres-refl)
apply (frule heapmap-hmr-refl)
unfolding mop-map-update-ex-alt
apply (rule ASSERT-refine-right; clarsimp)
apply (rule heapmap-nres-relI')
apply (rule hm-change-key-op-refine; assumption)
unfolding heapmap-rel-def hmr-rel-def in-br-conv
apply (rule h.change-key-op-invar; simp; fail )
apply ((refine-vcg hm-change-key-op-a-correct THEN leof-trans; simp))
done

Set

Realized as generic algorithm!

lemma (in −) op-pm-set-gen-impl: RETURN ooo op-map-update = (λk v m.
do {
c ← RETURN (op-map-contains-key k m);
if c then
  mop-map-update-ex k v m
else
  mop-map-update-new k v m
})
apply (intro ext)
unfolding op-map-contains-key-def mop-map-update-ex-def mop-map-update-new-def
by simp

definition hm-set-op k v hm ≡ do {
c ← hm-contains-key-op k hm;
if c then
  hm-change-key-op k v hm
else
  hm-insert-op k v hm
}

lemma hm-set-op-aref:
(hm-set-op, RETURN ooo op-map-update) ∈ Id → Id → heapmap-rel →
(heapmap-rel)nres-rel
unfolding op-pm-set-gen-impl
apply (intro fun-relI nres-relI)
unfolding hm-set-op-def o-def
apply (refine-reg
  hm-contains-key-op-aref[param-fo, unfolded o-def, THEN nres-relD]
  hm-change-key-op-aref[param-fo, THEN nres-relD]
  hm-insert-op-aref[param-fo, THEN nres-relD]
)
by auto

Pop-Min

definition hm-pop-min-op :: ('k,'v) ahm ⇒ (('k×'v) × ('k,'v) ahm) nres where
hm-pop-min-op hm ≡ do {
  ASSERT (heapmap-invar hm);
  ASSERT (hm-valid hm 1);
}
We demonstrate two different approaches for proving correctness here. The first approach uses the relation to plain heaps only to establish the invariant. The second approach also uses the relation to heaps to establish correctness of the result.

The first approach seems to be more robust against badly set up simpsets, which may be the case in early stages of development.

Assuming a working simpset, the second approach may be less work, and the proof may look more elegant.

**First approach** Transfer heapmin-property to heapmap-domain

```
lemma heapmap-min-prop:
  assumes INV: heapmap-invar hm
  assumes V': heapmap-α hm k = Some v'
  assumes NE: hm-valid hm (Suc 0)
  shows prio (the (heapmap-α hm (hm-key-of hm (Suc 0)))) ≤ prio v'
proof —
  — Transform into the domain of heaps
  obtain pq m where [simp]: hm=(pq,m) by (cases hm)
```
from NE have [simp]: \( pq\neq[] \) by (auto simp: hm-valid-def hm-length-def)

have CNV-LHS: \( \text{prio} \) (the (heapmap-\( \alpha \) \( hm \) (hm-key-of \( hm \) (Suc 0)))) = \( \text{prio-of} \) (hmr-\( \alpha \) \( hm \)) (Suc 0)
by (auto simp: heapmap-\( \alpha \)-def hm-key-of-def hmr-\( \alpha \)-def h.val-of-def)

from INV have INV': \( h \).heap-invar (hmr-\( \alpha \) \( hm \))
unfolding heapmap-invar-def by auto

from \( V' \) INV obtain \( i \) where IDX: \( h \).valid (hmr-\( \alpha \) \( hm \)) \( i \)
and CNV-RHS: \( \text{prio} \ v' = \text{prio-of} \) (hmr-\( \alpha \) \( hm \)) \( i \)
apply (clarsimp simp: heapmap-\( \alpha \)-def heapmap-invar-def hmr-invar-def
hmr-\( \alpha \)-def \( h \).valid-def \( h \).val-of-def)
by (metis (no-types, hide-lams) Suc-leI comp-apply diff-Suc-Suc
diff-zero domI index-less-size-conv neq0-conv nth-index nth-map
old.nat.distinct(2) option.sel)

from \( h \).heap-min-prop[OF INV' IDX] show \?thesis
unfolding CNV-LHS CNV-RHS .
qed

With the above lemma, the correctness proof is straightforward

lemma hm-pop-min-\( \alpha \)-correct: \( \text{hm-pop-min-op} \ h m \leq_n \text{SPEC} \ (\lambda((k,v),hm'). \\
\text{heapmap-\( \alpha \) \( h m \) \( k \) = Some v} \\
\land \text{heapmap-\( \alpha \) \( h m \)'} = (\text{heapmap-\( \alpha \) \( h m \)})(k:=None) \\
\land (\forall k' v'. \text{heapmap-\( \alpha \) \( h m \) \( k' \) = Some v'} \rightarrow \text{prio} v \leq \text{prio} v')\)
unfolding \( \text{hm-pop-min-op-def} \text{ heap-key-of-op-unfold \( h m \)-the-lookup-op-def} \)
apply (refine-vcg
hm-exch-op-\( \alpha \)-correct[THEN leof-trans] 
hm-butlast-op-\( \alpha \)-correct[THEN leof-trans] 
hm-sink-op-\( \alpha \)-correct[THEN leof-trans] 
)
apply (auto simp: heapmap-min-prop)
done

lemma heapmap-nres-rel-prodI:
assumes \( h m x \leq \psi(UNIV \times_r \text{hmr-rel}) \ h'x \)
assumes \( h'x \leq \text{SPEC} \ (\lambda(h'). \ h.\text{heap-invar} \ h') \)
assumes \( h m x \leq_n \text{SPEC} \ (\lambda(r,hm'). \ \text{RETURN} \ (r,\text{heapmap-\( \alpha \) \( h m \)'} \leq \psi(R\times_r \text{Id}) \ h x) \)
shows \( h m x \leq \psi(R\times_r \text{heapmap-rel}) \ h x \)
using assms
unfolding heapmap-rel-def hmr-rel-def br-def heapmap-invar-def
apply (auto simp: pw-le-iff pw-leof-iff refine-pw-simps; blast)
done

lemma hm-pop-min-op-aref: \( \text{hm-pop-min-op}, \ PR-\text{CONST} \ (\text{mop-pm-pop-min} \ n) \)
\(\text{prio}) \in \text{heapmap-rel} \rightarrow ((\text{Id} \times \text{Id}) \times \text{heapmap-rel}) \text{nres-rel}

unfolding PR-CONST-def
apply (intro fun-relI nres-relI)
apply (frule heapmap-hmr-relI)
unfolding mop-pm-pop-min-alt
apply (intro ASSERT-refine-right)
apply (rule heapmap-nres-rel-prodI)
unfolding mop-pm-pop-min-alt
apply (intro ASSERT-refine-right)
apply (rule heapmap-nres-rel-prodI)
apply (rule heapmap-nres-rel-prodI)
apply (rule hm-pop-min-op-refine[\text{param-fo}, \text{THEN nres-relD}; \text{assumption}]
unfolding heapmap-rel-def hmr-rel-def in-br-conv
apply (refine-vcg; simp)
apply (refine-vcg hm-pop-min-\alpha\text{-correct}[\text{THEN leaf-trans}]); simp split: prod.splits)
done

Second approach
definition \text{hm-kv-of-op hm i} \equiv \{\text{kvi-rel hm i} \}

definition \text{kvi-rel hm i} \equiv \{(\text{k,v}) | \text{k v. hm-key-of hm i} = \text{k}\}

lemma \text{hm-kv-op-refine[refine]:}
assumes \((\text{hm,h})\in\text{hmr-rel}\)
shows \((\text{hm-kv-of-op hm i}) \leq \nu((\text{kvi-rel hm i}) (\text{h.val-of-op h i}))
unfolding \text{hm-key-of-unfold hm-the-lookup-op-def}
apply simp
apply refine-vcg
using assms
by (auto
  simp: \text{hm-valid-def hm-length-def hmr-rel-defs heapmap-\alpha-def hm-key-of-def}
split: prod.splits)

definition \text{hm-pop-min-op'} :: (\text{'k,'v}) \text{ahm} \Rightarrow ((\text{'k x 'v}) \times (\text{'k,'v}) \text{ahm}) \text{nres}
where
\text{hm-pop-min-op'} \text{hm} \equiv \{
  \text{ASSERT (heapmap-invar hm)};
  \text{ASSERT (hm-valid hm 1)};
  \text{kv} \leftarrow \text{hm-kv-of-op hm 1};
  \text{let l} = \text{hm-length hm};;
  \text{hm} \leftarrow \text{hm-exch-op hm} \text{l l};;
  \text{hm} \leftarrow \text{hm-butlast-op hm};;
  \text{if} (\text{l} \neq 1) \text{then do} {
    \text{hm} \leftarrow \text{hm-sink-op hm} \text{l};;
    \text{RETURN (kv,hm)}
  } \text{else RETURN (kv,hm)}
\}

307
lemma hm-pop-min-op-refine':
\[ ([hm,h]) \in \text{hmr-rel} \implies \text{hm-pop-min-op'} \text{hm} \leq \downarrow (\text{kvi-rel} \text{hm} \times_r \text{hmr-rel}) \]

unfolding \text{hm-pop-min-op'-def} \text{h.pop-min-op-def}

unfolding \text{ignore-snd-refine-conv}

apply \text{refine-rcg}

unfolding \text{hmr-rel-def} \text{heapmap-rel-def}

apply (unfold \text{heapmap-invar-def}; simp add: in-br-conv)

apply (simp-all add: in-br-conv)
done

lemma heapmap-nres-rel-prodI':
assumes \text{hm}x \leq \downarrow (S \times_r \text{hmr-rel}) \text{hm}'x
assumes \text{hm}'x \leq \text{SPEC} \Phi
assumes \exists \text{hm}'r. \Phi (\text{r,hm}') \implies \text{h.heap-invar hm}'
assumes \text{hm}x \leq \land \text{SPEC} (\lambda (r,\text{hm}'). (\exists \text{r}'. (\text{r,r}') \in S \land \Phi (\text{r}',\text{hmr-\alpha hm}'))) \land
\text{hmr-invar hm}' \implies \text{RETURN} (\text{r,heapmap-\alpha hm}') \leq \downarrow (R \times_r \text{Id}) \text{hx}

shows \text{hm}x \leq \downarrow (R \times_r \text{heapmap-rel} \text{hx})

using \text{assms}

unfolding \text{heapmap-rel-def} \text{hmr-rel-def} \text{heapmap-invar-def}

apply (auto
  simp: pw-le-iff pw-leof-iff refine-pw-simps in-br-conv)

by meson

lemma ex-in-kvi-rel-conv:
(\exists \text{r}', (\text{r,r}') \in \text{kvi-rel hm} \text{i} \land \Phi \text{r}') \iff (\text{fst r} = \text{hm-key-of hm} \text{i} \land \Phi (\text{snd r}))

unfolding \text{kvi-rel-def}

apply (cases \text{r})

apply auto

done

lemma hm-pop-min-aref' :: \text{hm-pop-min-op', mop-pm-pop-min prio} \in \text{heapmap-rel}
\rightarrow (\langle Id \times_r Id \rangle \times_r \text{heapmap-rel} \text{nres-rel})

apply (intro fun-refl nres-refl)

apply (frule heapmap-hmr-refl)

unfolding mop-pm-pop-min-alt

apply (intro \text{ASSERT-refine-right})

apply (rule heapmap-nres-rel-prodI')

apply (erule \text{hm-pop-min-op-refine'})

apply (unfold heapmap-rel-def hmr-rel-def in-br-conv) []

308
apply (rule h.pop-min-op-correct)
apply simp
apply simp

apply simp

apply (clarsimp simp: ex-in-kvi-rel-conv split: prod.splits)
unfolding hm-pop-min-op'-def hm-kv-of-op-def hm-key-of-op-unfold
    hm-the-lookup-op-def
apply (refine-vcg
    hm-exch-op-α-correct[THEN leof-trans]
    hm-butlast-op-α-correct[THEN leof-trans]
)
unfolding heapmap-rel-def hmr-rel-def in-br-conv
apply (auto intro: ranI)
done

Remove

definition hm-remove-op k hm ≡ do {
  ASSERT (heapmap-invar hm);
  ASSERT (k ∈ dom (heapmap-α hm));
  i ← hm-index-op hm k;
  let l = hm-length hm;
  hm ← hm-exch-op hm i l;
  hm ← hm-butlast-op hm;
  if i ≠ l then
    hm-repair-op hm i
  else
    RETURN hm
}

definition (in heapstruct) remove-op i h ≡ do {
  ASSERT (heap-invar h);
  ASSERT (valid h i);
  let l = length h;
  h ← exch-op h i l;
  h ← butlast-op h;
  if i ≠ l then
    repair-op h i
  else
    RETURN h
}

lemma (in −) swap-empty-iff[iff]: swap l i j = [] ↔ l=[]
by (auto simp: swap-def)

lemma (in heapstruct)
butlast-exch-last: butlast (exch h i (length h)) = update (butlast h) i (last h)

unfolding exch-def update-def
apply (cases h rule: rev-cases)
apply (auto simp: swap-def butlast-list-update)
done

lemma (in heapstruct) remove-op-invar:
[[ heap-invar h; valid h i ]] \implies remove-op i h \leq SPEC heap-invar
unfolding remove-op-def
apply refine-vcg
apply (auto simp: valid-def) []
apply (auto simp: valid-def exch-def) []
apply (simp add: butlast-exch-last)
apply refine-vcg
apply auto []
apply auto []
apply (auto simp: valid-def) []
apply auto []
apply auto []
done

lemma hm-remove-op-refine[refine]:
[[ (hm,m)\in heapmap-rel; (hm,h)\in hmr-rel; heapmap-\alpha hm k \neq None ]] \implies hm-remove-op k hm \leq \downarrow hmr-rel (hm.remove-op (hm-index hm k) h)
unfolding hm-remove-op-def h.remove-op-def heapmap-rel-def
apply (refine-reg index-op-inline-refine)
unfolding hmr-rel-def
apply (auto simp: in-br-conv)
done

lemma hm-remove-op-\alpha-correct:
hm-remove-op k hm \leq_n SPEC (\lambda hm'. heapmap-\alpha hm' = (heapmap-\alpha hm)(k:=None))

unfolding hm-remove-op-def
apply (refine-vcg
  hm-exch-op-\alpha-correct[THEN leof-trans]
  hm-butlast-op-\alpha-correct[THEN leof-trans]
  hm-repair-op-\alpha-correct[THEN leof-trans]
  hm-index-op-inline-leof
)
apply (auto; fail)

apply clarsimp
apply (rewrite at hm-index - k = hm-length - in asm eq-commute)
apply (auto; fail)
done

lemma hm-remove-op-aref:

310
(hm-remove-op, mop-map-delete-ex) ∈ Id → heapmap-rel → (heapmap-rel) nres-rel
apply (intro fun-relI nres-relI)
unfolding mop-map-delete-ex-alt
apply (rule ASSERT-refine-right)
apply (rule heapmap-hmr-relI)
apply (rule heapmap-nres-relI ’)
also apply (rule hm-remove-op-refine; assumption?)
apply (unfold heapmap-rel-def in-br-conv; auto)

unfolding heapmap-rel-def hmr-rel-def in-br-conv
apply (refine-vec h.remove-op-invar; clarsimp; fail)
apply (refine-vec hm-remove-op-a-correct[THEN leof-trans]; simp; fail)
done

Peek-Min

definition hm-peek-min-op :: (’k,’v) ahm ⇒ (’k×’v) nres where
hm-peek-min-op hm ≡ hm-ke-of-op hm 1

lemma hm-peek-min-op-aref:
(hm-peek-min-op, PR-CONST (mop-pm-peek-min prio)) ∈ heapmap-rel →
(Id ×, Id) nres-rel
unfolding PR-CONST-def
apply (intro fun-relI nres-relI)
proof –
fix hm and m :: ’k ⇒ ’v
assume A: (hm,m)∈heapmap-rel

from A have [simp]: h.heap-invar (hmr-α hm) hmr-invar hm m=heapmap-α
hm
unfolding heapmap-rel-def in-br-conv heapmap-invar-def
by simp-all

have hm-peek-min-op hm ≤ ↓ (kvi-rel hm 1) (h.peek-min-op (hmr-α hm))
unfolding hm-peek-min-op-def h.peek-min-op-def
apply (refine-reg hm-kv-op-refine)
using A
apply (simp add: heapmap-hmr-relI)
done
also have [hmr-α hm ≠ [[]] ⇒ (h.peek-min-op (hmr-α hm))
≤ SPEC (λv. v∈ran (heapmap-α hm) ∧ (∀ v’∈ran (heapmap-α hm). prio v
≤ prio v ’))
apply refine-vec
by simp-all
finally show hm-peek-min-op hm ≤ ↓ (Id ×, Id) (mop-pm-peek-min prio m)

unfolding mop-pm-peek-min-alt
apply (simp add: pw-le-iff refine-pw-simps hm-peek-min-op-def hm-ke-of-op-def

311
theory IICF-Array
imports ../Intf/IICF-List
begin

Lists of fixed length are directly implemented with arrays.

definition is-array l p ≡ p ↦→ a l

lemma is-array-precise[safe-constraint-rules]: precise is-array
  apply rule
  unfolding is-array-def
  apply prec-extract-eqs
  by simp

definition array-assn where array-assn A ≡ hr-comp is-array (⟨the-pure A⟩ list-rel)
lemmas [safe-constraint-rules] = CN-FALSEI[of is-pure array-assn A for A]

definition [simp,code-unfold]: heap-array-empty ≡ Array.of-list []
definition [simp,code-unfold]: heap-array-set p i v ≡ Array.upd i v p

context
notes [fcomp-norm-unfold] = array-assn-def[symmetric]
notes [intro] = hfrefl hn-refineI[THEN hn-refine-preI]
notes [simp] = pure-def hn-ctxt-def is-array-def invalid-assn-def
begin

lemma array-empty-hnr-aux: (uncurry0 heap-array-empty,uncurry0 (RETURN op-list-empty)) ∈ unit-assn k →_a is-array
  by sep-auto

lemma array-replicate-hnr-aux:
  (uncurry Array.new, uncurry (RETURN oo op-list-replicate))
  ∈ nat-assn k *_a id-assn k →_a is-array
  by (sep-auto)

3.11 Plain Arrays Implementing List Interface
definition \[\text{simp}]: \text{op-array-replicate} \equiv \text{op-list-replicate} \\
sepref-register \text{op-array-replicate}

lemma \text{array-fold-custom-\text{-}replicate}:
  \text{replicate} = \text{op-array-replicate}
  \text{op-list-replicate} = \text{op-array-replicate}
  \text{mop-list-replicate} = \text{RETURN} o o \text{op-array-replicate}
  \text{by (auto simp: op-array-replicate-def intro!: ext)}

lemmas \text{array-replicate-\text{-}custom-\text{-}hnr[sepref-fr-rules]} = \text{array-replicate-\text{-}hnr[unfolded array-fold-\text{-}custom-\text{-}replicate]}

lemma \text{array-of-list-\text{-}hnr-aux}: (\text{Array.of-list,RETURN o op-list-copy}) \in (\text{list-assn id-assn})^k \rightarrow_a \text{is-array}
  unfolding \text{list-assn-pure-conv}
  \text{by (sep-auto)}

sepref-decl-impl \text{(no-register)} \text{array-of-list: array-of-list-\text{-}hnr-aux}.

definition \[\text{simp}]: \text{op-array-of-list} \equiv \text{op-list-copy} \\
sepref-register \text{op-array-of-list}

lemma \text{array-fold-custom-of-list}:
  \text{l = op-array-of-list l}
  \text{op-list-copy} = \text{op-array-of-list}
  \text{mop-list-copy} = \text{RETURN o op-array-of-list}
  \text{by (auto intro!: ext)}

lemmas \text{array-of-list-\text{-}custom-\text{-}hnr[sepref-fr-rules]} = \text{array-of-list-\text{-}hnr[folded op-array-of-list-def]}

lemma \text{array-copy-\text{-}hnr-aux}: (\text{array-copy,RETURN o op-list-copy}) \in \text{is-array}^k
  \rightarrow_a \text{is-array}
  \text{by sep-auto}

sepref-decl-impl \text{array-copy: array-copy-\text{-}hnr-aux}.

lemma \text{array-get-\text{-}hnr-aux}: (\text{uncurry Array.nth,uncurry (RETURN oo op-list-get)}) \in [\lambda (l,i). i<\text{length} l]_a \text{is-array}^k *_a \text{nat-assn}^k \rightarrow a \text{id-assn}
  \text{by sep-auto}

sepref-decl-impl \text{array-get: array-get-\text{-}hnr-aux}.

lemma \text{array-get-\text{-}hnr-aux}: (\text{uncurry2 heap-array-set,uncurry2 (RETURN ooo op-list-set)}) \in [\lambda (l,i). i<\text{length} l]_a \text{is-array}^d *_a \text{nat-assn}^k *_a \text{id-assn}^k \rightarrow \text{is-array}
  \text{by sep-auto}

sepref-decl-impl \text{array-set: array-set-\text{-}hnr-aux}.

lemma \text{array-length-\text{-}hnr-aux}: (\text{Array.len,RETURN o op-list-length}) \in \text{is-array}^k
  \rightarrow_a \text{nat-assn}
  \text{by sep-auto}

sepref-decl-impl \text{array-length: array-length-\text{-}hnr-aux}.
end

definition [simp]: op-array-empty ≡ op-list-empty
interpretation array: list-custom-empty array-assn A heap-array-empty op-array-empty
apply unfold-locales
apply (rule array-empty-hnr[simplified pre-list-empty-def])
by (auto)

end
theory IICF-MS-Array-List
imports
  ../Intf/IICF-List
  Separation-Logic-Imperative-HOL.Array-Blit
  ../../../Separation-Logic-Imperative-HOL/Examples/Default-Insts
begin
  type-synonym 'a ms-array-list = 'a Heap.array × nat

definition is-ms-array-list ms l ≡ λ(a,n). ∃ A l'. a ↦→ A l' ⊨ (n ≤ length l' ∧ l = take n l' ∧ ms=length l')

lemma is-ms-array-list-prec[safe-constraint-rules]: precise (is-ms-array-list ms)
  unfolding is-ms-array-list-def[abs-def]
  apply(rule preciseI)
  apply(simp split: prod.splits)
  using preciseD snga-prec by fastforce

definition marl-empty-sz maxsize ≡ do
  a ← Array.new maxsize default;
  return (a,0) 

definition marl-append ≡ λ(a,n) x. do
  a ← Array.upd n x a;
  return (a,n+1) 

definition marl-length :: 'a::heap ms-array-list ⇒ nat Heap where
  marl-length ≡ λ(a,n). return (n)

definition marl-is-empty :: 'a::heap ms-array-list ⇒ bool Heap where
  marl-is-empty ≡ λ(a,n). return (n=0)

definition marl-last :: 'a::heap ms-array-list ⇒ 'a Heap where
  marl-last ≡ λ(a,n). do 
    Array.nth a (n - 1) 

definition marl-butlast :: 'a::heap ms-array-list ⇒ 'a ms-array-list Heap
where
  marl-butlast ≡ λ(a,n). do {
    return (a,n - 1)
  }

definition marl-get :: 'a::heap ms-array-list ⇒ nat ⇒ 'a Heap
where
  marl-get ≡ λ(a,n) i. Array.nth a i

definition marl-set :: 'a::heap ms-array-list ⇒ nat ⇒ 'a ⇒ 'a ms-array-list Heap
where
  marl-set ≡ λ(a,n) i x. do { a ← Array.update i x a; return (a,n)}

lemma marl-empty-sz-rule [sep-heap-rules]: < emp > marl-empty-sz N <is-ms-array-list N []>
  by (sep-auto simp: marl-empty-sz-def is-ms-array-list-def)

lemma marl-append-rule [sep-heap-rules]: length l < N ⇒
  < is-ms-array-list N l a >
  marl-append a x
  <λa. is-ms-array-list N (l@[x]) a >
  by (sep-auto
      simp: marl-append-def is-ms-array-list-def take-update-last
      split: prod.splits)

lemma marl-length-rule [sep-heap-rules]:
  < is-ms-array-list N l a >
  marl-length a
  <λr. is-ms-array-list N l a * (r=length l)> 
  by (sep-auto simp: marl-length-def is-ms-array-list-def)

lemma marl-is-empty-rule [sep-heap-rules]:
  < is-ms-array-list N l a >
  marl-is-empty a
  <λr. is-ms-array-list N l a * (r⟷(l=[]))>
  by (sep-auto simp: marl-is-empty-def is-ms-array-list-def)

lemma marl-last-rule [sep-heap-rules]:
  l≠[] ⇒
  < is-ms-array-list N l a >
  marl-last a
  <λr. is-ms-array-list N l a * (r⟷(last l))>
  by (sep-auto simp: marl-last-def is-ms-array-list-def last-take-nth-conv)

lemma marl-butlast-rule [sep-heap-rules]:
  l≠[] ⇒
  < is-ms-array-list N (butlast l)>
  marl-butlast a
  < is-ms-array-list N (butlast l)>
by (sep-auto
  split: prod.splits
  simp: marl-butlast-def is-ms-array-list-def butlast-take)

lemma marl-get-rule [sep-heap-rules]:
  \( i < \text{length } l \Rightarrow \langle \text{is-ms-array-list } N \text{ l a} \rangle \)
  \(<\lambda r. \text{is-ms-array-list } N \text{ l a} \ast \hat{(r = \text{!i})}>\)
by (sep-auto simp: marl-get-def is-ms-array-list-def split: prod.split)

lemma marl-set-rule [sep-heap-rules]:
  \( i < \text{length } l \Rightarrow \langle \text{is-ms-array-list } N \text{ l a} \rangle \)
  \(<\text{marl-set a i x} \ast \text{is-ms-array-list } N \text{ (l[i:=x])}>\)
by (sep-auto simp: marl-set-def is-ms-array-list-def split: prod.split)

definition marl-assn N A \equiv hr-comp (is-ms-array-list N) ((the-pure A) list-rel)

context
  notes [fcomp-norm-unfold] = marl-assn-def[symmetric]
  notes [intro!] = hffH hn-refineI[THEN hn-refine-preI]
  notes [simp] = pure-def hn-ctxt-def invalid-assn-def
begin

definition [simp]: op-marl-empty-sz (N::nat) \equiv op-list-empty
context fixes N :: nat begin
  sepr pref-register PR-CONST (op-marl-empty-sz N)
end

lemma [def-pat-rules]: op-marl-empty-sz$N \equiv UNPROTECT (op-marl-empty-sz N) by simp

lemma marl-fold-custom-empty-sz:
  op-list-empty = op-marl-empty-sz N
  mop-list-empty = RETURN (op-marl-empty-sz N)
  [] = op-marl-empty-sz N
by auto

lemma marl-empty-hnr-aux: (uncurry0 (marl-empty-sz N), uncurry0 (RETURN op-list-empty)) \(\in\) unit-assn\(\rightarrow\) is-ms-array-list N
by sep-auto
lemmas marl-empty-hnr = marl-empty-hnr-aux[FCOMP op-list-empty.fref[of the-pure A for A]]
lemmas marl-empty-hnr-mop = marl-empty-hnr[FCOMP mop-list-empty.alt[OF mop-list-empty-alt]]
lemma marl-empty-sz-hnr\[sepref-fr-rules\]:
(uncurry0 (marl-empty-sz N), uncurry0 (RETURN (PR-CONST (op-marl-empty-sz N)))) \in\ unit-assn \ k \rightarrow\ a \ marl-assn \ N \ A

using marl-empty-hnr
by simp

lemma marl-append-hnr-aux: (uncurry marl-append, uncurry (RETURN oo op-list-append)) \in\ [\lambda \langle l, \cdot \rangle . \ \mathrm{length} \ l < N]_a \ ((\mathrm{is-ms-array-list} \ N)^d \ \ast_a \ \mathrm{id-assn}^k) \rightarrow\ \mathrm{is-ms-array-list} \ N

by sep-auto
lemmas marl-append-hnr[sepref-fr-rules] = marl-append-hnr-aux[FCOMP op-list-append.fref]

lemmas marl-append-hnr-mop[sepref-fr-rules] = marl-append-hnr[FCOMP mk-mop-rl2-np(OF mop-list-append-alt)]

lemma marl-length-hnr-aux: (marl-length, RETURN o op-list-length) \in\ (\mathrm{is-ms-array-list} \ N)^k \rightarrow\ a \ \mathrm{nat-assn}

by sep-auto
lemmas marl-length-hnr[sepref-fr-rules] = marl-length-hnr-aux[FCOMP op-list-length.fref[of the-pure A for A]]

lemmas marl-length-hnr-mop[sepref-fr-rules] = marl-length-hnr[FCOMP mk-mop-rl1-np(OF mop-list-length-alt)]

lemma marl-is-empty-hnr-aux: (marl-is-empty, RETURN o op-list-is-empty) \in\ (\mathrm{is-ms-array-list} \ N)^k \rightarrow\ a \ \mathrm{bool-assn}

by sep-auto
lemmas marl-is-empty-hnr[sepref-fr-rules] = marl-is-empty-hnr-aux[FCOMP op-list-is-empty.fref[of the-pure A for A]]

lemmas marl-is-empty-hnr-mop[sepref-fr-rules] = marl-is-empty-hnr[FCOMP mk-mop-rl1-np(OF mop-list-is-empty-alt)]

lemma marl-last-hnr-aux: (marl-last, RETURN o op-list-last) \in\ [\lambda x . \ x \neq \ ]_a (\mathrm{is-ms-array-list} \ N)^k \rightarrow\ a \ \mathrm{id-assn}

by sep-auto
lemmas marl-last-hnr[sepref-fr-rules] = marl-last-hnr-aux[FCOMP op-list-last.fref]

lemmas marl-last-hnr-mop[sepref-fr-rules] = marl-last-hnr[FCOMP mk-mop-rl1(OF mop-list-last-alt)]

lemma marl-butlast-hnr-aux: (marl-butlast, RETURN o op-list-butlast) \in\ [\lambda x. \ x \neq \ ]_a (\mathrm{is-ms-array-list} \ N)^d \rightarrow\ (\mathrm{is-ms-array-list} \ N)

by sep-auto
lemmas marl-butlast-hnr[sepref-fr-rules] = marl-butlast-hnr-aux[FCOMP op-list-butlast.fref[of the-pure A for A]]

lemmas marl-butlast-hnr-mop[sepref-fr-rules] = marl-butlast-hnr[FCOMP mk-mop-rl1(OF mop-list-butlast-alt)]

lemma marl-get-hnr-aux: (uncurry marl-get, uncurry (RETURN oo op-list-get)) \in\ [\lambda \langle l, \cdot \rangle . \ \mathrm{length} \ l < N]_a \ ((\mathrm{is-ms-array-list} \ N)^k \ \ast_a \ \mathrm{nat-assn}^k) \rightarrow\ \mathrm{id-assn}

by sep-auto

lemma marl-set-hnr-aux: (uncurry2 marl-set, uncurry2 (RETURN ooo op-list-set)) ∈ [λ((l,i),). i<length l] a ((is-ms-array-list N) * a nat-assn * a id-assn) → (is-ms-array-list N)
  by sep-auto

end

context
  fixes N :: nat
  assumes N-sz: N>10
begin

schematic-goal hn-refine (emp) (?c::?c c Heap) ?Γ' ?R (do {
  let x = op-marl-empty-sz N;
  RETURN (x@[1::nat])
})
  using N-sz
  by sepref

end

schematic-goal hn-refine (emp) (?c::?c c Heap) ?Γ' ?R (do {
  let x = op-list-empty;
  RETURN (x@[1::nat])
})
  apply (subst marl-fold-custom-empty-sz[where N=10])
  apply sepref
  done

end

theory IICF-Indexed-Array-List
imports
  HOL-Library.Rewrite
  .. /Intf /IICF-List
  List-Index.List-Index
  IICF-Array
  IICF-MS-Array-List
begin

We implement distinct lists of natural numbers in the range {0..<N} by a length counter and two arrays of size N. The first array stores the list, and the second array stores the positions of the elements in the list, or N if the element is not in the list.

318
This allows for an efficient index query. The implementation is done in two steps: First, we use a list and a fixed size list for the index mapping. Second, we refine the lists to arrays.

**type-synonym**  
\[
\text{type-synonym aial = nat list } \times \text{nat list}
\]

**locale**  
\[
\text{locale iai-invar = fixes maxsize :: nat and } l :: \text{nat list and } qp :: \text{nat list}
\]

**assumes**  
- \text{maxsize-eq[simp]: maxsize = length qp}
- \text{l-distinct[simp]: distinct l}
- \text{l-set: set l } \subseteq \{0..<\text{length qp}\}
- \text{qp-def: } \forall k <\text{length qp. qp!k = (if } k \in\text{set l then List-Index.index l k else length qp)}

**begin**

**lemma**  
\[
\text{l-len: length l } \leq \text{length qp}
\]

**proof**
- from \text{card-mono[OF - l-set]} have \text{card (set l) } \leq \text{length qp} by \text{auto}
  - with \text{distinct-card[OF l-distinct]} show \text{?thesis by simp}

**qed**

**lemma**  
\[
\text{idx-len[simp]: i<length l } \Longrightarrow \forall i < \text{length qp}
\]

**using** \text{l-set}
- \text{by (metis atLeastLessThan-iff nth-mem psubsetD psubsetI)}

**lemma**  
\[
\text{l-set-simp[simp]: k } \in \text{set l } \Longrightarrow k < \text{length qp}
\]

**by** \text{(auto dest: set-mp[OF l-set])}

**lemma**  
\[
\text{qpk-idx: k<length qp } \Longrightarrow \text{qp!k < length l } \iff k \in \text{set l}
\]

**proof** \text{(rule iffI)}
- assume A: k < length qp
  - \text{assume qp!k < length l}
- hence qp!k < length qp using l-len by simp
  - with spec[OF qp-def, of k] A show k \in set l
    - by (auto split: if-split-asm)
  
- \text{assume k } \in \text{set l}
- thus qp!k < length l
  - using qp-def by (auto split: if-split-asm) []

**qed**

**lemma**  
\[
\text{lqpk[simp]: k } \in \text{set } l \Longrightarrow l ! (qp!k) = k
\]

**using** \text{spec[OF qp-def, of k]} by \text{auto}

**lemma**  
\[
[i<\text{length } l; j<\text{length } l; l!i=l!j] \Longrightarrow i=j
\]

319
by (simp add: nth-eq-iff-index-eq)

lemmas index-swap[simp] = index-swap-if-distinct[folded swap-def, OF l-distinct]

lemma swap-invar:
  assumes i<length l j<length l
  shows ial-invar (length qp) (swap l i j) (qp[l ! j := i, l ! i := j])
  using assms
  apply (unfold-locale)
  apply auto []
  apply auto []
  apply auto []
  apply (auto simp: simp: nth-list-update nth-eq-iff-index-eq index-nth-id) []
  using qp-def apply auto [2]
  done

end

definition ial-rel1 maxsize ≡ br fst (uncurry (ial-invar maxsize))

definition ial-assn2 :: nat ⇒ nat list * nat list ⇒ - where
  ial-assn2 maxsize ≡ prod-assn (marl-assn maxsize nat-assn) (array-assn nat-assn)

definition ial-assn maxsize A ≡ hr-comp (hr-comp (ial-assn2 maxsize) (ial-rel1 maxsize)) ((the-pure A)list-rel)
  lemmas [safe-constraint-rules] = CN-FALSEI[of is-pure ial-assn maxsize A for maxsize A]

3.11.1 Empty

definition op-ial-empty-sz :: nat ⇒ 'a list
  where [simp]: op-ial-empty-sz ms ≡ op-list-empty

lemma [def-pat-rules]: op-ial-empty-sz$maxsize ≡ UNPROTECT (op-ial-empty-sz maxsize)
  by simp

context fixes maxsize :: nat begin
sepref-register PR-CONST (op-ial-empty-sz maxsize)
end

context fixes maxsize :: nat
notes [fcomp-norm-unfold = ial-assn-def[symmetric]]
notes [simp] = hn-ctxt-def pure-def
begin
definition aial-empty \equiv do 
   let l = op-marl-empty-sz maxsize; 
   let qp = op-array-replicate maxsize maxsize; 
   RETURN (l,qp) 
}

lemma aial-empty-impl: (aial-empty,RETURN op-list-empty) \in (ial-rel1 max-size)\text{nres-rel}
  unfolding aial-empty-def
  apply (refine-vcg nres-rell)
  apply (clarsimp simp: ial-rel1-def br-def)
  apply unfold-locales
  apply auto
  done

context
   notes [id-rules] = itypeI[Pure.of maxsize TYPE(nat)]
   notes [sepref-import-param] = IdI[of maxsize]
begin
   sepref-definition aial-empty is uncurry0 aial-empty :: unit-assn \(k \rightarrow_a ial-assn2 maxsize\)
  unfolding aial-empty-def ial-assn2-def
  using [[id-debug]]
  by sepref
end

sepref-decl-impl (no-register) aial-empty: aial-empty.refine[FCOMP aial-empty-impl]
.

lemma aial-empty-sz-hnr[sepref-fr-rules]:
  (uncurry0 local.ial-empty, uncurry0 (RETURN (PR-CONST (op-ial-empty-sz maxsize)))) \in unit-assn\(k \rightarrow_a ial-assn\ maxsize\ A\)
  using aial-empty-hnr[of A] by simp

3.11.2 Swap

definition aial-swap \equiv \lambda(l,qp) i j. do 
   vi \leftarrow mop-list-get l i;
   vj \leftarrow mop-list-get l j;
   l \leftarrow mop-list-set l i vi;
   l \leftarrow mop-list-set l j vj;
   qp \leftarrow mop-list-set qp vj i;
   qp \leftarrow mop-list-set qp vi j;
   RETURN (l,qp) 
}

lemma in-ial-rel1-conv:
  ((pq, qp), l) \in ial-rel1 ms \leftrightarrow pq=l \land ial-invar ms l q
by (auto simp: ial-rel1-def in-br-conv)

lemma aial-swap-impl:
  (aial-swap, mop-list-swap) ∈ ial-rel1 maxsize → nat-rel → nat-rel → ial-rel1 maxsize nres-rel
proof (intro fun-relI nres-relI; clarsimp simp: in-ial-rel1-conv; refine-vcg; clarsimp)
  fix l qp i j
  assume [simp]: i < length l j < length l and ial-invar maxsize l qp
  then interpret ial-invar maxsize l qp by simp
  show aial-swap (l, qp) i j ≤ SPEC (λc. (c, swap l i j) ∈ ial-rel1 maxsize)
    unfolding aial-swap-def
    apply refine-vcg
    apply (vc-solve simp add: in-ial-rel1-conv swap-def [symmetric] swap-invar)
done
qed

sepref-definition aial-swap is
  uncurry2 aial-swap :: (ial-assn2 maxsize) * nat-assn k * nat-assn k → ial-assn2 maxsize
  unfolding aial-swap-def aial-assn2-def
  by sepref

sepref-decl-impl (ismop) test: aial-swap.refine[FCOMP aial-swap-impl]
  uses mop-list-swap.fref .

3.11.3 Length

definition aial-length :: aial ⇒ nat nres
  where aial-length ≡ λ(l, -). RETURN (op-list-length l)

lemma aial-length-impl: (aial-length, mop-list-length) ∈ ial-rel1 maxsize → (nat-rel) nres-rel
  apply (intro fun-relI nres-relI)
  unfolding ial-rel1-def in-br-conv aial-length-def
  by auto

sepref-definition aial-length is aial-length :: (ial-assn2 maxsize) k → nat-assn
  unfolding aial-length-def aial-assn2-def
  by sepref

sepref-decl-impl (ismop) aial-length: aial-length.refine[FCOMP aial-length-impl]
.

3.11.4 Index

definition aial-index :: aial ⇒ nat ⇒ nat nres where
  aial-index ≡ λ(l, qp) k. do { ASSERT k∈set l;
\[ i \leftarrow \text{mop-list-get}\ qp\ k; \]
\[ \text{RETURN } i \]
\]

\textbf{lemma} \texttt{aial-index-impl}:
\[(\text{uncurry aial-index, uncurry mop-list-index}) \in \]
\[\lambda(\text{l},k). \ \text{k} \in \text{set } \text{l} \rightarrow (\text{ial-rel1 maxsize } \times, \text{nat-rel } \rightarrow (\text{nat-rel})\text{nres-rel})\]
\textbf{apply} (intro fun-refl nres-refl frefl)
\textbf{unfolding} \texttt{ial-rel1-def}
\textbf{proof} (clarsimp simp: in-br-conv)
\textbf{fix} \text{l} \text{qp} \text{k}
\textbf{assume} \text{ial-invar maxsize l qp}
\textbf{then interpret} \text{ial-invar maxsize l qp}.
\textbf{assume} \text{k} \in \text{set } \text{l}
\textbf{then show} \text{ial-index } (\text{l},\text{qp}) \text{k} \leq \text{RETURN } (\text{index l } \text{k})
\textbf{unfolding} \texttt{ial-index-def}
\textbf{apply} (refine-vcg)
\textbf{by} (auto simp: qp-def)
\textbf{qed}

\textbf{sepref-definition} \texttt{ial-index} \textbf{is} \texttt{uncurry aial-index} :: \texttt{(ial-assn2 maxsize)}^k \ast_{a} \texttt{nat-assn}^k \rightarrow_{a} \texttt{nat-assn}
\textbf{unfolding} \texttt{ial-index-def} \texttt{ial-assn2-def}
\textbf{by} \texttt{sepref}

\textbf{sepref-decl-impl} (ismop) \texttt{ial-index} :: \texttt{ial-index.refine[FCOMP aial-index-impl]}

3.11.5 Butlast

\textbf{definition} \texttt{aial-butlast} :: \texttt{aial} \Rightarrow \texttt{aial nres where}
\texttt{aial-butlast} \equiv \lambda(\text{l},\text{qp}) . \texttt{do} \{\}
\textbf{ASSERT} (\text{l} \neq []) ;
\texttt{len} \leftarrow \text{mop-list-length l} ;
\texttt{k} \leftarrow \text{mop-list-get}\ l\ (\text{len} - 1) ;
\texttt{l} \leftarrow \text{mop-list-butlast}\ l ;
\texttt{qp} \leftarrow \text{mop-list-set}\ \text{qp} \text{k} (\text{length qp}) ;
\text{RETURN} (\text{l},\text{qp})\}

\textbf{lemma} \texttt{aial-butlast-refine}: \texttt{(aial-butlast, mop-list-butlast) \in ial-rel1 maxsize} \rightarrow \texttt{(ial-rel1 maxsize)\text{nres-rel}}
\textbf{apply} (intro fun-refl nres-refl)
\textbf{unfolding} \texttt{ial-rel1-def}
\textbf{proof} (clarsimp simp: in-br-conv simp del: mop-list-butlast-alt)
\textbf{fix} \text{l} \text{qp}
\textbf{assume} \text{ial-invar maxsize l qp}
\textbf{then interpret} \text{ial-invar maxsize l qp} .

323
{ assume A: "l≠[]
  have "ial-invar (length qp) (butlast l) (qp[l ! (length l – Suc 0) := length qp])
  apply standard
  apply clarsimp-all
  apply (auto simp: distinct-butlast []
    using l-set apply (auto dest: in-set-butlastD [])
    using qp-def A l-distinct
  apply (auto simp: nth-list-update neq-Nil-rev-conv index-append simp del: l-distinct)
  done
} note aux1 = this

show "ial-butlast (l, qp) ≤⇓ (br fst (uncurry (ial-invar maxsize))) (mop-list-butlast l)"
  unfolding aial-butlast-def mop-list-butlast-alt
  apply refine-vcg
  apply (clarsimp simp: in-br-conv aux1)
  done
qed

sepref-definition aial-butlast is aial-butlast :: (ial-assn2 maxsize) → aial-assn2 maxsize
  unfolding aial-butlast-def ial-assn2-def by sepref


3.11.6 Append

definition aial-append :: aial ⇒ nat ⇒ aial nres where
aial-append ≡ λ(l,qp) k. do { Assert (k<length qp ∧ k∉ set l ∧ length l < length qp);
  len ← mop-list-length l;
  l ← mop-list-append l k;
  qp ← mop-list-set qp k len;
  RETURN (l,qp) }

lemma aial-append-refine:
  (uncurry aial-append, uncurry mop-list-append) ∈
  [Λ(l,k). k<maxsize ∧ k∉ set l]f ial-rel1 maxsize ×, nat-rel → (ial-rel1 maxsize)nres-rel
  apply (intro frefl nres-relI)
  unfolding ial-rel1-def
  proof (clarsimp simp: in-br-conv)
  fix l qp k

324
assume KLM: $k < \text{maxsize}$ and KNL: $k \notin \text{set } l$
assume ial-invar maxsize $l \; qp$
then interpret ial-invar maxsize $l \; qp$.

from KLM have KLL: $k < \text{length } qp$ by simp

note distinct-card[OF l-distinct, symmetric]
also from KNL l-set have set $l \subseteq \{0..<k\} \cup \{\text{Suc } k..<\text{length } qp\}$
by (auto simp: nat-less-le)
from card-mono[OF - this] have card (set $l$) $\leq$ card ...
by simp
also note card-Un-le
also have card $\{0..<k\} +$ card $\{\text{Suc } k..<\text{length } qp\} = k + (\text{length } qp - \text{Suc } k)$
by simp
also have ... $<$ length $qp$ using KLL by simp
finally have LLEN: length $l < \text{length } qp$.

have aux1[simp]: ial-invar (length $qp$) ($l @ [k]$) ($qp[k := \text{length } l]$)
apply standard
apply (clarsimp simp: KLL)
using KLL apply (auto simp: Suc-le-eq LLEN)
apply (simp add: $qp$-def)
apply (simp add: $qp$-def)
done

show ial-append ($l$, $qp$) $k \leq \downarrow (\text{br } \text{fst} (\text{uncurry (ial-invar maxsize))))$ (RETURN ($l@[k]$))
unfolding ial-append-def mop-list-append-def
apply refine-vcg
apply (clarsimp simp: in-br-conv KLL KNL LLEN)
done
qed

private lemma ial-append-impl-aux: $((l, qp), l') \in \text{ial-rel1 maxsize} \Longrightarrow l'=l$
\begin{align*}
a & \land \text{maxsize} = \text{length } qp\\
& \text{unfolding ial-rel1-def}\\
& \text{by (clarsimp simp: in-br-conv ial-invar.maxsize-eq[ symmetric])}
\end{align*}

context
notes [dest!] = ial-append-impl-aux
begin

sepref-definition ial-append is
\begin{align*}
\text{uncurry ial-append } & : \lambda (\text{lqp}.-). \text{lqp} \in \text{Domain (ial-rel1 maxsize)} |_a (\text{ial-assn2} \\
& \text{maxsize})^a \cdot \text{nat-assn}^b \rightarrow \text{ial-assn2 maxsize} \\
& \text{unfolding ial-append-def ial-assn2-def} \\
& \text{by sepref}
\end{align*}
lemma \((\lambda b. b < \text{maxsize}, X) \in A \rightarrow \text{bool-rel}\)
apply auto
oops

context begin

private lemma append-fref': \([\text{IS-BELOW-ID } R]\) 
\(\Rightarrow (\text{uncurry mop-list-append, uncurry mop-list-append}) \in \langle R \rangle \text{list-rel} \times_{r} R\) 
\(\rightarrow_f (\langle R \rangle \text{list-rel}) \text{nres-rel}\)
by (rule mop-list-append.fref)

sepref-decl-impl (ismop) ial-append: ial-append.refine[FCOMP aial-append-refine]
uses append-fref'
unfolding IS-BELOW-ID-def
apply (parametricity; auto simp: single-valued-below-Id)
done
end

3.11.7 Get

definition aial-get :: aial \Rightarrow nat \Rightarrow nat nres where
aial-get \equiv \lambda (l,qp) i. mop-list-get l i

lemma aial-get-refine: (aial-get,mop-list-get) \in ial-rel1 maxsize \rightarrow nat-rel \rightarrow (nat-rel) nres-rel
apply (intro fun-rell nres-rell)
unfolding aial-get-def ial-rel1-def mop-list-get-def in-br-conv
apply refine-vcg
apply clarsimp-all
done

sepref-definition ial-get is uncurry aial-get :: (ial-assn2 maxsize)^k \ast_a nat-assn^k
\rightarrow_a nat-assn
unfolding aial-get-def ial-assn2-def by sepref


3.11.8 Contains

definition aial-contains :: nat \Rightarrow aial \Rightarrow bool nres where
aial-contains \equiv \lambda k (l,qp). do {
if k<\text{maxsize} then do {
i \leftarrow \text{mop-list-get qp} k;
RETURN (i<\text{maxsize})
} else RETURN False
}
lemma aial-contains-refine: (uncurry aial-contains, uncurry mop-list-contains) ∈ (nat-rel × ial-rel maxsize) → f (bool-rel)nres-rel
apply (intro frefI nres-rell)
unfolding ial-rell-def
proof (clarsimp simp: in-br-conv)
  fix l qp k

  assume ial-invar maxsize l qp
  then interpret ial-invar maxsize l qp.

  show aial-contains k (l, qp) ≤ RETURN (k∈set l)
    unfolding aial-contains-def
    apply refine-vcg
    by (auto simp: l-len qp-def split: if-split_asm)
qed

context
notes [id-rules] = itypeI[Type.of maxsize TYPE(nat)]
notes [sepref-import-param] = IdI[of maxsize]
begin
  sepref-definition aial-contains is uncurry aial-contains :: nat-assn k∗a (ial-assn2 maxsize)k → a bool-assn
    unfolding aial-contains-def aial-assn2-def by sepref
end

sepref-decl-impl (ismop) aial-contains: aial-contains.refine[FCOMP aial-contains-refine]
.
end

end

3.12 Implementation of Heaps by Arrays

theory IICF-Impl-Heapmap
imports IICF-Abs-Heapmap ../IICF-Indexed-Array-List
begin

Some setup to circumvent the really inefficient implementation of division in the code generator, which has to consider several cases for negative divisors and dividends.

definition [code-unfold]:
  efficient-nat-div2 n ≡ nat-of-integer (fst (Code-Numeral.divmod-abs (integer-of-nat n) 2))

lemma efficient-nat-div2[simp]: efficient-nat-div2 n = n div 2
  by (simp add: efficient-nat-div2-def nat-of-integer.rep-eq)

type-synonym 'v hma = nat list × ('v list)

327
locale hmstruct-impl = hmstruct prio for prio :: 'v::heap ⇒ 'p::linorder
begin
  lemma param-prio: (prio,prio) ∈ Id → Id by simp
  lemmas [sepref-import-param] = param-prio
  sepref-register prio
end

context
  fixes maxsize :: nat
  fixes prio :: 'v::heap ⇒ 'p::linorder
  notes [map-type-eqs] = map-type-eqI [Pure.of TYPE((nat,'v) ahm) TYPE('v i-hma)]
begin
  interpretation hmstruct .
  interpretation hmstruct-impl .

definition hm-impl1-α ≡ λ(pq,ml).
  (pq,λk. if k∈ set pq then Some (ml!k) else None)

definition hm-impl1-invar ≡ λ(pq,ml).
  hmr-invar (hm-impl1-α (pq,ml))
  ∧ set pq ⊆ {0..<maxsize}
  ∧ ((pq=[] ∧ ml=[]) ∨ (length ml = maxsize))

definition hm-impl1-weak-invar ≡ λ(pq,ml).
  set pq ⊆ {0..<maxsize}
  ∧ ((pq=[] ∧ ml=[]) ∨ (length ml = maxsize))

definition hm-impl1-rel ≡ br hm-impl1-α hm-impl1-invar
  definition hm-weak-impl'¬rel ≡ br hm-impl1-α hm-impl1-weak-invar

  lemmas hm-impl1-rel-defs =
  hm-impl1-rel-def hm-weak-impl'¬rel-def hm-impl1-weak-invar-def hm-impl1-invar-def
  hm-impl1-α-def in-br-conv

  lemma hm-impl-α-fst-eq:
    (x1, x2) = hm-impl1-α (x1a, x2a) ⇒ x1 = x1a
  unfolding hm-impl1-α-def by (auto split: if-split-asm)

term hm-empty-op
  definition hm-empty-op' :: 'v hma nres
    where hm-empty-op' ≡ do {
      let pq = op-ial-empty-sz maxsize;
\begin{verbatim}
let ml = op-list-empty;
RETURN (pq,ml)
}

lemma hm-empty-op'-refine: (hm-empty-op', hm-empty-op) ∈ (hm-impl1-rel)nres-rel
  apply (intro fun-relI nres-relI)
  unfolding hm-empty-op'-def hm-empty-op-def hm-impl1-rel-defs
  by (auto simp: in-br-conv)

definition hm-length' :: 'v hma ⇒ nat where hm-length' ≡ λ(pq,ml). length pq

lemma hm-length'-refine: (RETURN o hm-length',RETURN o hm-length) ∈ hm-impl1-rel → (nat-rel)nres-rel
  apply (intro fun-relI nres-relI)
  unfolding hm-length'-def hm-length-def hm-impl1-rel-defs
  by (auto)

term hm-key-of-op
  definition hm-key-of-op' ≡ λ(pq,ml) i. ASSERT (i>0) ⇒ mop-list-get pq (i - 1)

lemma hm-key-of-op'-refine: (hm-key-of-op', hm-key-of-op) ∈ hm-impl1-rel → nat-rel → (nat-rel)nres-rel
  apply (intro fun-relI nres-relI)
  unfolding hm-key-of-op'-def hm-key-of-op-def hm-impl1-rel-defs
  by (auto)

term hm-lookup
  definition hm-lookup-op' ≡ λ(pq,ml) k. do {
      if (k<maxsize) then do { — TODO: This check can be eliminated, but this
        let c = op-list-contains k pq;
        if c then do {
          v ← mop-list-get ml k;
          RETURN (Some v)
        } else RETURN None
      } else RETURN None
  }

lemma hm-lookup-op'-refine: (uncurry hm-lookup-op', uncurry (RETURN oo hm-lookup))
  ∈ (hm-impl1-rel ×r nat-rel) ⇒ ((Id?option-rel)nres-rel
  apply (intro freflI nres-relI)
  unfolding hm-lookup-op-def hm-lookup-op'-def o-def uncurry-def
  apply refine-vcg
  apply (auto simp: hm-impl1-rel-defs heapmap-\alpha-def hmr-invar-def)
  done
\end{verbatim}
term \textit{hm-contains-key-op}

definition \textit{hm-contains-key-op'} \equiv \lambda (pq, ml). \text{do } \{ 
\text{if } (\text{\textless; } \text{\textless; } \text{maxsize}) \text{ then do } \{ 
\text{— TODO: This check can be eliminated, but this will complicate refinement of keys in basic ops} 
\text{RETURN } (\text{op-list-contains } k \ pq) 
\} \text{ else RETURN False} 
\}

lemma \textit{hm-contains-key-op'}-refine: (uncurry \textit{hm-contains-key-op'}, uncurry \textit{hm-contains-key-op}) 
\in (\text{nat-rel } \times, \text{hm-impl1-rel}) \rightarrow_f (\text{bool-rel})nres-rel 
apply (intro frefl nres-refl) 
unfolding \textit{hm-contains-key-op-def} \textit{hm-contains-key-op'}-def o-def uncurry-def 
PR-CONST-def 
apply refine-vcg 
apply (auto simp: \textit{hm-impl1-rel-defs heapmap-\alpha-def hmr-invar-def}) 
done

term \textit{hm-valid}

definition \textit{hm-exch-op'} \equiv \lambda (pq, ml) i j. \text{do } \{ 
\text{ASSERT } (\text{hm-valid } (\text{hm-impl1-\alpha} (pq, ml))) i; 
\text{ASSERT } (\text{hm-valid } (\text{hm-impl1-\alpha} (pq, ml))) j; 
\text{pq } \leftarrow \text{map-list-swap } pq (i - 1) (j - 1); 
\text{RETURN } (pq, ml) 
\}

lemma \textit{hm-impl1-relI}:
\text{assumes } \text{hm-r-inv b} 
\text{assumes } (a, b) \in \text{hm-weak-impl'}-rel 
\text{shows } (a, b) \in \text{hm-impl1-rel} 
using \text{assns} 
unfolding \textit{hm-rel-def} \textit{hm-impl1-rel-def} \textit{hm-weak-impl'}-rel-def in-br-conv 
\textit{hm-impl1-weak-invar-def} \textit{hm-impl1-invar-def} 
by auto

lemma \textit{hm-impl1-nres-relI}:
\text{assumes } b \leq_n \text{SPEC } \text{hm-r-inv} 
\text{assumes } (a, b) \in (\text{hm-weak-impl'}-rel) \text{nres-rel} 
\text{shows } (a, b) \in (\text{hm-impl1-rel}) \text{nres-rel} 
using \text{assns} \text{hm-impl1-relI} 
apply (auto simp: pw-le-iff pw-leof-iff refine-pw-simps in-br-conv nres-rel-def) 
apply blast 
done

lemma \textit{hm-exch-op'}-refine: (\textit{hm-exch-op'}, \textit{hm-exch-op}) \in \textit{hm-impl1-rel} \rightarrow \text{nat-rel}
→ nat-rel → (hm-impl1-rel)nres-rel
apply (intro fun-relI hm-impl1-nres-rell[OF hm-exch-op-invar])
unfolding hm-exch-op'-def hm-exch-op-def
apply (auto simp: pw-le-iff refine-pw-simps nres-rel-def
       hm-impl1-rel-def in-br-cone split: prod.splits)
apply (auto simp: hm-impl1-α-def)
unfolding hm-impl1-rel-defs
apply auto
done

term hm-index-op

definition hm-index-op' ≡ λ(pq,ml) k.
do {
  ASSERT (hm-impl1-invar (pq,ml) ∧ heapmap-α (hm-impl1-α (pq,ml))) k ≠ None ∧ k∈set pq);
i ← mop-list-index pq k;
  RETURN (i+1)
}

lemma hm-index-op'-refine: (hm-index-op',hm-index-op) ∈ hm-impl1-rel → nat-rel → (nat-rel)nres-rel
apply (intro fun-relI nres-rell)
unfolding hm-index-op'-def hm-index-op-def hm-impl1-rel-defs
apply (auto simp: pw-le-iff refine-pw-simps heapmap-α-def split: if-split-asm)
done

definition hm-update-op' where
hm-update-op' ≡ λ(pq,ml) i v. do {
  ASSERT (hm-valid (hm-impl1-α (pq,ml)) i ∧ hm-impl1-invar (pq,ml));
k ← mop-list-get pq (i - 1);
  ml ← mop-list-set ml k v;
  RETURN (pq, ml)
}

lemma hm-update-op'-refine: (hm-update-op', hm-update-op) ∈ hm-impl1-rel
→ nat-rel → Id → (hm-impl1-rel)nres-rel
apply (intro fun-relI hm-impl1-nres-rell[OF hm-update-op-invar])
unfolding hm-update-op'-def hm-update-op-def
apply (auto simp: pw-le-iff refine-pw-simps nres-rel-def
       hm-impl1-rel-def in-br-cone split: prod.splits)
apply (auto simp: hm-impl1-α-def)
unfolding hm-impl1-rel-defs
apply (auto simp: subset-code(1))
done

term hm-butlast-op

lemma hm-butlast-op-invar: hm-butlast-op hm ≤ₜ SPEC hm-r-invar
unfolding  hm-butlast-op-def  h.butlast-op-def
apply  refine-vcg
apply  (clarsimp-all simp: hmr-rel-defs map-butlast distinct-butlast)
apply  safe

apply  (auto simp: in-set-cone-nth nth-butlast) []
apply  (metis Suc-pred len-greater-imp-nonempty length-greater-0-conv less-antisym)

apply  (auto dest: in-set-butlastD) []
apply  (metis One-nat-def append-butlast-last-id distinct-butlast last-conv-nth
not-distinct-conv-prefix)
done

definition  hm-butlast-op’  where
hm-butlast-op’  ≡ λ(pq,ml). do {
  ASSERT (hmr-invar (hm-impl1-α (pq,ml)));
  pq ← mop-list-butlast pq;
  RETURN (pq,ml)
}

lemma  set-butlast-distinct-conv: 
[distinct l]  ⇒  set (butlast l) = set l - {last l}
by  (cases l rule: rev-cases; auto)

lemma  hm-butlast-op’-refine: (hm-butlast-op’, hm-butlast-op) ∈ hm-impl1-rel
→ (hm-impl1-rel)nres-rel
apply  (intro fun-rell hm-impl1-nres-rell[OF hm-butlast-op-invar])
unfolding  hm-butlast-op’-def  hm-butlast-op-def
apply  (auto simp: pw-le-iff refine-pw-simps nres-rel-def
hm-impl1-rel-def in-br-conv split: prod.splits)
apply  (auto simp: hm-impl1-α-def)
unfolding  hm-impl1-rel-defs
apply  (auto simp: restrict-map-def) []

defer

apply  (auto dest: in-set-butlastD) []
apply  (auto intro!: ext
simp: hmr-invar-def set-butlast-distinct-cone last-conv-nth
dest: in-set-butlastD) []
done

definition  hm-append-op’  where
hm-append-op’  ≡ λ(pq,ml) k v. do {
  ASSERT (k /∈ set pq ∧ k<maxsize);
  ASSERT (hm-impl1-invar (pq,ml));
  pq ← mop-list-append pq k;

332
\[
ml \leftarrow \text{(if length } ml = 0 \text{ then mop-list-replicate maxsize } v \text{ else RETURN } ml); \nlml \leftarrow \text{mop-list-set } ml \ k \ v; \n\text{RETURN (pq,ml)} \}
\]

\textbf{lemma} \ hml-\text{append-op}\rprime\text{-refine:} \ (\text{uncurry2 } hml-\text{append-op}\rprime, \text{uncurry2 } hml-\text{append-op})

\in \ [\lambda((hml,k),v). \ k < \text{maxsize}]_f (hml-\text{impl1-rel} \times_r \text{nat-rel}) \times_r \text{Id} \rightarrow (hml-\text{impl1-rel})\text{nres-rel}

\textbf{apply} \ (\text{intro freI } hml-\text{impl1-nres-relI}[OF hml-\text{append-op-invar}])
\textbf{unfolding} \ hml-\text{append-op}\rprime\text{-def } hml-\text{append-op}\text{-def}
\textbf{apply} \ (\text{auto simp: pw-le-iff refine-pw-simps nres-rel-def}
\quad \text{hm-impl1-rel-def in-br-conv split: prod.splits})
\textbf{unfolding} \ hml-\text{impl1-rel-defs}\n\textbf{apply} \ (\text{auto simp: restrict-map-def hmr-invar-def split: prod.splits if-split-asm})
\textbf{done}

\textbf{definition} \ hml-\text{impl2-rel} \equiv \text{prod-assn (ial-assn maxsize id-assn) (array-assn id-assn)}
\textbf{definition} \ hml-\text{impl-rel} \equiv \text{hr-comp hml-\text{impl2-rel } hml-\text{impl1-rel}}

\textbf{lemmas} \ [\text{fcomp-norm-unfold} = hml-\text{impl-rel-def [symmetric]}]

\section{3.12.1 Implement Basic Operations}

\textbf{lemma} \ param-parent: \ (\text{efficient-nat-div2} h.parent) \in \text{Id} \rightarrow \text{Id}
\textbf{by} \ (\text{intro fun-relI}) \ (\text{simp add: h.parent-def})
\textbf{lemmas} \ [\text{sepref-import-param} = param-parent]
\textbf{sepref-register} \ h.parent

\textbf{lemma} \ param-left: \ (h.left,h.left) \in \text{Id} \rightarrow \text{Id} \text{ by simp}
\textbf{lemmas} \ [\text{sepref-import-param} = param-left]
\textbf{sepref-register} \ h.left

\textbf{lemma} \ param-right: \ (h.right,h.right) \in \text{Id} \rightarrow \text{Id} \text{ by simp}
\textbf{lemmas} \ [\text{sepref-import-param} = param-right]
\textbf{sepref-register} \ h.right

\textbf{abbreviation} \ (\text{input}) \ prio-rel \equiv (\text{Id::(’p×’p) set})

\textbf{lemma} \ param-prio-le: \ ((\leq), (\leq)) \in \text{prio-rel} \rightarrow \text{prio-rel} \rightarrow \text{bool-rel by simp}
\textbf{lemmas} \ [\text{sepref-import-param} = param-prio-le]

\textbf{lemma} \ param-prio-lt: \ ((<), (<)) \in \text{prio-rel} \rightarrow \text{prio-rel} \rightarrow \text{bool-rel by simp}
\textbf{lemmas} \ [\text{sepref-import-param} = param-prio-lt]

\textbf{abbreviation} \ I-HM-UNF \equiv TYPE(\text{nat list } × \text{’v list})
sepref-definition \( \text{hm-length-impl} \) is \( \text{return} \ o \ \text{hm-length}' :: \text{hm-impl2-rel}^k \rightarrow_a \text{nat-assn} \)

unfolding \( \text{hm-length}'\)-def \text{hm-impl2-rel-def}

by sepref

lemmas [sepref-frules] = \text{hm-length-impl}\.refine[FCOMP \text{hm-length}'\.-refine]

sepref-register \( \text{hm-length}::(nat,'v) ahm \Rightarrow - \)

sepref-definition \( \text{hm-key-of-op-impl} \) is \( \text{uncurry} \ \text{hm-key-of-op}' :: \text{hm-impl2-rel}^k *_a \text{nat-assn}^k \rightarrow_a \text{nat-assn} \)

unfolding \( \text{hm-key-of-op}'\)-def \text{hm-impl2-rel-def}

by sepref

lemmas [sepref-frules] = \text{hm-key-of-op-impl}\.refine[FCOMP \text{hm-key-of-op}'\.-refine]

sepref-register \( \text{hm-key-of-op}::(nat,'v) ahm \Rightarrow - \)

context

notes [id-rules] = \text{itypeI}[\text{Pure}\.of maxsize TYPE(nat)]

notes [sepref-import-param] = \text{IdI}[of maxsize]

begin

sepref-definition \( \text{hm-lookup-impl} \) is \( \text{uncurry} \ \text{hm-lookup-op}' :: (\text{hm-impl2-rel}^k *_a \text{nat-assn}^k \rightarrow_a \text{option-assn} \ id-assn) \)

unfolding \( \text{hm-lookup-op}'\)-def \text{hm-impl2-rel-def}

by sepref

lemmas [sepref-frules] = \text{hm-lookup-impl}\.refine[FCOMP \text{hm-lookup-op}'\.-refine]

sepref-register \( \text{hm-lookup}::(nat,'v) ahm \Rightarrow - \)

sepref-definition \( \text{hm-exch-op-impl} \) is \( \text{uncurry2} \ \text{hm-exch-op}' :: \text{hm-impl2-rel}^d *_a \text{nat-assn}^k *_a \text{nat-assn}^k \rightarrow_a \text{hm-impl2-rel} \)

unfolding \( \text{hm-exch-op}'\)-def \text{hm-impl2-rel-def}

by sepref

lemmas [sepref-frules] = \text{hm-exch-op-impl}\.refine[FCOMP \text{hm-exch-op}'\.-refine]

sepref-register \( \text{hm-exch-op}::(nat,'v) ahm \Rightarrow - \)

sepref-definition \( \text{hm-index-op-impl} \) is \( \text{uncurry} \ \text{hm-index-op}' :: \text{hm-impl2-rel}^k *_a \text{id-assn}^k \rightarrow_a \text{id-assn} \)

unfolding \( \text{hm-index-op}'\)-def \text{hm-impl2-rel-def}

by sepref

lemmas [sepref-frules] = \text{hm-index-op-impl}\.refine[FCOMP \text{hm-index-op}'\.-refine]

sepref-register \( \text{hm-index-op}::(nat,'v) ahm \Rightarrow - \)

sepref-definition \( \text{hm-update-op-impl} \) is \( \text{uncurry2} \ \text{hm-update-op}' :: \text{hm-impl2-rel}^d *_a \text{id-assn}^k *_a \text{id-assn}^k \rightarrow_a \text{hm-impl2-rel} \)

unfolding \( \text{hm-update-op}'\)-def \text{hm-impl2-rel-def}

by sepref

lemmas [sepref-frules] = \text{hm-update-op-impl}\.refine[FCOMP \text{hm-update-op}'\.-refine]

sepref-register \( \text{hm-update-op}::(nat,'v) ahm \Rightarrow - \)

334
sepref-definition hm-butlast-op-impl is hm-butlast-op' :: hm-impl2-rel\(d\) \(\rightarrow_a\)

hm-impl2-rel

unfolding hm-butlast-op'-def hm-impl2-rel-def by sepref
lemmas [sepref-fr-rules] = hm-butlast-op-impl.refine[FCOMP hm-butlast-op'-refine]
sepref-register hm-butlast-op::(nat,'v) ahm \(\Rightarrow\) -

sepref-definition hm-append-op-impl is uncurry2 hm-append-op' :: hm-impl2-rel\(d\)

\(\ast_a\) id-assn\(k\) \(\ast_a\) id-assn\(k\) \(\rightarrow_a\) hm-impl2-rel

unfolding hm-append-op'-def hm-impl2-rel-def

apply (rewrite array-fold-custom-replicate)
by sepref
lemmas [sepref-fr-rules] = hm-append-op-impl.refine[FCOMP hm-append-op'-refine]
sepref-register hm-append-op::(nat,'v) ahm \(\Rightarrow\) -

3.12.2 Auxiliary Operations

lemmas [intf-of-assn] = intf-of-assnI[where R=hm-impl-rel :: (nat,'v) ahm
\(\Rightarrow\) - and 'a='v i-hma]

sepref-definition hm-valid-impl is uncurry (RETURN oo hm-valid) :: hm-impl-rel\(k\) \(\ast_a\) nat-assn\(k\)
\(\rightarrow_a\) bool-assn
unfolding hm-valid-def[abs-def]
by sepref
lemmas [sepref-fr-rules] = hm-valid-impl.refine
sepref-register hm-valid::(nat,'v) ahm \(\Rightarrow\) -

definition hm-the-lookup-op' hm k \equiv do {
let (pq,ml) = hm;
ASSERT (heapmap-\(\alpha\) (hm-impl1-\(\alpha\) hm) k \(\neq\) None \(\land\) hm-impl1-invar hm);
v \leftarrow mop-list-get ml k;
RETURN v
}

lemma hm-the-lookup-op'-refine:
(hm-the-lookup-op', hm-the-lookup-op) \(\in\) hm-impl1-rel \(\rightarrow\) nat-rel \(\rightarrow\) (Id)nres-rel
apply (intro fun-relI nres-relI)
unfolding hm-the-lookup-op'-def hm-the-lookup-op-def
apply refine-vcg
  apply (auto simp: hm-impl1-rel-defs heapmap-\(\alpha\)-def hmr-invar-def split: if-split-asm)
done

sepref-definition hm-the-lookup-op-impl is uncurry hm-the-lookup-op' :: hm-impl2-rel\(k\) \(\ast_a\) id-assn\(k\)
\(\rightarrow_a\) id-assn
unfolding hm-the-lookup-op'-def[abs-def] hm-impl2-rel-def
by sepref
sepref-register \( \text{hm-the-lookup-op} :: (\text{nat}', v) \text{ ahm} \Rightarrow - \)

sepref-definition \( \text{hm-val-of-op-impl is uncurry} \ \text{hm-val-of-op} :: \text{hm-impl-rel}^{a}_a \text{id-assn}^k \rightarrow_{a} \text{id-assn} \)

unfolding \( \text{hm-val-of-op-def by sepref} \)

lemmas [sepref-fr-rules] = \text{hm-val-of-op-impl.refine}

sepref-register \( \text{hm-val-of-op} :: (\text{nat}', v) \text{ ahm} \Rightarrow - \)

sepref-definition \( \text{hm-prio-of-op-impl is uncurry} \ (\text{PR-CONST} \ \text{hm-prio-of-op}) \)

:: \( \text{hm-impl-rel}^{a}_a \text{id-assn}^k \rightarrow_{a} \text{id-assn} \)

unfolding \( \text{hm-prio-of-op-def[abs-def]} \ \text{PR-CONST-def by sepref} \)

lemmas [sepref-fr-rules] = \text{hm-prio-of-op-impl.refine}

sepref-register \( \text{PR-CONST} \ \text{hm-prio-of-op} :: (\text{nat}', v) \text{ ahm} \Rightarrow - \)

lemma [def-pat-rules]: \text{hmstruct.} \text{hm-prio-of-op}$\text{prio} \equiv \text{PR-CONST} \ \text{hm-prio-of-op}$

by simp

No code theorem preparation, as we define optimized version later

sepref-definition \( \text{(no-prep-code) hm-swim-op-impl is uncurry} \ (\text{PR-CONST} \ \text{hm-swim-op}) :: \text{hm-impl-rel}^{a}_a \text{id-assn}^k \rightarrow_{a} \text{id-assn} \)

unfolding \( \text{hm-swim-op-def[abs-def]} \ \text{PR-CONST-def} \)

using [[goals-limit = 1]]

by sepref

lemmas [sepref-fr-rules] = \text{hm-swim-op-impl.refine}

sepref-register \( \text{PR-CONST} \ \text{hm-swim-op} :: (\text{nat}', v) \text{ ahm} \Rightarrow - \)

lemma [def-pat-rules]: \text{hmstruct.} \text{hm-swim-op}$\text{prio} \equiv \text{PR-CONST} \ \text{hm-swim-op}$

by simp

No code theorem preparation, as we define optimized version later

sepref-definition \( \text{(no-prep-code) hm-sink-op-impl is uncurry} \ (\text{PR-CONST} \ \text{hm-sink-op}) :: \text{hm-impl-rel}^{a}_a \text{id-assn}^k \rightarrow_{a} \text{id-assn} \)

unfolding \( \text{hm-sink-op-def[abs-def]} \ \text{PR-CONST-def} \)

by sepref

lemmas [sepref-fr-rules] = \text{hm-sink-op-impl.refine}

sepref-register \( \text{PR-CONST} \ \text{hm-sink-op} :: (\text{nat}', v) \text{ ahm} \Rightarrow - \)

lemma [def-pat-rules]: \text{hmstruct.} \text{hm-sink-op}$\text{prio} \equiv \text{PR-CONST} \ \text{hm-sink-op}$

by simp

sepref-definition \( \text{hm-repair-op-impl is uncurry} \ (\text{PR-CONST} \ \text{hm-repair-op}) :: \text{hm-impl-rel}^{a}_a \text{id-assn}^k \rightarrow_{a} \text{id-assn} \)

unfolding \( \text{hm-repair-op-def[abs-def]} \ \text{PR-CONST-def} \)

by sepref

lemmas [sepref-fr-rules] = \text{hm-repair-op-impl.refine}

sepref-register \( \text{PR-CONST} \ \text{hm-repair-op} :: (\text{nat}', v) \text{ ahm} \Rightarrow - \)

lemma [def-pat-rules]: \text{hmstruct.} \text{hm-repair-op}$\text{prio} \equiv \text{PR-CONST} \ \text{hm-repair-op}$

by simp

3.12.3 Interface Operations

definition \( \text{hm-rel-np where} \)
\( \text{hm-rel-np} \equiv \text{hr-comp hm-impl-rel heapmap-rel} \)

lemmas \( \text{fcomp-norm-unfold} = \text{hm-rel-np-def}\) [symmetric]

definition \( \text{hm-rel} \) where
\( \text{hm-rel } K \ V \equiv \text{hr-comp hm-rel-np } ((\text{the-pure } K, \text{the-pure } V) \text{map-rel}) \)

lemmas \( \text{fcomp-norm-unfold} = \text{hm-rel-def}\) [symmetric]

lemmas \( \text{intf-of-assn} = \text{intf-of-assnI}\) [where \( R=\text{hm-rel } K \ V \) and \( 'a=( 'k, 'v) \) i-map for \( K \ V \)]

lemma \( \text{hm-rel-id-conv: hm-rel id-assn id-assn = hm-rel-np} \)
— Used for generic algorithms: Unfold with this, then let decl-impl compose with map-rel again.

unfolding \( \text{hm-rel-def by simp} \)

Synthesis

definition \( \text{op-hm-empty-sz :: nat } \Rightarrow \ 'k \rightarrow 'v \)

where [simp]: \( \text{op-hm-empty-sz sz } \equiv \text{op-map-empty}\)

sepref-register \( \text{PR-CONST } (\text{op-hm-empty-sz maxsize }) :: ( 'k, 'v ) \) i-map

lemma [def-pat-rules]: \( \text{op-hm-empty-sz sz maxsize } \equiv \text{UNPROTECT } (\text{op-hm-empty-sz maxsize}) \) by simp

lemma \( \text{hm-fold-custom-empty-sz:} \)

\( \text{op-map-empty } = \text{op-hm-empty-sz sz} \)

\( \text{Map.empty } = \text{op-hm-empty-sz sz} \)

by auto

sepref-definition \( \text{hm-empty-op-impl is uncurry0 } \text{hm-empty-op}' :\text{ unit-assn } k \rightarrow_a \text{hm-impl2-rel} \)

unfolding \( \text{hm-empty-op'-def} \text{ hm-impl2-rel-def array.fold-custom-empty} \)

by sepref

sepref-definition \( \text{hm-insert-op-impl is uncurry2 } \text{hm-insert-op} :: [\lambda((k, _), _). k <\text{maxsize}]_a \text{id-assn}^k *_a \text{id-assn}^k *_a \text{hm-impl-rel}^k \rightarrow_a \text{hm-impl-rel} \)

unfolding \( \text{hm-insert-op-def} \)

by sepref

sepref-definition \( \text{hm-is-empty-op-impl is hm-is-empty-op :: hm-impl-rel}^k \rightarrow_a \text{bool-assn} \)

unfolding \( \text{hm-is-empty-op-def} \)

by sepref

sepref-definition \( \text{hm-lookup-op-impl is uncurry } \text{hm-lookup-op} :: \text{id-assn}^k *_a \text{hm-impl-rel}^k \rightarrow_a \text{option-assn id-assn} \)

unfolding \( \text{hm-lookup-op-def by sepref} \)

sepref-definition \( \text{hm-contains-key-impl is uncurry } \text{hm-contains-key-op'} :: \text{id-assn}^k *_a \text{hm-impl2-rel}^k \rightarrow_a \text{bool-assn} \)

337
unfolding \(\text{hm-contains-key-op}'\)-def \(\text{hm-impl2-rel-def}\) by sepref

sepref-definition \(\text{hm-decrease-key-op-impl}\) is uncurry2 \(\text{hm-decrease-key-op} :: \text{id-assn}^k \ast \text{id-assn}^k \ast \text{hm-impl-reld} \to_a \text{hm-impl-rel}\)
unfolding \(\text{hm-decrease-key-op-def}\) by sepref

sepref-definition \(\text{hm-increase-key-op-impl}\) is uncurry2 \(\text{hm-increase-key-op} :: \text{id-assn}^k \ast \text{id-assn}^k \ast \text{hm-impl-reld} \to_a \text{hm-impl-rel}\)
unfolding \(\text{hm-increase-key-op-def}\) by sepref

sepref-definition \(\text{hm-change-key-op-impl}\) is uncurry2 \(\text{hm-change-key-op} :: \text{id-assn}^k \ast \text{id-assn}^k \ast \text{hm-impl-reld} \to_a \text{hm-impl-rel}\)
unfolding \(\text{hm-change-key-op-def}\) by sepref

sepref-definition \(\text{hm-pop-min-op-impl}\) is \(\text{hm-pop-min-op} :: \text{hm-impl-reld} \to_a \text{hm-impl-rel}\)
prod-assn (prod-assn nat-assn id-assn) \(\text{hm-impl-rel}\)
unfolding \(\text{hm-pop-min-op-def}[\text{abs-def}]\) by sepref

sepref-definition \(\text{hm-remove-op-impl}\) is uncurry \(\text{hm-remove-op} :: \text{id-assn}^k \ast \text{hm-impl-reld} \to_a \text{hm-impl-rel}\)
unfolding \(\text{hm-remove-op-def}[\text{abs-def}]\) by sepref

sepref-definition \(\text{hm-peek-min-op-impl}\) is \(\text{hm-peek-min-op} :: \text{hm-impl-rel}^k \to_a \text{prod-assn nat-assn id-assn}\)
unfolding \(\text{hm-peek-min-op-def}[\text{abs-def}]\) \(\text{hm-kv-of-op-def}\) by sepref

Setup of Refinements

sepref-decl-impl \((\text{no-register})\) \(\text{hm-empty}:\)
\(\text{hm-empty-op-impl}\).refine\([\text{FCOMP \(\text{hm-empty-op}'\)-refine, FCOMP \(\text{hm-empty-aref}\)}]\).

context fixes \(K\) assumes \(\text{IS-BELOW-ID} K\) begin
lemmas mop-map-update-new-fref' = mop-map-update-new.fref[of K]
lemmas op-map-update-fref' = op-map-update.fref[of K]
end

sepref-decl-impl \((\text{ismop})\) \(\text{hm-insert}: \text{hm-insert-op-impl}\).refine\([\text{FCOMP \(\text{hm-insert-op-aref}\)}]\)
uses mop-map-update-new-fref'
unfolding \(\text{IS-BELOW-ID-def}\)
apply (parametricity; auto simp: single-valued-below-Id)
done

sepref-decl-impl \(\text{hm-is-empty}: \text{hm-is-empty-op-impl}\).refine\([\text{FCOMP \(\text{hm-is-empty-aref}\)}]\).

sepref-decl-impl \(\text{hm-lookup}: \text{hm-lookup-op-impl}\).refine\([\text{FCOMP \(\text{hm-lookup-op-aref}\)}]\)

338
sepref-decl-impl hm-contains-key:
hm-contains-key-impl.refine[FCOMP hm-contains-key-op'-refine, FCOMP hm-contains-key-op-aref]
.

sepref-decl-impl (ismop) hm-decrease-key: hm-decrease-key-op-impl.refine[FCOMP hm-decrease-key-op-aref] .
sepref-decl-impl (ismop) hm-increase-key: hm-increase-key-op-impl.refine[FCOMP hm-increase-key-op-aref] .
sepref-decl-impl (ismop) hm-change-key: hm-change-key-op-impl.refine[FCOMP hm-change-key-op-aref] .

sepref-decl-impl (ismop) hm-remove: hm-remove-op-impl.refine[FCOMP hm-remove-op-aref] .


— Realized as generic algorithm. Note that we use id-assn for the elements.
sepref-definition hm-upd-op-impl is uncurry2 (RETURN ooo op-map-update)
:: \(\lambda(k, -). \ k<\text{maxsize}_a \ \text{id-assn}^k \ *_a \ \text{id-assn}^k \ *_a \ (\text{hm-rel id-assn id-assn})^t \ \rightarrow \ \text{hm-rel id-assn id-assn} \)
unfolding op-pm-set-gen-impl by sepref
sepref-decl-impl hm-upd-op-impl.refine[unfolded hm-rel-id-conv] uses op-map-update-fref'
unfolding IS-BELOW-ID-def
apply (parametricity; auto simp: single-valued-below-Id)
done
end
end

interpretation hm: map-custom-empty PR-CONST (op-hm-empty-sz maxsize)
apply unfold-locales by simp

lemma op-hm-empty-sz-hnr[sepref-fr-rules]:
(uncurry0 (hm-empty-op-impl maxsize), uncurry0 (RETURN (PR-CONST (op-hm-empty-sz maxsize)))) \in \text{unit-assn}^k \rightarrow_\text{a} \ \text{hm-rel maxsize prio K V}
using hm-empty-hnr by simp

3.12.4 Manual fine-tuning of code-lemmas

context
notes [simp del] = CNV-def efficient-nat-div2
begin
lemma nested-case-bind:
(case p of (a,b) ⇒ bind (case a of (a1,a2) ⇒ m a b a1 a2) (f a b))
= (case p of ((a1,a2),b) ⇒ bind (m a1 a2) b a1 a2) (f (a1,a2) b))
(case p of (a,b) ⇒ bind (case b of (b1,b2) ⇒ m a b b1 b2) (f a b))
= (case p of (a,b1,b2) ⇒ bind (m a (b1,b2) b1 b2) (f a (b1,b2)))
by (simp-all split: prod.splits)

lemma it-case: (case p of (a,b) ⇒ f p a b) = (case p of (a,b) ⇒ f (a,b) a b)
by (auto split: prod.split)

lemma c2l: (case p of (a,b) ⇒ bind (m a b) (f a b)) =
do { let (a,b) = p; bind (m a b) (f a b)} by simp

lemma bind-Let: do { x ← do { let y = v; (f g :: 'a Heap)}; g x } = do { let y=v; x ← f y; g x } by auto

lemma bind-case: do { x ← (case y of (a,b) ⇒ f a b); (g x :: 'a Heap) } = do {
let (a,b) = y; x ← f a b; g x }
by (auto split: prod.splits)

lemma bind-case-mvup: do { x ← f; case y of (a,b) ⇒ g a b x }
= do { let (a,b) = y; x ← f; (g a b x :: 'a Heap) }
by (auto split: prod.splits)

lemma if-case-mvup: (if b then case p of (x1,x2) ⇒ f x1 x2 else e)
= (case p of (x1,x2) ⇒ if b then f x1 x2 else e) by auto

lemma nested-case: (case p of (a,b) ⇒ (case p of (c,d) ⇒ f a b c d)) =
(case p of (a,b) ⇒ f a b a b)
by (auto split: prod.split)

lemma split-prod-bound: (λp. f p) = (λ(a,b). f (a,b)) by auto

lemma bpc-conv: do { (a,b) ← (m::(-*-) Heap); f a b } = do {
ab ← (m);
f (fst ab) (snd ab)
}
apply (subst (2) split-prod-bound)
by simp

lemma it-case-pp: (case p of ((p1,p2)) ⇒ case p of ((p1’,p2’)) ⇒ f p1 p2 p1’ p2’)
= (case p of ((p1,p2)) ⇒ f p1 p2 p1 p2)
by (auto split: prod.split)

lemma it-case-ppp: (case p of ((p1,p2),p3) ⇒ case p of ((p1’,p2’,p3’)) ⇒ f p1 p2 p3 p1’ p2’ p3’)
= (case p of ((p1,p2),p3) ⇒ f p1 p2 p3 p1 p2 p3)
by (auto split: prod.split)
lemma it-case-pppp: (case a1 of
  (((a, b), c), d) ⇒
  case a1 of
    (((a', b'), c'), d') ⇒ f a b c d a' b' c' d') =
  (case a1 of
    (((a, b), c), d) ⇒ f a b c d a b c d)
by (auto split: prod.splits)

private lemmas inlines = hm-append-op-impl-def ial-append-def
  marl-length-def marl-append-def hm-length-impl-def ial-length-def
  hm-valid-impl-def hm-prio-of-op-impl-def hm-val-if-op-impl-def
  hm-key-of-op-impl-def
  ial-get-def hm-the-lookup-op-impl-def heap-array-set-def
  mart-get-def
  it-case-ppp it-case-pppp bind-case bind-case-mvup
  nested-case if-case-mvup
  it-case-pp

schematic-goal [code]: hm-insert-op-impl maxsize prio hm k v = ?f
unfolding hm-insert-op-impl-def
apply (rule CNV-eqD)
apply (simp add: inlines cong: if-cong)
by (rule CNV-I)

schematic-goal hm-swim-op-impl prio hm i ≡ ?f
unfolding hm-swim-op-impl-def
apply (rule eq-reflection)
apply (rule CNV-eqD)
apply (simp add: inlines efficient-nat-div2 cong: if-cong)
by (rule CNV-I)

lemma hm-swim-op-impl-code [code]: hm-swim-op-impl prio hm i ≡ ccpo.fixp (fun-lub
Heap-lub) (fun-ord Heap-ord)
  (λcf (a1, a2).
  case a1 of
    (((a1b, a2b), a2a) ⇒
      case a1b of
        (a, b) ⇒ do {
          let d2 = efficient-nat-div2 a2;
          if 0 < d2 ∧ d2 ≤ b
          then do {
            x ← (case a1b of (a, n) ⇒ Array.nth a) (d2 − Suc 0);
            x ← Array.nth a2a x;
            xa ← (case a1b of (a, n) ⇒ Array.nth a) (a2 − Suc 0);
            xa ← Array.nth a2a xa;
            if prio x ≤ prio xa then return a1
            else do {
              x'g ← hm-exch-op-impl a1 a2 (d2);
            }
          }
        }
      }
    )
cf \( (x'g, d2) \)

\}

else return \( a1 \)

\}

\}

\}

unfolding \( \text{hm-swim-op-impl-def} \)

apply (rule eq-reflection)

apply (simp add: inlines \( \text{efficient-nat-div2} \) Let-def cong: if-cong)

done

prepare-code-thms \( \text{hm-swim-op-impl-code} \)

schematic-goal \( \text{hm-sink-opt-impl-code}[\text{code}] \): \( \text{hm-sink-op-impl prio hm i} \equiv \ ?f \)

unfolding \( \text{hm-sink-op-impl-def} \)

apply (rule eq-reflection)

apply (rule CNV-eqD)

apply (simp add: inlines cong: if-cong)

by (rule CNV-I)

prepare-code-thms \( \text{hm-sink-opt-impl-code} \)

export-code \( \text{hm-swim-op-impl in} \ SML-imp\) module-name \( \text{Test} \)

schematic-goal \( \text{hm-change-key-opt-impl-code}[\text{code}] \):
\( \text{hm-change-key-op-impl prio k v hm} \equiv \ ?f \)

unfolding \( \text{hm-change-key-op-impl-def} \)

apply (rule eq-reflection)

apply (rule CNV-eqD)

apply (simp add: inlines \( \text{hm-contains-key-impl-def} \) \( \text{ial-contains-def} \) \( \text{hm-change-key-op-impl-def} \) \( \text{hm-index-op-impl-def} \) \( \text{hm-update-op-impl-def} \) \( \text{ial-index-def} \) cong: if-cong split: prod.splits)

oops

schematic-goal \( \text{hm-change-key-opt-impl-code}[\text{code}] \):
\( \text{hm-change-key-op-impl prio k v hm} \equiv \ case \ hm \ of \ (((a, b), ba), x2) \Rightarrow \)
\( \) (do 
\( x \leftarrow \) Array.nth ba k;
\( xa \leftarrow \) Array.nth a x;
\( xa \leftarrow \) Arrayupd xa v x2;
\( \text{hm-repair-op-impl prio } (((a, b), ba), xa) \ (\text{Suc x}) \)
\) 

unfolding \( \text{hm-change-key-op-impl-def} \)

apply (rule eq-reflection)
apply (simp add: inlines hm-contains-key-impl-def ial-contains-def
hm-change-key-op-impl-def hm-index-op-impl-def hm-update-op-impl-def
ial-index-def
cong: if-cong split: prod.splits)
done

schematic-goal hm-set-opt-impl-code[code]: hm-upd-op-impl maxsize prio hm k v
≡ ?f
  unfolding hm-upd-op-impl-def
  apply (rule eq-reflection)
  apply (rule CNV-eqD)
  apply (simp add: inlines hm-contains-key-impl-def ial-contains-def
hm-change-key-op-impl-def hm-index-op-impl-def hm-update-op-impl-def
ial-index-def
cong: if-cong)
  by (rule CNV-I)

schematic-goal hm-pop-min-opt-impl-code[code]: hm-pop-min-op-impl prio hm
≡ ?f
  unfolding hm-pop-min-op-impl-def
  apply (rule eq-reflection)
  apply (rule CNV-eqD)
  apply (simp add: inlines hm-contains-key-impl-def ial-contains-def
hm-change-key-op-impl-def hm-index-op-impl-def hm-update-op-impl-def
hm-butlast-op-impl-def ial-butlast-def
ial-index-def
cong: if-cong)
  by (rule CNV-I)
end

export-code
hm-empty-op-impl
hm-insert-op-impl
hm-is-empty-op-impl
hm-lookup-op-impl
hm-contains-key-impl
hm-decrease-key-op-impl
hm-increase-key-op-impl
hm-change-key-op-impl
hm-upd-op-impl
hm-pop-min-op-impl
hm-remove-op-impl
hm-peek-min-op-impl
checking SML-imp
end
3.13 Matrices

theory ICF-Matrix
imports ../../../Sepref
begin

3.13.1 Relator and Interface

definition [to-relAPP]: mtx-rel A ≡ nat-rel ×r nat-rel → A

lemma mtx-rel-id [simp]: ⟨Id⟩ mtx-rel = Id unfolding mtx-rel-def by auto

type-synonym ′a mtx = nat × nat ⇒ ′a

type-synonym ′a i-mtx is nat × nat ⇒ ′a

lemma [synth-rules]: INTF-OF-REL A TYPE(′a) ⇒ INTF-OF-REL (⟨A⟩ mtx-rel)
by simp

3.13.2 Operations

definition op-mtx-new :: ′a mtx ⇒ ′a mtx where [simp]: op-mtx-new c = c

sepref-decl-op (no-def) mtx-new: op-mtx-new :: (nat-rel ×r nat-rel → A) → ⟨A⟩ mtx-rel
apply (rule fref-ncl) unfolding op-mtx-new-def [abs-def] mtx-rel-def
by parametricity

lemma mtx-init-adhoc-frame-match-rule [sepref-frame-match-rules]:
hn-val (nat-rel ×r nat-rel → A) x y =⇒ hn-val (nat-rel ×r nat-rel → the-pure (pure A)) x y
by simp

definition op-mtx-copy :: ′a mtx ⇒ ′a mtx where [simp]: op-mtx-copy c = c

sepref-decl-op (no-def) mtx-copy: op-mtx-copy :: ⟨A⟩ mtx-rel → ⟨A⟩ mtx-rel.

sepref-decl-op mtx-get: λ(c::′a mtx) ij. c ij :: ⟨A⟩ mtx-rel → (nat-rel ×r nat-rel) → A
apply (rule fref-ncl) unfolding mtx-rel-def
by parametricity

sepref-decl-op mtx-set: fun-apd::′a mtx ⇒ - :: ⟨A⟩ mtx-rel → (nat-rel ×r nat-rel) → A → ⟨A⟩ mtx-rel
apply (rule fref-ncl)
unfolding mtx-rel-def
proof goal-cases case 1
have [param]: ((=), (=)) ∈ nat-rel ×r nat-rel → nat-rel ×r nat-rel → bool-rel
by simp

344
show ?case by parametricity
qed

definition mtx-nonzero :: - mtx ⇒ (nat×nat) set where mtx-nonzero m ≡ {(i,j). m (i,j)≠0}

sepref-decl-op mtx-nonzero: mtx-nonzero :: (A) mtx-rel → (nat-rel×nat-rel) set-rel
  where IS-ID (A::(-×(-::zero)) set)
proof goal-cases
  case 1 assume IS-ID A hence U: A=Id by (simp only: IS-ID-def)
  have [param]: ((=),(=))∈A→A→bool-rel using U by simp
  show ?case
    apply (rule fref-ncI)
    unfolding mtx-rel-def
    apply parametricity
    unfolding U by simp-all
qed

3.13.3 Patterns

lemma pat-amtx-get: c$≡op-mtx-get$'c$'e by simp
lemma pat-amtx-set: fun-upd$'c$'e ≡op-mtx-set$'c$'e$'v by simp
lemmas amtx-pats[pat-rules] = pat-amtx-get pat-amtx-set

3.13.4 Pointwise Operations

Auxiliary Definitions and Lemmas

locale pointwise-op =
  fixes f :: 'p ⇒ 's ⇒ 's
  fixes q :: 's ⇒ 'p ⇒ 'a
  assumes upd-indep1[simp, intro]: p≠p' ⇒ q (f p s) p' = q s p'
  assumes upd-indep2[simp, intro]: p≠p' ⇒ q (f p (f p' s)) p = q (f p s) p
begin
  lemma pointwise-upd-fold: distinct ps
    ⇒ q (fold f ps s) p = (if p∈set ps then q (f p s) p else q s p)
    by (induction ps arbitrary: s) auto
end

lemma pointwise-fun-fold:
  fixes f :: 'a ⇒ ('a ⇒ 'b) ⇒ ('a ⇒ 'b)
  fixes s :: 'a ⇒ 'b
  assumes indep1: ∀x x'. x ≠ x' ⇒ f x s x' = s x'
  assumes indep2: ∀x x'. x ≠ x' ⇒ f (f x' s) x = f x s x
  assumes [simp]: distinct xs
  shows fold f xs s x = (if x ∈ set xs then f x s x else s x)

proof 
  interpret pointwise-op f λs. s 
  by unfold-locales fact+

show ?thesis 
  using pointwise-upd-fold[of xs s x] 
  by auto 
qed

lemma list-prod-divmod-eq: List.product [0..<M] [0..<N] = map (λi. (i div N, i mod N)) [0..<N*M]
proof 
  have [simp]: i < m*n ⇒ (i::nat) div n < m for i m n 
  by (metis mult.commute div-eq-0-iff div-mult2-eq gr-implies-not-zero mult-not-zero)

  have [simp]: i<N*M ⇒ N>0 ∧ M>0 for i 
  by (cases N; cases M; auto) 

show ?thesis 
  by (rule nth-equalityI) (auto simp add: product-nth algebra-simps) 
qed

lemma nfoldli-prod-divmod-conv: 
  nfoldli (List.product [0..<N] [0..<M]) ctd (λ(i,j). f i j) = nfoldli [0..<N*M] 
  ctd (λi. f (i div M) (i mod M)) 
  apply (intro ext)
  apply (subst list-prod-divmod-eq)
  apply (simp add: nfoldli-map)
  apply (fo-rule cong)+
  apply (auto simp: algebra-simps)
  done

lemma nfoldli-prod-divmod-conv': 
  nfoldli [0..<M] ctd (λi. nfoldli [0..<N] ctd (f i)) = nfoldli [0..<N*M] ctd 
  (λi. f (i div N) (i mod N)) 
  apply (intro ext)
  apply (subst nfoldli-nfoldli-prod-conv)
  by (simp add: nfoldli-prod-divmod-conv algebra-simps)

lemma foldli-prod-divmod-conv': 
  foldli [0..<M] ctd (λi. foldli [0..<N] ctd (f i)) = foldli [0..<N*M] ctd (λi. f 
  (i div N) (i mod N)) 
  (is ?lhs=?rhs) 
proof 
  have RETURN (?lhs s) = RETURN (?rhs s) for s 
    apply (subst foldli-eq-nfoldli)+
  apply (subst nfoldli-prod-divmod-conv')

..
thus ?thesis by auto
qed

lemma fold-prod-divmod-conv': fold (λi. fold (f i) [0..<N]) [0..<M] = fold (λi. f (i div N) (i mod N)) [0..<N*M]
  using foldl-prod-diemod-conv[of M λ. True N f, THEN fun-cong]
  apply (intro ext)
  apply (simp add: foldl-foldl foldl-cone-fold)
done

lemma mtx-nonzero-cases[consumes 0, case-names nonzero zero]:
  obtains (i,j)∈mtx-nonzero m | m (i,j) = 0
by (auto simp: mtx-nonzero-def)

Unary Pointwise

definition mtx-pointwise-unop :: (nat×nat ⇒ 'a) ⇒ 'a mtx ⇒ 'a mtx
where
  mtx-pointwise-unop f m ≡ λ(i,j). f (i,j) (m(i,j))

casefixes f :: (nat×nat ⇒ 'a) ⇒ 'a
begin
  sepref-register PR-CONST (mtx-pointwise-unop f) :: 'a i-mtx ⇒ 'a i-mtx
  lemma [def-pat-rules]: mtx-pointwise-unop f m = opr-fold-impl m
  apply (rule ext)
  unfolding opr-fold-impl_def
  apply (simp add: fold-fold-prod-conv)
  apply (subst pointwise-fun-fold)
  apply (auto simp: mtx-pointwise-unop_def distinct-product [3])
  clarsimp
  subgoal for a b
    apply (cases a b m rule: mtx-nonzero-cases)
    using assms
    apply (auto simp: mtx-pointwise-unop-def)
end

locale mtx-pointwise-unop-loc =
  fixes N :: nat and M :: nat
  fixes f :: (nat×nat ⇒ 'a) ⇒ 'a
  assumes pres-zero|simp: \[ i\geq N \lor j\geq M \] =⇒ f (i,j) 0 = 0
begin
  definition opr-fold-impl-eq:
  assumes mtx-nonzero m ⊆ {0..<N}×{0..<M}
  shows mtx-pointwise-unop f m = opr-fold-impl m
  apply (rule ext)
unfolding opr-fold-impl_def
apply (simp add: fold-fold-prod-conv)
apply (subst pointwise-fun-fold)
apply (auto simp: mtx-pointwise-unop-def distinct-product [3])
apply clarsimp
subgoal for a b
  apply (cases a b m rule: mtx-nonzero-cases)
  using assms
  apply (auto simp: mtx-pointwise-unop-def)
end

347
done
done

lemma opr-fold-impl-refine: (opr-fold-impl, mtx-pointwise-unop f) ∈ [λm. mtx-nonzero m ⊆ {0..<N}×{0..<M}]/Id → Id
  apply (rule refl)
  using opr-fold-impl-eq
  by auto

end

locale mtx-pointwise-unop-gen-impl = mtx-pointwise-unop-loc +
  fixes assn :: 'a mtx ⇒ i ⇒ assn
  fixes A :: 'a ⇒ 'ai ⇒ assn
  fixes get-impl :: 'i ⇒ nat×nat ⇒ 'ai Heap
  fixes set-impl :: 'i ⇒ nat×nat ⇒ 'ai ⇒ 'i Heap
  fixes fi :: nat×nat ⇒ 'ai ⇒ 'ai Heap
  assumes assn-range: rdomp assn m ⇒ mtx-nonzero m ⊆ {0..<N}×{0..<M}
  assumes get-impl-hnr: (uncurry get-impl,uncurry (RETURN oo op-mtx-get)) ∈ assn k *ₐ (prod-assn (nbn-assn N) (nbn-assn M))^k →ₐ A
  assumes set-impl-hnr: (uncurry2 set-impl,uncurry2 (RETURN ooo op-mtx-set)) ∈ assn d *ₐ (prod-assn (nbn-assn N) (nbn-assn M))^k *ₐ A^k →ₐ assn
  assumes fi-hnr: (uncurry fi,uncurry (RETURN oo f)) ∈ (prod-assn nat-assn nat-assn)^k *ₐ A^k →ₐ A
begin

lemma this-loc: mtx-pointwise-unop-gen-impl N M f assn A get-impl set-impl fi
  by unfold-locales

context
  notes [[sepref-register-adhoc f N M]]
  notes [intf-of-assn] = intf-of-assn[where R=assn and 'a='a i-mtx]
  notes [sepref-import-param] = IdI[of N] IdI[of M]
begin

sepref-thm opr-fold-impl1 is RETURN o opr-fold-impl :: assn^d →ₐ assn
  unfolding opr-fold-impl-def
  supply [[goals-limit = 1]]
  by sepref
end

concrete-definition (in –) mtx-pointwise-unop-fold-impl1 uses mtx-pointwise-unop-gen-impl opr-fold-impl

prepare-code-thms (in –) mtx-pointwise-unop-fold-impl1-def

lemma op-hnr[sepref-fr-rules]: (mtx-pointwise-unop-fold-impl1 N M get-impl set-impl fi, RETURN ° PR-CONST (mtx-pointwise-unop f)) ∈ assn^d →ₐ assn

348
unfolding PR-CONST-def
apply (rule hfwf-weaken-pre[of mtx-pointwise-annop-fold-impl1.refine[of this-loc, FCOMP opr-fold-impl-refine]])
by (simp add: assn-range)
end

Binary Pointwise

definition mtx-pointwise-binop :: (′a ⇒ ′a ⇒ ′a) ⇒ ′a mtx ⇒ ′a mtx ⇒ ′a mtx
where
  mtx-pointwise-binop f m n ≡ λ(i,j). f (m(i,j)) (n(i,j))
context fixes f :: ′a ⇒ ′a ⇒ ′a
begin
  sepref-register PR-CONST (mtx-pointwise-binop f) :: ′a i-mtx ⇒ ′a i-mtx
  lemma [def-pat-rules]: mtx-pointwise-binop f ≡ UNPROTECT (mtx-pointwise-binop f) by simp
end
locale mtx-pointwise-binop-loc =
fixes N :: nat and M :: nat
fixes f :: ′a ::{zero}⇒ ′a ⇒ ′a ⇒ ′a
assumes pres-zero[simp]: f 0 0 = 0
begin
  definition opr-fold-impl m n ≡ fold (λi. fold (λj m. (i,j) := f (m(i,j))) (n(i,j))) [0..<M] [0..<N] m

  lemma opr-fold-impl-eq:
  assumes mtx-nonzero m ⊆ {0..<N}×{0..<M}
  assumes mtx-nonzero n ⊆ {0..<N}×{0..<M}
  shows mtx-pointwise-binop f m n = opr-fold-impl m n
  apply (rule ext)
  unfolding opr-fold-impl-def
  apply (simp add: fold-fold-prod-conv)
  apply (subst pointwise-fun-fold)
  apply (auto simp: mtx-pointwise-binop-def distinct-product) [3]
  apply clarsimp
  subgoal for a b
  apply (cases a b m rule: mtx-nonzero-cases; cases a b n rule: mtx-nonzero-cases)
  using assms
  apply (auto simp: mtx-pointwise-binop-def)
  done
  done

  lemma opr-fold-impl-refine: (uncurry opr-fold-impl, uncurry (mtx-pointwise-binop f)) ∈ [λ(m,n). mtx-nonzero m ⊆ {0..<N}×{0..<M} ∧ mtx-nonzero n ⊆ {0..<N}×{0..<M}]f Id×, Id → Id
  apply (rule frefl)
  using opr-fold-impl-eq

349
by auto

end

locale mtx-pointwise-binop-gen-impl = mtx-pointwise-binop-loc +
  fixes assn :: 'a mtx ⇒ 'i ⇒ assn
  fixes A :: 'a ⇒ 'ai ⇒ assn
  fixes get-impl :: 'i ⇒ nat×nat ⇒ 'ai Heap
  fixes set-impl :: 'i ⇒ nat×nat ⇒ 'ai ⇒ 'i Heap
  fixes fi :: 'ai ⇒ 'ai ⇒ 'ai Heap
  assumes assn-range: rdomp assn m ⇒ mtx-nonzero m ⊆ {0..<N}×{0..<M}
  assumes get-impl-hnr: (uncurry get-impl,uncurry (RETURN oo op-mtx-get)) ∈ assn^k *a (prod-assn (nbn-assn N) (nbn-assn M))^k →_a A
  assumes set-impl-hnr: (uncurry2 set-impl,uncurry2 (RETURN ooo op-mtx-set)) ∈ assn^d *a (prod-assn (nbn-assn N) (nbn-assn M))^k *a A^k →_a assn
  assumes fi-hnr: (uncurry f,uncurry (RETURN oo f)) ∈ A^k *a A^k →_a A

begin

lemma this-loc: mtx-pointwise-binop-gen-impl N M f assn A get-impl set-impl fi
  by unfold-locales

context
  notes [[sepref-register-adhoc f N M]]
  notes [intf-of-assn] = intf-of-assn[where R=assn and 'a='a i-mtx]
  notes [sepref-import-param] = IdI[of N] IdI[of M]
begin

sepref-thm opr-fold-impl1 is uncury (RETURN oo opr-fold-impl) :: assn^d *a assn^k →_a assn
  unfolding opr-fold-impl-def[abs-def]
  by sepref

end

concrete-definition (in −) mtx-pointwise-binop-fold-impl1
  uses mtx-pointwise-binop-gen-impl.opr-fold-impl1.refine-raw is (uncurry ?f,−)∈-
  prepare-code-thms (in −) mtx-pointwise-binop-fold-impl1-def

lemma op-hnr[sepref-fr-rules]: (uncurry (mtx-pointwise-binop-fold-impl1 N M get-impl set-impl fi), uncury (RETURN oo PR-CONST (mtx-pointwise-binop f))) ∈ assn^d *a assn^k →_a assn
  unfolding PR-CONST-def
  apply (rule href-weaken-pre[OF - mtx-pointwise-binop-fold-impl1.refine[OF this-loc,FCOMP opr-fold-impl-refine]])
  apply (auto dest: assn-range)
done

end

Compare Pointwise

definition mtx-pointwise-cmpop :: ('a ⇒ 'a ⇒ bool) ⇒ ('a ⇒ 'a ⇒ bool) ⇒ 'a
  mtx ⇒ 'a mtx ⇒ bool where
  mtx-pointwise-cmpop f g m n ≡ (∀ i j. f (m(i,j)) (n(i,j))) ∧ (∃ i j. g (m(i,j)) (n(i,j)))

class fixes f g :: 'a ⇒ 'a ⇒ bool

locale mtx-pointwise-cmpop-loc =
fixes N :: nat
fixes M :: nat
fixes f g :: 'a ⇒ 'a ⇒ bool
assumes pres-zero [simp]: f 0 0 = True g 0 0 = False

begin

definition opr-fold-impl m n ≡ do
  s ← nfoldli (List.product [0..<N] [0..<M]) (λs. s≠2) (λ(i,j) s. do {
    if f (m(i,j)) (n(i,j)) then
      if s=0 then
        if g (m(i,j)) (n(i,j)) then RETURN 1 else RETURN s
      else RETURN s
    else RETURN 2

  });
  RETURN (s=1)

lemma opr-fold-impl-eq:
  assumes mtx-nonzero m ⊆ {0..<N}×{0..<M}
  assumes mtx-nonzero n ⊆ {0..<N}×{0..<M}
  shows opr-fold-impl m n ≤ RETURN (mtx-pointwise-cmpop f g m n)
proof -
  have (∀ i< N. ∀ j< M. f (m (i, j)) (n (i, j))) ⇒ f (m (i, j)) (n (i, j)) for
  i

351
apply \((\text{cases } i < N; \text{ cases } j < M)\)
using \(\text{assms}\) by \((\text{auto simp: mtx-nonzeroD})\)
moreover have \(g (m (i, j)) (n (i, j)) \implies (\exists i < N. \exists j < M. g (m (i, j)) (n (i, j)))\)
for \(i, j\)
apply \((\text{cases } i < N; \text{ cases } j < M)\)
using \(\text{assms}\) by \((\text{auto simp: mtx-nonzeroD})\)
ultimately have \(\text{EQ: mtx-pointwise-cmpop} f g m n\)
\(\iff (\forall i < N. \forall j < M. f (m(i, j)) (n(i, j))) \land (\exists i < N. \exists j < M. g (m(i, j)) (n(i, j)))\)
ungfolding \(\text{mtx-pointwise-cmpop-def}\) by \(\text{meson}\)

have \(\text{aux: List.product} [0..<N] [0..<M] = \text{l1} @ (i, j) \# \text{l2} \implies i < N \land j < M\)
for \(l1 \ i \ j \ l2\)
proof --
  assume \(\text{List.product} [0..<N] [0..<M] = \text{l1} @ (i, j) \# \text{l2}\)
  hence \((i, j) \in \text{set} (\text{List.product} [0..<N] [0..<M])\) by \(\text{simp}\)
  thus \(?\text{thesis}\) by \(\text{simp}\)
qued

show \(?\text{thesis}\)
unfolding \(\text{opr-fold-impl-def}\)
apply \((\text{refine-vec}\ n\text{foldli-rule}[\text{where } I = \lambda l1 - s.\])\)
  \(\text{if } s = 2\) then \(\exists i < N. \exists j < M. \neg f (m(i, j)) (n(i, j))\)
  else \((s = 0 \lor s = 1) \land (\forall (i, j) \in \text{set} l1. f (m(i, j)) (n(i, j))) \land (s = 1 \iff (\exists (i, j) \in \text{set} l1. g (m(i, j)) (n(i, j))))\)
)\napply \((\text{vc-solve dest: aux solve: asm-rl simp: EQ}) [6]\)
apply \((\text{fastforce simp: EQ})\)
done
qed

lemma \(\text{opr-fold-impl-refine}\):
  \((\text{uncurry opr-fold-impl}, \text{uncurry} \{\text{RETURN oo mtx-pointwise-cmpop} f g\})\) \in 
  \([\lambda(m, n). \text{mtx-nonzero} n \subseteq \{0..<N\} \times \{0..<M\} \land \text{mtx-nonzero} n \subseteq \{0..<N\} \times \{0..<M\}] \times \text{Id} \times \text{Id} \rightarrow \langle \text{bool-rel} \rangle \text{nres-rel}\)
apply \((\text{rule frefl})\)
using \(\text{opr-fold-impl-eq}\)
by \((\text{auto intro: nres-relI})\)
end

locale \(\text{mtx-pointwise-cmpop-gen-impl} = \text{mtx-pointwise-cmpop-loc} +\)
  fixes \(\text{assn} :: \text{a mtx ⇒ 'i ⇒ assn}\)
  fixes \(\text{A :: 'a ⇒ 'ai ⇒ assn}\)
  fixes \(\text{get-impl :: 'i ⇒ nat×nat ⇒ 'ai Heap}\)
\begin{verbatim}
fixes fi :: 'ai ⇒ 'ai ⇒ bool Heap
fixes gi :: 'ai ⇒ 'ai ⇒ bool Heap
assumes assn-range: rdomp assn m ⇒ mtx-nonzero m ⊆ {0..<N} × {0..<M}
assumes get-impl-hnr:
  (uncurry get-impl, uncurry (RETURN oo op-mtx-get)) ∈ assn^k *_a (prod-assn
  (nbn-assn N) (nbn-assn M))^k →_a A
assumes fi-hnr:
  (uncurry fi, uncurry (RETURN oo f)) ∈ A^k *_a A^k →_a bool-assn
assumes gi-hnr:
  (uncurry gi, uncurry (RETURN oo g)) ∈ A^k *_a A^k →_a bool-assn

begin

lemma this-loc: mtx-pointwise-cmpop-gen-impl N M f g assn A get-impl fi gi
  by unfold-locales

context
  notes [[sepref-register-adhoc f g N M]]
  notes [[intf-of-assn] = intf-of-assnI[where R=assn and 'a='a i-mtx]
  notes [[sepref-import-param] = IdI[of N] IdI[of M]
  notes [[sepref-fr-rules] = get-impl-hnr fi-hnr gi-hnr

begin

  sepref-thm opr-fold-impl1 is uncurry opr-fold-impl :: assn^d *_a assn^k →_a
  bool-assn
  unfolding opr-fold-impl-def[abs-def] nfoldli-nfoldli-prod-conv[symmetric]
  by sepref

end

concrete-definition (in –) mtx-pointwise-cmpop-fold-impl1
  uses mtx-pointwise-cmpop-gen-impl.opr-fold-impl1.refine-raw
  is (uncurry ?f, -) ∈-
  prepare-code-thms (in –) mtx-pointwise-cmpop-fold-impl1-def

lemma op-hnr[sepref-fr-rules]: (uncurry (mtx-pointwise-cmpop-fold-impl1 N M
  get-impl fi gi), uncurry (RETURN oo PR-CONST (mtx-pointwise-cmpop f g))) ∈
  assn^d *_a assn^k →_a bool-assn
  unfolding PR-CONST-def
  apply (rule nhref-weaken-pre[\OF - mtx-pointwise-cmpop-fold-impl1.refine[\OF
  this-loc.FCOMP opr-fold-impl-refine]])
  apply (auto dest: assn-range)
  done

end

end

3.14 Matrices by Array (Row-Major)

theory IICF-Array-Matrix
\end{verbatim}
imports ../Intf/IICF-Matrix Separation-Logic-Imperative-HOL.Array-Blit

begin

definition is-amtx \( N M \ c \ mtx \equiv \exists l. \ mtx \mapsto a \uparrow \) 
\[
\begin{aligned}
& \text{length } l = N \cdot M \\
& \land (\forall i < N, \forall j < M. \ ! (i \cdot M + j) = c (i, j)) \\
& \land (\forall i, j. (i \geq N \lor j \geq M) \rightarrow c (i, j) = 0)
\end{aligned}
\]

lemma is-amtx-precise[safe-constraint-rules]: precise \((is-amtx N M)\)
apply rule
unfolding is-amtx-def
apply clarsimp
apply prec-extract-eqs
apply (rule ext)
apply (rename-tac \( x \))
apply (case-tac \( x < N \); case-tac \( x < M \); simp)
done

lemma is-amtx-bounded:
shows rdomp \((is-amtx N M)\) \( m \Rightarrow \) \( \text{mtx-nonzero } m \subseteq \{0..<N\} \times \{0..<M\} \)
unfolding rdomp-def
apply (clarsimp simp: mtx-nonzero-def is-amtx-def)
by (meson not-less)

definition mtx-tabulate \( N M c \equiv \) do 
\[
\begin{aligned}
m \leftarrow \text{Array.new } (N \cdot M) \ 0; \\
(-,-,m) \leftarrow \text{imp-for' } 0 (N \cdot M) (\lambda k (i,j,m). \text{ do } \) \\
\text{Array.upd } k \ (c (i,j)) \ m; \\
\text{let } j = j + 1; \\
\text{if } j < M \text{ then return } (i,j,m) \\
\text{else return } (i+1,0,m) \\
\}) \ (0,0,m); \\
\text{return } m
\end{aligned}
\]

definition amtx-copy \equiv array-copy

definition amtx-dflt \( N M v \equiv \) Array.make \((N \cdot M)\) \((\lambda i. v)\)

definition mtx-get \( M \ mtx e \equiv \) Array.nth \( \text{mtx} (\text{fst } e * M + \text{snd } e)\)
definition mtx-set \( M \ mtx e v \equiv \) Array.upd \((\text{fst } e * M + \text{snd } e)\) \( v \ mtx\)

lemma mtx-idx-valid[simp]: \( [i < (N::nat); j < M] \Rightarrow i \cdot M + j < N \cdot M \)
by (rule mlex-bound)

lemma mtx-idx-unique-conv[simp]:
fixes M :: nat
assumes j < M j' < M
shows (i * M + j = i' * M + j') <-> (i = i' ∧ j = j')
using assms
apply auto
subgoal
by (metis add-right-cancel div-if div-mult-self3 linorder-neqE-nat not-less0)
subgoal
using [j < M; j' < M; i * M + j = i' * M + j'] = \Rightarrow i = i' by auto
done

lemma mtx-tabulate-rl[sep-heap-rules]:
assumes NONZ: mtx-nonzero c ⊆ {0..<N}×{0..<M}
shows <emp> mtx-tabulate N M c <HCF-Array-Matrix.is-amtx N M c>
proof (cases M = 0)
case True thus ?thesis
  unfolding mtx-tabulate-def
  using mtx-nonzeroD[OF - NONZ]
  by (sep-auto simp: is-amtx-def)
next
case False hence M-POS: 0 < M by auto
show ?thesis
  unfolding mtx-tabulate-def
  apply (sep-auto)
  decon: imp-for'-rule[where
  I = \lambda k (i,j,mi). \exists a m. m \mapsto_a m
  * \mapsto (k = i * M + j ∧ j < M ∧ k \leq N * M ∧ length m = N * M )
  * \mapsto (∀ i < i'. ∀ j < M. m!(i' * M + j) = c (i',j) )
  * \mapsto (∀ i < i. m!(i * M + j') = c (i, j') )
  ]
  simp: nth-list-update M-POS dest: Suc-lessI
  )
  unfolding is-amtx-def
  using mtx-nonzeroD[OF - NONZ]
  apply sep-auto
  by (metis add.right-neutral M-POS mtx-idx-unique-conv)
qed

lemma mtx-copy-rl[sep-heap-rules]:
<is-amtx N M c mtx> amtx-copy mtx <λr. is-amtx N M c mtx * is-amtx N M c r>
by (sep-auto simp: amtx-copy-def is-amtx-def)

355
definition PRES-ZERO-UNIQUE A ≡ (A''\{0\} = \{0\}) \wedge A^{-1}''\{0\} = \{0\})
lemma IS-ID-imp-PRES-ZERO-UNIQUE[constraint-rules]: IS-ID A \implies PRES-ZERO-UNIQUE A
  unfolding IS-ID-def PRES-ZERO-UNIQUE-def by auto

definition op-amtx-dfltNxM :: nat \Rightarrow nat \Rightarrow 'a :: zero \Rightarrow nat \times nat \Rightarrow 'a
  where
context fixes N M :: nat begin
sepref-decl-op (no-def) op-amtx-dfltNxM: op-amtx-dfltNxM N M :: A \rightarrow (A) mtx-rel
  where CONSTRAINT PRES-ZERO-UNIQUE A
apply (rule cref-ncl) unfolding op-amtx-dfltNxM-def [abs-def] mtx-rel-def
apply parametricity
by (auto simp add: PRES-ZERO-UNIQUE-def)
end

lemma mtx-dflt-rl [sep-heap-rules]: <emp> amtx-dflt N M k <is-amtx N M
  (op-amtx-dfltNxM N M k)>
  by (sep-auto simp: amtx-dflt-def is-amtx-def)

lemma mtx-get-rl [sep-heap-rules]: [i<N; j<M] \implies <is-amtx N M c mtx>
  mtx-get M mtx (i,j) <\lambda r. is-amtx N M c mtx * \uparrow (r = c (i,j))>
  by (sep-auto simp: mtx-get-def is-amtx-def)

lemma mtx-set-rl [sep-heap-rules]: [i<N; j<M] = \implies <is-amtx N M c mtx>
  mtx-set M mtx (i,j) v <\lambda r. is-amtx N M (c((i,j) := v)) r>
  by (sep-auto simp: mtx-set-def is-amtx-def nth-list-update)

definition amtx-assn N M A ≡ hr-comp (is-amtx N M) ((the-pure A) mtx-rel)
lemmas [fcomp-norm-unfold] = amtx-assn-def [symmetric]
lemmas [safe-constraint-rules] = CN-FALSEI[of is-pure amtx-assn N M A for N M A]

lemma [intf-of-assn]: intf-of-assn A TYPE('a) \implies intf-of-assn (amtx-assn N M A) TYPE('a i-mtx)
  by simp

abbreviation asmtx-assn N A ≡ amtx-assn N N A

lemma mtx-rel-pres-zero:
  assumes PRES-ZERO-UNIQUE A
  assumes (m,m')\in(A) mtx-rel
  shows m i j = 0 \iff m' i j = 0
  using assms
apply1 (clarsimp simp: IS-PURE-def PRES-ZERO-UNIQUE-def is-pure-conv mtx-rel-def)
apply (drule fun-relD) applyS (rule IdI[of i j]) applyS auto
done

356
lemma amtx-assn-bounded:
  assumes CONSTRAINT (IS-PURE PRES-ZERO-UNIQUE) A
  shows rdomp (amtx-assn N M A) m \rightarrow mtx-nonzero m \subseteq \{0..<N\} \times \{0..<M\}
  apply (clarsimp simp: mtx-nonzero-def amtx-assn-def rdomp-hrcmp-cone)
  apply (erule is-amtx-bounded)
  using assms
  by (fastforce simp: IS-PURE-def is-pure-conv mtx-rel-pres-zero[ symmetric] mtx-nonzero-def)

lemma mtx-tabulate-aref:
  (mtx-tabulate N M, RETURN o op-mtx-new) 
  \in [\lambda c. mtx-nonzero c \subseteq \{0..<N\} \times \{0..<M\} \rangle_a id-assn \rightarrow \langle HCF-Array-Matrix.is-amtx \times N M \rangle
  by sepref-to-hoare sep-auto

lemma mtx-copy-aref:
  (amtx-copy, RETURN o op-mtx-copy) \in (is-amtx N M)^k \rightarrow_a is-amtx N M
  apply rule
  apply (sepref-auto simp: pure-def)
  done

lemma mtx-nonzero-bid-eq:
  assumes R \subseteq Id
  assumes (a, a') \in Id \rightarrow R
  shows mtx-nonzero a = mtx-nonzero a'
  using assms
  apply (clarsimp simp: mtx-nonzero-def)
  apply (metis fun-relE2 pair-in-Id-conv subsetCE)
  done

lemma mtx-nonzero-zu-eq:
  assumes PRES-ZERO-UNIQUE R
  assumes (a, a') \in Id \rightarrow R
  shows mtx-nonzero a = mtx-nonzero a'
  using assms
  apply (clarsimp simp: mtx-nonzero-def PRES-ZERO-UNIQUE-def)
  by (metis (no-types, hide-lams) IdI Image-singleton-iff converse-iff singletonD tagged-fun-relD-none)

lemma op-mtx-new-fref' :
  CONSTRAINT PRES-ZERO-UNIQUE A \rightarrow (RETURN \circ op-mtx-new, RETURN \circ op-mtx-new) \in (nat-rel \times_r nat-rel \rightarrow A) \rightarrow_f (\langle A \rangle mtx-rel \langle nres-rel \rangle)
  by (rule op-mtx-new-fref)

  by (auto simp: mtx-nonzero-zu-eq)

definition [simp]: op-amtx-new (N::nat) (M::nat) ≡ op-mtx-new
lemma amtx-fold-custom-new:
op-mtx-new ≡ op-amtx-new N M
mop-mtx-new ≡ λc. RETURN (op-amtx-new N M c)
by (auto simp: mop-mtx-new-alt[abs-def])

context fixes N M :: nat begin
sepref-register PR-CONST (op-amtx-new N M) :: (nat × nat ⇒ 'a) ⇒ 'a
i-mtx
end

lemma amtx-new-hnr[sepref-fr-rules]:
fixes A :: 'a::zero ⇒ 'b::{zero,heap} ⇒ assn
shows CONSTRAINT (IS-PURE PRES-ZERO-UNIQUE) A ⇒
(mtx-tabulate N M, (RETURN o PR-CONST (op-amtx-new N M)))
∈ [x. mtx-nonzero x ⊆ {0..<N} × {0..<M}] (pure (nat-rel × nat-rel → the-pure A))k → amtx-assn N M A
using amtx-new-by-tab-hnr[of A N M] by simp

lemma [def-pat-rules]: op-amtx-new$N$M ≡ UNPROTECT (op-amtx-new N M) by simp

context fixes N M :: nat notes [param] = IdI[of N] IdI[of M] begin

lemma mtx-dflt-aref:
(op-amtx-dfltNxM N M, RETURN o PR-CONST (op-amtx-dfltN M N M)) ∈ id-assn k
→ a is-amtx N M
apply rule apply rule
apply (sep-auto simp: pure-def)
done

lemma amtx-get-aref:
(uncurry (mtx-get M), uncurry (RETURN oo op-mtx-get)) ∈ [λ(·,(i,j)). i<N ∧ j<M] (is-amtx N M)k *a (prod-assn nat-assn nat-assn)k → id-assn
apply rule apply rule
apply (sep-auto simp: pure-def)
done

lemma amtx-set-aref: (uncurry2 (mtx-set M), uncurry2 (RETURN ooo op-mtx-set))
∈ [λ((·,(i,j)),·). i<N ∧ j<M] (is-amtx N M)k *a (prod-assn nat-assn nat-assn)k *a id-assn k → is-amtx N M
apply rule apply (rule hn-refine-preI) apply rule
apply (sep-auto simp: pure-def hn-ctxt-def invalid-assn-def)
done


lemma amtx-get-aref':
  (uncurry (mtx-get M), uncurry (RETURN oo op-mtx-get)) ∈ (is-amtx N M)^k
  *a (prod-assn (pure (nbn-rel N)) (pure (nbn-rel M)))^k → a id-assn
  apply rule apply rule
  apply (sep-auto simp: pure-def IS-PURE-def IS-ID-def)
done

sepref-decl-impl amtx-get': amtx-get-aref' .

lemma amtx-set-aref': (uncurry2 (mtx-set M), uncurry2 (RETURN ooo op-mtx-set))
  ∈ (is-amtx N M)^d *a (prod-assn (pure (nbn-rel N)) (pure (nbn-rel M)))^k *a
  id-assn^k → a is-amtx N M
  apply rule apply (rule hn-refine-preI) apply rule
  apply (sep-auto simp: pure-def IS-PURE-def IS-ID-def)
done

sepref-decl-impl amtx-set': amtx-set-aref' .

end

3.14.1 Pointwise Operations

context
  fixes M N :: nat
begin
  sepref-decl-op amtx-lin-get: λ f i. op-mtx-get f (i div M, i mod M) :: ⟨A⟩ mtx-rel
  → nat-rel → A
  unfolding op-mtx-get-def mtx-rel-def
  by (rule frefl) (parametricity; simp)

  sepref-decl-op amtx-lin-set: λ f i. op-mtx-set f (i div M, i mod M) x ::
  ⟨A⟩ mtx-rel → nat-rel → A → ⟨A⟩ mtx-rel
  unfolding op-mtx-set-def mtx-rel-def
  apply (rule frefl) apply parametricity by simp-all

  lemma op-amtx-lin-get-aref: (uncurry Array.nth, uncurry (RETURN oo PR-CONST
  op-amtx-lin-get)) ∈ [λ(-,i). i < N * M] a (is-amtx N M)^k *a nat-assn^k → id-assn
  apply sepref-to-hoare
  unfolding is-amtx-def
  apply sep-auto
  apply (metis mult.commute div-eq-0-iff div-mult2-eq div-mult-mod-eq mod-less-divisor
  mult-is-0 not-less0)

359
done

sepref-decl-impl amtx-lin-get: op-amtx-lin-get-aref by auto

lemma op-amtx-lin-set-aref: (uncurry2 (λm i x. Array.upd i x m), uncurry2 (RETURN ooo PR-CONST op-amtx-lin-set)) ∈ [λ((i,-),-). i ≤ N * M]a (is-amtx N M)k *a nat-assnk *a id-assnk → is-amtx N M

proof –
have [simp]: i < N * M ⇒ ¬(M ≤ i mod M) for i
  by (cases N = 0 ∨ M = 0) (auto simp add: not-le)
have [simp]: i < N * M ⇒ ¬(N ≤ i div M) for i
  apply (cases N = 0 ∨ M = 0)
  apply (auto simp add: not-le)
  apply (metis mult.commute div-eq-0-iff div-mult2-eq neq0-conv)
done
show ?thesis
  apply sepref-to-hoare
  unfolding is-amtx-def
  by (sep-auto simp: nth-list-update)
qed

end

lemma amtx-fold-lin-get: m (i div M, i mod M) = op-amtx-lin-get M m i by simp
lemma amtx-fold-lin-set: m ((i div M, i mod M) := x) = op-amtx-lin-set M m i x by simp

locale amtx-pointwise-unop-impl = mtx-pointwise-unop-loc +
  fixes A :: 'a ⇒ 'ai::{zero,heap} ⇒ assn
  fixes fi :: nat×nat ⇒ 'ai ⇒ 'ai Heap
  assumes fi-hnr:
    (uncurry fi,uncurry (RETURN oo f)) ∈ (prod-assn nat-assn nat-assn)k *a Ak
  →a A
begin

lemma this-loc: amtx-pointwise-unop-impl N M f A fi by unfold-locales

context
  assumes PURE: CONSTRAINT (IS-PURE PRES-ZERO-UNIQUE) A
begin
  context
    notes [[sepref-register-adhoc f N M]]
    notes [sepref-import-param] = IdI[of N] IdI[of M]
    notes [sepref-fr-rules] = fi-hnr
    notes [safe-constraint-rules] = PURE

360
notes \[\[\text{simp} = \text{algebra-simps}\]
begin
sepref-thm opr-fold-impl1 is \(\text{RETURN } \circ \text{opr-fold-impl} : (\text{amtx-assn } N \times M) \times A \rightarrow \text{amtx-assn } N \times M \times A\)
unfolding opr-fold-impl-def fold-prod-divmod-conv'
apply (rewrite amtx-fold-lin-set)
apply (rewrite \text{in } f - \Pi \text{amtx-fold-lin-get})
by sepref
end
end
concrete-definition (in –) amtx-pointwise-unnop-fold-impl1 uses amtx-pointwise-unnop-impl opr-fold-impl1
prepare-code-thms (in –) amtx-pointwise-unnop-fold-impl1-def

lemma op-hnr[sepref-fr-rules]:
assumes PURE: CONSTRAINT (IS-PURE PRES-ZERO-UNIQUE) A
shows (amtx-pointwise-unnop-fold-impl1 N M fi, RETURN \circ PR-CONST (mtx-pointwise-unop f)) \in (amtx-assn N M A) \rightarrow a amtx-assn N M A
unfolding PR-CONST-def
apply (rule href-weaken-pre[\(\text{OF - amtx-pointwise-unnop-fold-impl1.refine}[\text{OF this-loc PURE,FCOMP opr-fold-impl-refine}]\])
by (simp add: amtx-assn-bounded[\(\text{OF PURE}]\)
end

locale amtx-pointwise-binop-impl = mtx-pointwise-binop-loc +
fixes A :: \('a \Rightarrow \text{assn}\)
fixes fi :: \('ai \Rightarrow \text{Heap}'\)
assumes fi-hnr: (uncurry fi,uncurry (RETURN oo f)) \in A^k \rightarrow a A
begin
lemma this-loc: amtx-pointwise-binop-impl f A fi
by unfold-locales

context
notes \[\text{sepref-register-adhoc } f N M][]
notes \[\text{sepref-import-param} = \text{IdI}[\text{of } N] \text{IdI}[\text{of } M][]
notes \[\text{sepref-fr-rules} = \text{fi-hnr}\]
assumes PURE[\text{safe-constraint-rules}]: CONSTRAINT (IS-PURE PRES-ZERO-UNIQUE) A
notes \[\text{simp} = \text{algebra-simps}\]
begin
sepref-thm opr-fold-impl1 is uncurry (RETURN oo opr-fold-impl) :: (amtx-assn N M A)^d \rightarrow a amtx-assn N M A
unfolding opr-fold-impl-def[abs-def] fold-prod-divmod-conv'
apply (rewrite amtx-fold-lin-set)
apply (rewrite \text{in } f - \Pi \text{amtx-fold-lin-get})
apply (rewrite \text{in } f - \Pi \text{amtx-fold-lin-get})
by sepref
concrete-definition (in −) amtx-pointwise-binop-fold-impl1 for fi N M
uses amtx-pointwise-binop-impl.opr-fold-impl1.refine-raw is (uncurry ?f,−)∈-
prepare-code-thms (in −) amtx-pointwise-binop-fold-impl1-def

lemma op-hnr[sepref-fr-rules]:
  assumes PURE: CONSTRAINT (IS-PURE PRES-ZERO-UNIQUE) A
  shows (uncurry (amtx-pointwise-binop-fold-impl1 fi N M), uncurry (RETURN oo PR-CONST (mtx-pointwise-binop f))) ∈ (amtx-assn N M A)^d *_a (amtx-assn N M A)^k →_a amtx-assn N M A
  unfolding PR-CONST-def
  apply (rule href-weaken-pre[OF - amtx-pointwise-binop-fold-impl1.refine[OF this-loc PURE.FCOMP opr-fold-impl-refine]])
  apply (auto dest: amtx-assn-bounded[OF PURE])
done
end

locale amtx-pointwise-cmpop-impl = mtx-pointwise-cmpop-loc +
fixes A :: 'a ⇒ 'ai::{zero,heap} ⇒ assn
fixes fi :: 'ai ⇒ 'ai ⇒ bool Heap
fixes gi :: 'ai ⇒ 'ai ⇒ bool Heap
assumes fi-hnr:
  (uncurry fi,uncurry (RETURN oo f)) ∈ A^k *_a A^k →_a bool-assn
assumes gi-hnr:
  (uncurry gi,uncurry (RETURN oo g)) ∈ A^k *_a A^k →_a bool-assn
begin

lemma this-loc: amtx-pointwise-cmpop-impl f g A fi gi
  by unfold-locales

context
  notes [[sepref-register-adhoc f g N M]]
  notes [sepref-import-param] = IdI[of N] IdI[of M]
  notes [sepref-fr-rules] = fi-hnr gi-hnr
assumes PURE[safe-constraint-rules]: CONSTRAINT (IS-PURE PRES-ZERO-UNIQUE) A
begin
  sepref-thm opr-fold-impl1 is uncurry opr-fold-impl :: (amtx-assn N M A)^d *_a (amtx-assn N M A)^k →_a bool-assn
  unfolding opr-fold-impl-def[abs-def] nfoldli-prod-divmod-conv
  apply (rewrite in f ⊨ - amtx-fold-lin-get)
  apply (rewrite in f ⊨ amtx-fold-lin-get)
  apply (rewrite in g ⊨ amtx-fold-lin-get)
  apply (rewrite in g ⊨ amtx-fold-lin-get)
  by sepref
end
concrete-definition (in −) amtx-pointwise-cmpop-fold-impl1 for N M fi gi
uses amtx-pointwise-cmpop-impl1.opr-fold-impl1.refine-raw is (uncurry ?f,−)∈-
prepare-code-thms (in −) amtx-pointwise-cmpop-fold-impl1-def

lemma op-hnr(sepref-fr-rules):
  assumes PURE: CONSTRAINT (IS-PURE PRES-ZERO-UNIQUE) A
  shows (uncurry (amtx-pointwise-cmpop-fold-impl1 N M fi gi)). uncurry (RETURN
  oo PR-CONST (amtx-pointwise-cmpop f g))) ∈ (amtx-assn N M A)W *)a (amtx-assn
  N M A)k → bool-assn

  unfolding PR-CONST-def
  apply (rule href-weaken-pre(OF - amtx-pointwise-cmpop-fold-impl1.refine[OF
  this-loc PURE,FCOMP opr-fold-impl-refine]))
  apply (auto dest: amtx-assn-bounded[OF PURE])
  done

end

3.14.2 Regression Test and Usage Example
context begin
To work with a matrix, the dimension should be fixed in a context

context
  fixes N M :: nat
  — We also register the dimension as an operation, such that we can use it like
  a constant
  notes [[sepref-register-adhoc N M]]
  notes [sepref-import-param] = rIdI
  — Finally, we fix a type variable with the required type classes for matrix
  entries
  fixes dummy:: 'a::{times,zero,heap}
begin

First, we implement scalar multiplication with destructive update of the
matrix:

private definition scmul :: 'a ⇒ 'a mtx ⇒ 'a mtx nres where
  scmul x m ≡ nfoldli [0..<N] (λ- . True) (λi m.
  nfoldli [0..<M] (λ- . True) (λj m. do {
    let mij = m(i,j);
    RETURN (m((i,j) := x * mij))
  }) m
) m

After declaration of an implementation for multiplication, refinement is
straightforward. Note that we use the fixed N in the refinement assertions.

private lemma times-param: (( * ),( * ): 'a⇒−) ∈ Id → Id → Id by simp
context
notes \([sepref-import-param] = times-param\)

begin

sepref-definition scmul-impl
  is uncurry scmul \((id-assn^k \ast_a (amtx-assn N M id-assn)^d \rightarrow_a amtx-assn)\)
  unfolding scmul-def[abs-def]
  by sepref
end

Initialization with default value

private definition init-test \(\equiv\) do {
  let m = op-amtx-dfltNxM 10 5 (0::nat);
  RETURN (m(1,2))
}

private sepref-definition init-test-impl is uncurry0 init-test :: unit-assn^k \rightarrow_a nat-assn
  unfolding init-test-def
  by sepref

Initialization from function diagonal is more complicated: First, we have to define the function as a new constant

qualified definition diagonalN k \(\equiv\) \(\lambda(i,j).\) if \(i=j\) \& \(j<N\) then \(k\) else \(0\)

If it carries implicit parameters, we have to wrap it into a PR-CONST tag:

private sepref-register PR-CONST diagonalN
private lemma [def-pat-rules]: HICF-Array-Matrix.diagonalN$N \equiv UNPROTECT diagonalN by simp

Then, we have to implement the constant, where the result assertion must be for a pure function. Note that, due to technical reasons, we need the the-pure in the function type, and the refinement rule to be parameterized over an assertion variable (here \(A\)). Of course, you can constrain \(A\) further, e.g., CONSTRAINT \((IS-PURE IS-ID) A\)

private lemma diagonalN-hnr[sepref-fr-rules]:
  assumes CONSTRAINT \((IS-PURE PRES-ZERO-UNIQUE) A\)
  shows \((\text{return o diagonalN, RETURN o (PR-CONST diagonalN)}) \in A^k \rightarrow_a \text{pure (nat-rel } \times_a \text{ nat-rel } \rightarrow \text{the-pure A)}\)
  using assms
  apply sepref-to-hoare
  apply (sep-auto simp: diagonalN-def is-pure-conv IS-PURE-def PRES-ZERO-UNIQUE-def)
  done

In order to discharge preconditions, we need to prove some auxiliary lemma that non-zero indexes are within range

lemma diagonal-nonzero-ltN[simp]: \((a,b)\in\text{mtx-nonzero (diagonalN k)} \implies a<N \& b<N\)
by (auto simp: mtx-nonzero-def diagonalN-def split: if-split_asm)

**private definition** init-test2 ≡ do {
  ASSERT (N > 2); — Ensure that the coordinate (1,2) is valid
  let m = op-mtx-new (diagonalN (1::int));
  RETURN (m(1,2))
}

**private sepref-definition** init-test2-impl is uncurry0 init-test2 :: unit-assn k → int-assn
  unfolding init-test2-def amtx-fold-custom-new[of N N]
  by sepref

end

**export-code** scmul-impl in SML-imp
end
**hide-const** scmul-impl


**hide-const(open)** is-amtx

end

### 3.15 Sepref Bindings for Imp/HOL Collections

**theory** IICF-Sepl-Binding

**imports**
  Separation-Logic-Imperative-HOL.Imp-Map-Spec
  Separation-Logic-Imperative-HOL.Imp-Set-Spec
  Separation-Logic-Imperative-HOL.Imp-List-Spec
  Separation-Logic-Imperative-HOL.Hash-Map-Impl
  Separation-Logic-Imperative-HOL.Array-Map-Impl
  Separation-Logic-Imperative-HOL.To-List-GA
  Separation-Logic-Imperative-HOL.Hash-Set-Impl
  Separation-Logic-Imperative-HOL.Array-Set-Impl
  Separation-Logic-Imperative-HOL.Open-List
  Separation-Logic-Imperative-HOL.Circ-List

  ../Intf/IICF-Map
  ../Intf/IICF-Set
  ../Intf/IICF-List

  Collections.Locale-Code

begin

365
This theory binds collection data structures from the basic collection framework established in AFP/Separation-Logic-Imperative-HOL for usage with Sepref.

locale imp-map-contains-key = imp-map +
    constrains is-map :: ('k ⇒ 'v) ⇒ 'm ⇒ assn
    fixes contains-key :: 'k ⇒ 'm ⇒ bool Heap
    assumes contains-key-rule[sep-heap-rules]:
    <is-map m p> contains-key k p <λr. is-map m p * ↑(r⟵k∈dom m)>1

locale gen-contains-key-by-lookup = imp-map-lookup
begin
  definition contains-key k m ≡ do { r ← lookup k m; return (¬is-None r) }

  sublocale imp-map-contains-key is-map contains-key
    apply unfold-locales
    unfolding contains-key-def
    apply (sep-auto split: option.splits)
  done

locale imp-list-tail = imp-list +
    constrains is-list :: 'a list ⇒ 'l ⇒ assn
    fixes tail :: 'l ⇒ 'l Heap
    assumes tail-rule[sep-heap-rules]:
    l≠[] ⇒ <is-list l p> tail p <is-list (tl l)>1

definition os-head :: 'a::heap os-list ⇒ ('a) Heap where
  os-head p ≡ case p of
    None ⇒ raise STR "os-head: Empty list"
  | Some p ⇒ do { m ←!p; return (val m) }

primrec os-tl :: 'a::heap os-list ⇒ ('a os-list) Heap where
  os-tl None = raise STR "os-tl: Empty list"
  | os-tl (Some p) = do { m ←!p; return (next m) }

interpretation os: imp-list-head os-list os-head
  by unfold-locales (sep-auto simp: os-head-def neq-Nil-conv)

interpretation os: imp-list-tail os-list os-tl
  by unfold-locales (sep-auto simp: os-tl-def neq-Nil-conv)

definition cs-is-empty :: 'a::heap cs-list ⇒ bool Heap where
  cs-is-empty p ≡ return (is-None p)
interpretation cs: imp-list-is-empty cs-list cs-is-empty
by unfold-locales (sep-auto simp: cs-is-empty-def split: option.splits)

definition cs-head :: 'a::heap cs-list ⇒ 'a Heap where
  cs-head p ≡ case p of
    None ⇒ raise STR "cs-head: Empty list"
  | Some p ⇒ do { n ← !p; return (val n)}
interpretation cs: imp-list-head cs-list cs-head
by unfold-locales (sep-auto simp: neq-Nil-conv cs-head-def)

definition cs-tail :: 'a::heap cs-list ⇒ 'a cs-list Heap where
  cs-tail p ≡ do { (-,r) ← cs-pop p; return r }
interpretation cs: imp-list-tail cs-list cs-tail
by unfold-locales (sep-auto simp: cs-tail-def)

lemma is-hashmap-finite[simp]: h | is-hashmap m mi ⇒ finite (dom m)
unfolding is-hashmap-def is-hashmap'-def by auto

lemma is-hashset-finite[simp]: h | is-hashset s si ⇒ finite s
unfolding is-hashset-def by (auto dest: is-hashmap-finite)

definition ias-is-it s a si ≡ λ(a′,i). ∃_l. a→_a l * ↑(a′=a ∧ s=ias-of-list l ∧ (i=length l ∧ si={}) ∨ i<length l ∧ i∈s ∧ si=s ∩ {x. x≥i})

context begin
private function first-memb where
  first-memb lmax a i = do { if i<lmax then do { x ← Array.nth a i; if x then return i else first-memb lmax a (Suc i) } else return i }
by pat-completeness auto
termination by (relation measure (λ(l,-,i). l−i)) auto
declare first-memb.simps[simp del]

private lemma first-memb-rl-aux:
  assumes lmax ≤ length l i≤lmax
  shows  < a →o l >
proof (induction lmax a i rule: first-memb.induct)
  case (1 lmax a i)
  show ?case
  apply (subst first-memb.simps)
  using 1.prems
  apply (sep-auto heap: 1.IH; ((sep-auto; fail) | metis eq-iff not-less-eq-eq))
  done
qed

private lemma first-memb-rl[sep-heap-rules]:
  assumes lmax ≤ length l i ≤ lmax
  shows < a ↦→ a l > first-memb lmax a i
  using assms
  by (sep-auto simp: ias-of-list-def heap: first-memb-rl-aux)

definition ias-it-init a = do {
  l ← Array.len a;
  i ← first-memb l a 0;
  return (a, i)
}

definition ias-it-has-next ≡ λ(a, i). do {
  l ← Array.len a;
  return (i < l)
}

definition ias-it-next ≡ λ(a, i). do {
  l ← Array.len a;
  i' ← first-memb l a (Suc i);
  return (i, (a, i'))
}

lemma ias-of-list-bound: ias-of-list l ⊆ {0..<length l} by (auto simp: ias-of-list-def)
end

interpretation ias: imp-set-iterate is-ias ias-is-it ias-it-init ias-it-has-next ias-it-next
  apply unfold-locales
  unfolding is-ias-def ias-is-it-def
  unfolding ias-it-init-def using ias-of-list-bound
  apply (sep-auto)
unfolding ias-it-next-def using ias-of-list-bound
apply (sep-auto; fastforce)
unfolding ias-it-has-next-def
apply sep-auto
apply sep-auto
done

lemma ias-of-list-finite[simp, intro!]: finite (ias-of-list l)
using finite-subset[OF ias-of-list-bound] by auto

lemma is-ias-finite[simp]: h | is-ias S x ⟹ finite S
unfolding is-ias-def by auto

lemma to-list-ga-rec-rule:
  assumes imp-set-iterate is-set is-it it-init it-has-next it-next
  assumes imp-list-prepend is-list l-prepend
  assumes FIN: finite it
  assumes DIS: distinct l set l ∩ it = {}
shows
  < is-it s si iti ∗ is-list l li >
  to-list-ga-rec it-has-next it-next l-prepend iti li
  < λr. ∃A′. is-set s si
  ∗ is-list l′ r
  ∗ ↑(distinct l′ ∧ set l′ = set l∪it) >;
proof –
  interpret imp-set-iterate is-set is-it it-init it-has-next it-next
  + imp-list-prepend is-list l-prepend
  by fact+

from FIN DIS show ?thesis
proof (induction arbitrary: l li iti rule: finite-psubset-induct)
  case (psubset it)
  show ?case
   apply (subst to-list-ga-rec.simps)
   using psubset.prems apply (sep-auto heap: psubset.IH)
   apply (rule ent-frame-fwd[OF quit-iteration])
   apply frame-inference
   apply solve-entails
  done
qed

lemma to-list-ga-rule:
  assumes IT: imp-set-iterate is-set is-it it-init it-has-next it-next
  assumes EM: imp-list-empty is-list l-empty
  assumes PREP: imp-list-prepend is-list l-prepend
  assumes FIN: finite s
shows
\begin{verbatim}
<is-set s si>
to-list-ga it-init it-has-next it-next
l-empty l-prepend si
<λr. ∃A l. is-set s r * is-list l r * true * ↑(distinct l ∧ set l = s)>
proof -
interpret imp-list-empty is-list l-empty +
  imp-set-iterate is-set it-init it-has-next it-next
by fact+

note [sep-heap-rules] = to-list-ga-rec-rule[OF IT PREP]
show ?thesis
unfolding to-list-ga-def
by (sep-auto simp: FIN)
qed

3.15.1 Binding Locales

method solve-sepl-binding = ( unfold-locales;
  (unfold option-assn-pure-conv)?;
  sep-auto
  intro!: hfreqI hn-refineI[THEN hn-refine-preI]
  simp: invalid-assn-def hn-ctxt-def pure-def
)

Map

locale bind-map = imp-map is-map for is-map :: ('ki → 'vi) ⇒ 'm ⇒ assn
begin
  definition assn K V ≡ hr-comp is-map ((the-pure K, the-pure V) map-rel)
  lemmas [fcomp-norm-unfold] = assn-def[symmetric]
  lemmas [safe-constraint-rules] = CN-FALSEI[of is-pure assn K V for K V]
end

locale bind-map-empty = imp-map-empty + bind-map
begin
  lemma empty-hnr-aux: (uncurry0 empty, uncurry0 (RETURN op-map-empty))
  ∈ unit-assn^k ⇒ a is-map
  by solve-sepl-binding

  sepref-decl-impl (no-register) empty: empty-hnr-aux .
end

locale bind-map-is-empty = imp-map-is-empty + bind-map
begin
  lemma is-empty-hnr-aux: (is-empty, RETURN op-map-is-empty) ∈ is-map^k
  ⇒ a bool-assn
  by solve-sepl-binding

end
\end{verbatim}
locale bind-map-update = imp-map-update + bind-map
begin
lemma update-hnr-aux: (uncurry2 update, uncurry2 (RETURN ooo op-map-update)) ∈ id-assn * a id-assn * a is-map → a is-map
  by solve-sepl-binding

end

locale bind-map-delete = imp-map-delete + bind-map
begin
lemma delete-hnr-aux: (uncurry delete, uncurry (RETURN oo op-map-delete)) ∈ id-assn * a is-map → a is-map
  by solve-sepl-binding

locale sepref-decl-impl delete: delete-hnr-aux .
end

locale bind-map-lookup = imp-map-lookup + bind-map
begin
lemma lookup-hnr-aux: (uncurry lookup, uncurry (RETURN oo op-map-lookup)) ∈ id-assn * a is-map → a id-assn
  by solve-sepl-binding

end

locale bind-map-contains-key = imp-map-contains-key + bind-map
begin
lemma contains-key-hnr-aux: (uncurry contains-key, uncurry (RETURN oo op-map-contains-key)) ∈ id-assn * a is-map → a bool-assn
  by solve-sepl-binding

locale sepref-decl-impl contains-key: contains-key-hnr-aux .
end

Set
locale bind-set = imp-set is-set for is-set :: ('ai set) ⇒ 'm ⇒ assn +
  fixes A :: 'a ⇒ 'ai ⇒ assn
begin
definition assn ≡ hr-comp is-set ((the-pure A) set-rel)
lemmas [safe-constraint-rules] = CN-\text{FALSE}i \[\text{of is-pure \text{assn}}\]
end

locale bind-set-setup = bind-set
begin

lemmas [fcomp-norm-unfold] = \text{assn-def}[\text{symmetric}]
lemma APA: \([\text{PROP Q}; \text{CONSTRAINT is-pure A}] \Rightarrow \text{PROP Q} \).
lemma APAdu: \([\text{PROP Q}; \text{CONSTRAINT (IS-PURE IS-LEFT-UNIQUE) A}] \Rightarrow \text{PROP Q} \).
lemma APAru: \([\text{PROP Q}; \text{CONSTRAINT (IS-PURE IS-RIGHT-UNIQUE) A}] \Rightarrow \text{PROP Q} \).
lemma APAbu: \([\text{PROP Q}; \text{CONSTRAINT (IS-PURE IS-LEFT-UNIQUE) A}; \text{CONSTRAINT (IS-PURE IS-RIGHT-UNIQUE) A}] \Rightarrow \text{PROP Q} \).

end

locale bind-set-empty = imp-set-empty + bind-set
begin

lemma hnr-empty-aux: \((\text{uncurry0 empty, uncurry0 (RETURN op-set-empty)}) \in \text{unit-assn}^k\) \to_a \text{is-set}
by solve-sepl-binding

interpretation bind-set-setup by standard

lemmas hnr-op-empty = hnr-empty-aux[FCOMP op-set-empty.fref[\text{where} A=\text{the-pure A}]]
lemmas hnr-mop-empty = hnr-op-empty[FCOMP mk-mop-rl0-np[\text{OF mop-set-empty-alt}]]
end

locale bind-set-is-empty = imp-set-is-empty + bind-set
begin

lemma hnr-is-empty-aux: \((\text{is-empty, RETURN o op-set-is-empty}) \in \text{is-set}^k \to_a \text{bool-assn}\)
by solve-sepl-binding

interpretation bind-set-setup by standard

lemmas hnr-op-is-empty[sepref-fr-rules] = hnr-is-empty-aux[THEN APA,FCOMP op-set-is-empty.fref[\text{where} A=\text{the-pure A}]]
lemmas hnr-mop-is-empty[sepref-fr-rules] = hnr-op-is-empty[FCOMP mk-mop-rl1-np[\text{OF mop-set-is-empty-alt}]]
end

locale bind-set-member = imp-set-memb + bind-set
begin

lemma hnr-member-aux: \((\text{uncurry memb, uncurry (RETURN oo op-set-member)}) \in \text{id-assn}^k\)
\(*_a \text{is-set}^k \to_a \text{bool-assn}\)
by solve-sepl-binding
interpretation bind-set-setup by standard
lemmas hnr-op-member[sepref-fr-rules] = hnr-member-aux[THEN APAbu,FCOMP
op-set-member.fref[where A=the-pure A]]
lemmas hnr-mop-member[sepref-fr-rules] = hnr-member[FCOMP mk-mop-rl2-np[OF
mop-set-member-alt]]
end
locale bind-set-insert = imp-set-ins + bind-set
begin
lemma hnr-insert-aux: (uncurry ins, uncurry (RETURN oo op-set-insert))∈id-assnk
  *a is-setd →a is-set
  by solve-sept-binding
interpretation bind-set-setup by standard
op-set-insert.fref[where A=the-pure A]]
mop-set-insert-alt]]
end
locale bind-set-delete = imp-set-delete + bind-set
begin
lemma hnr-delete-aux: (uncurry delete, uncurry (RETURN oo op-set-delete))∈id-assnk
  *a is-setd →a is-set
  by solve-sept-binding
interpretation bind-set-setup by standard
lemmas hnr-op-delete[sepref-fr-rules] = hnr-delete-aux[THEN APAru,FCOMP
op-set-delete.fref[where A=the-pure A]]
mop-set-delete-alt]]
end
primrec sorted-wrt' where
  sorted-wrt' R [] ←→ True
  | sorted-wrt' R (x#xs) ←→ list-all (R x) xs ∧ sorted-wrt' R xs
lemma sorted-wrt'-eq: sorted-wrt' = sorted-wrt
proof (intro ext iffI)
  fix R :: 'a ⇒ 'a ⇒ bool and xs :: 'a list
  { assume sorted-wrt R xs
    thus sorted-wrt' R xs
    by (induction xs)(auto simp: list-all-iff sorted-sorted-wrt[symmetric])
  }
  { assume sorted-wrt' R xs
    thus sorted-wrt R xs
  }
373
by (induction xs) (auto simp: list-all-iff) 

qed

lemma param-sorted-wrt[param]: (sorted-wrt, sorted-wrt) ∈ (A → A → bool-rel) → (A)list-rel → bool-rel

unfolding sorted-wrt'-eq[symmetric] sorted-wrt'-def by parametricity

lemma obtain-list-from-setrel:

assumes SV: single-valued A

assumes (set l, s) ∈ ⟨A⟩set-rel

obtains m where s = set m (l, m)∈⟨A⟩list-rel

using assms(2)

proof (induction l arbitrary: s thesis)

next

next

next

next

next

next

next

next

next

next

next

next

next
subgoal
apply (rule set-relI)
subgoal using False X1 by fastforce
subgoal using IS-RIGHT-UNIQUE SV X0(2) X2 by fastforce
done
done
qed
qed
moreover from Cons.IH[OF - ((set l,s')∈(A)set-rel] obtain m where s' = set m (l,m)∈(A)list-rel .
ultimately show thesis
apply –
apply (rule Cons.prems(1)[of y≠m])
by auto
qed

lemma param-it-to-sorted-list[params]: [IS-LEFT-UNIQUE A; IS-RIGHT-UNIQUE A] ⇒ (it-to-sorted-list, it-to-sorted-list) ∈ (A → A → bool-rel) → (A)set-rel → ((A)list-rel)nres-rel
unfolding it-to-sorted-list-def[abs-def]
apply (auto simp: it-to-sorted-list-def pw-nres-rel-iff refine-pw-simps)
apply (rule obtain-list-from-setrel; assumption?; clarsimp)
apply (intro exI conjI; assumption?)
using param-distinct[params] apply blast
apply simp
using param-sorted-wrt[params] apply blast
done

locale bind-set-iterate = imp-set-iterate + bind-set +
assumes is-set-finite: h |⇒ is-set S x ⇒ finite S
begin
context begin
private lemma is-imp-set-iterate: imp-set-iterate is-set is-it it-init it-has-next it-next by unfold-locales
private lemma is-imp-list-empty: imp-list-empty (list-assn id-assn) (return [])
apply unfold-locales
apply solve-constraint
apply sep-auto
done

private lemma is-imp-list-prepend: imp-list-prepend (list-assn id-assn) (return oo List.Cons)
apply unfold-locales
apply solve-constraint
apply (sep-auto simp: pure-def)
done

definition to-list ≡ to-list-ga it-init it-has-next it-next (return []) (return oo
List.Cons)

private lemmas tl-rl = to-list-ga-rule[OF is-imp-set-iterate is-imp-list-empty
is-imp-list-prepend, folded to-list-def]

private lemma to-list-sorted1: (to-list,PR-CONST (it-to-sorted-list (λ- -. True))) ∈ is-set k → a list-assn id-assn

unfolding PR-CONST-def
apply (intro hfrefI)
apply (rule hn-refine-preI)
apply (rule hn-refineI)
unfolding it-to-sorted-list-def
apply (sep-auto intro: hfrefI hn-refineI intro: is-set-finite heap: tl-rl)
done

private lemma to-list-sorted2: []
CONSTRAINT (IS-PURE IS-LEFT-UNIQUE) A;
CONSTRAINT (IS-PURE IS-RIGHT-UNIQUE) A] "⇒
(PR-CONST (it-to-sorted-list (λ- -. True)), PR-CONST (it-to-sorted-list
(λ- -. True))) ∈ (the-pure A) set-rel → ((the-pure A) list-rel) nres-rel

unfolding PR-CONST-def CONSTRAINT-def IS-PURE-def
by clarify parametricity

lemmas to-list-hnr = to-list-sorted1[FCOMP to-list-sorted2, folded assn-def]

lemmas to-list-is-to-sorted-list = IS-TO-SORTED-LISTI[OF to-list-hnr]
lemma to-list-gen[sepref-gen-algo-rules]: [CONSTRAINT (IS-PURE IS-LEFT-UNIQUE)
A; CONSTRAINT (IS-PURE IS-RIGHT-UNIQUE) A]]
"⇒ GEN-ALGO to-list (IS-TO-SORTED-LIST (λ- -. True) (bind-set.assn
is-set A) A)
by (simp add: GEN-ALGO-def to-list-is-to-sorted-list)

end
end

List
locale bind-list = imp-list is-list for is-list :: (’ai list) ⇒ ’m ⇒ assn +
fixes A :: ’a ⇒ ’ai ⇒ assn
begin

definition assn ≡ hr-comp is-list ((the-pure A)list-rel)
lemmas [safe-constraint-rules] = CN-FALSEI[af is-pure assn]
end

376
locale bind-list-empty = imp-list-empty + bind-list
begin
  lemma hnr-aux: (uncurry0 empty, uncurry0 (RETURN op-list-empty)) ∈ (pure unit-rel)^k →ₜ a is-list
    apply rule apply rule apply (sep-auto simp: pure-def) done

  lemmas hnr = hnr-aux[FCOMP op-list-empty.fref[of the-pure A], folded assn-def]
  lemmas hnr-mop = hnr[FCOMP mk-mop-rl0-np[OF mop-list-empty-alt]]
end

locale bind-list-is-empty = imp-list-is-empty + bind-list
begin
  lemma hnr-aux: (is-empty, RETURN o op-list-is-empty) ∈ (is-list)^k →ₜ a is-list
    apply rule apply rule apply (sep-auto simp: pure-def) done

  lemmas hnr = hnr-aux[FCOMP op-list-is-empty.fref, of the-pure A]
  lemmas hnr-mop = hnr[FCOMP mk-mop-rl1-np[OF mop-list-is-empty-alt]]
end

locale bind-list-append = imp-list-append + bind-list
begin
  lemma hnr-aux: (uncurry (swap-args2 append), uncurry (RETURN oo op-list-append))
    ∈ (is-list)^d *ₜ (pure Id)^k →ₜ a is-list
    by solve-sepl-binding

  lemmas hnr = hnr-aux[FCOMP op-list-append.fref, of A]
  lemmas hnr-mop = hnr[FCOMP mk-mop-rl2-np[OF mop-list-append-alt]]
end

locale bind-list-prepend = imp-list-prepend + bind-list
begin
  lemma hnr-aux: (uncurry prepend, uncurry (RETURN oo op-list-prepend))
    ∈ (is-list)^d *ₜ (pure Id)^k →ₜ a is-list
    by solve-sepl-binding

  lemmas hnr = hnr-aux[FCOMP op-list-prepend.fref, of A]
  lemmas hnr-mop = hnr[FCOMP mk-mop-rl2-np[OF mop-list-prepend-alt]]
end

locale bind-list-hd = imp-list-head + bind-list
begin
  lemma hnr-aux: (head, RETURN o op-list-hd)
    ∈ [∀l. l ≠ []]ₜ (is-list)^d → pure Id
    by solve-sepl-binding

  lemmas hnr = hnr-aux[FCOMP op-list-hd.fref, of A]
end
lemmas hnr-mop[sepref-fr-rules] = hnr[FCOMP mk-mop-rl1[OF mop-list-tl-alt]]
end

locale bind-list-tl = imp-list-tail + bind-list
begin
lemma hnr-aux: (tail,RETURN o op-list-tl)
∈[l. l≠[]]a (is-list)₄ → is-list
by solve-sepl-binding

lemmas hnr[sepref-fr-rules] = hnr-aux[FCOMP op-list-tl,fref,of the-pure A, folded assn-def]
lemmas hnr-mop[sepref-fr-rules] = hnr[FCOMP mk-mop-rl1[OF mop-list-tl-alt]]
end

locale bind-list-rotate1 = imp-list-rotate + bind-list
begin
lemma hnr-aux: (rotate,RETURN o op-list-rotate1)
∈(is-list)₄ → a is-list
by solve-sepl-binding

lemmas hnr[sepref-fr-rules] = hnr-aux[FCOMP op-list-rotate1,fref,of the-pure A, folded assn-def]
lemmas hnr-mop[sepref-fr-rules] = hnr[FCOMP mk-mop-rl1-np[OF mop-list-rotate1-alt]]
end

locale bind-list-rev = imp-list-reverse + bind-list
begin
lemma hnr-aux: (reverse,RETURN o op-list-rev)
∈(is-list)₄ → a is-list
by solve-sepl-binding

lemmas hnr[sepref-fr-rules] = hnr-aux[FCOMP op-list-rev,fref,of the-pure A, folded assn-def]
lemmas hnr-mop[sepref-fr-rules] = hnr[FCOMP mk-mop-rl1-np[OF mop-list-rev-alt]]
end

3.15.2 Array Map (iam)

definition op-iam-empty ≡ IICF-Map.op-map-empty
interpretation iam: bind-map-empty is-iam iam-new
by unfold-locales

interpretation iam: map-custom-empty op-iam-empty
by unfold-locales (simp add: op-iam-empty-def)
lemmas [sepref-fr-rules] = iam.empty-hnr[folded op-iam-empty-def]

definition [simp]: op-iam-empty-sz (N::nat) ≡ IICF-Map.op-map-empty
lemma [def-pat-rules]: \( \text{op-iam-empty-sz} N \equiv \text{UNPROTECT} \ (\text{op-iam-empty-sz} N) \)
  by simp

interpretation iam-sz: map-custom-empty PR-CONST (op-iam-empty-sz N)
  apply unfold-locales
  apply (simp)
  done

lemma [sepref-fr-rules]:
  \((\text{uncurry0} \ \text{iam-new}, \ \text{uncurry0} \ \text{RETURN} \ \text{PR-CONST} (\text{op-iam-empty-sz} N))) \in \text{unit-assn}^k \rightarrow_a \text{iam-assn} K V
  using \text{iam-empty-hnr}[\text{of} \ K V] \text{ by simp}

interpretation iam: bind-map-update is-iam Array-Map-Impl.iam-update
  by unfold-locales

interpretation iam: bind-map-delete is-iam Array-Map-Impl.iam-delete
  by unfold-locales

interpretation iam: bind-map-lookup is-iam Array-Map-Impl.iam-lookup
  by unfold-locales

setup Locale-Code.open-block
interpretation iam: gen-contains-key-by-lookup is-iam Array-Map-Impl.iam-lookup
  by unfold-locales
setup Locale-Code.close-block

interpretation iam: bind-map-contains-key is-iam iam.contains-key
  by unfold-locales

3.15.3 Array Set (ias)

definition [simp]: \( \text{op-ias-empty} \equiv \text{op-set-empty} \)
interpretation ias: bind-set-empty is-ias ias-new for A
  by unfold-locales

interpretation ias: set-custom-empty ias-new op-ias-empty
  by unfold-locales simp

definition [simp]: \( \text{op-ias-empty-sz} (N::nat) \equiv \text{op-set-empty} \)
lemma [def-pat-rules]: \( \text{op-ias-empty-sz} N \equiv \text{UNPROTECT} \ (\text{op-ias-empty-sz} N) \)
  by simp

interpretation ias-sz: bind-set-empty is-ias ias-new-sz N for N A
  by unfold-locales

interpretation ias-sz: set-custom-empty ias-new-sz N PR-CONST (op-ias-empty-sz
N) for A
by unfold-locales simp
lemma [sepref-fr-rules]:
(uncurry0 (ias-new-sz N), uncurry0 (RETURN (PR-CONST (op-ias-empty-sz N)))) ∈ unit-assn_k →_a ias-assn A
using ias-sz.hnr-op-empty[of N A] by simp

interpretation ias: bind-set-member is-ias Array-Set-Impl.ias-memb for A
by unfold-locales

interpretation ias: bind-set-insert is-ias Array-Set-Impl.ias-ins for A
by unfold-locales

interpretation ias: bind-set-delete is-ias Array-Set-Impl.ias-delete for A
by unfold-locales

setup Locale-Code.open-block
interpretation ias: bind-set-iterate is-ias-is-it ias-it-init ias-it-has-next ias-it-next
for A
by unfold-locales auto
setup Locale-Code.close-block

3.15.4 Hash Map (hm)

interpretation hm: bind-map-empty is-hashmap hm-new
by unfold-locales

definition op-hm-empty ≡ HCF-Map.op-map-empty
interpretation hm: map-custom-empty op-hm-empty
by unfold-locales (simp add: op-hm-empty-def)
lemmas [sepref-fr-rules] = hm.empty-hnr[folded op-hm-empty-def]

interpretation hm: bind-map-is-empty is-hashmap Hash-Map.hm-isEmpty
by unfold-locales

interpretation hm: bind-map-update is-hashmap Hash-Map.hm-update
by unfold-locales

interpretation hm: bind-map-delete is-hashmap Hash-Map.hm-delete
by unfold-locales

interpretation hm: bind-map-lookup is-hashmap Hash-Map.hm-lookup
by unfold-locales

setup Locale-Code.open-block
interpretation hm: gen-contains-key-lookup is-hashmap Hash-Map.hm-lookup
by unfold-locales
setup Locale-Code.close-block
interpretation hm: bind-map-contains-key is-hashmap hm.contains-key
  by unfold-locales

3.15.5 Hash Set (hs)

interpretation hs: bind-set-empty is-hashset hs-new for A
  by unfold-locales

definition op-hs-empty ≡ IICF-Set.op-set-empty
interpretation hs: set-custom-empty hs-new op-hs-empty for A
  by unfold-locales (simp add: op-hs-empty-def)

interpretation hs: bind-set-is-empty is-hashset Hash-Set-Impl.hs-isEmpty for A
  by unfold-locales

interpretation hs: bind-set-member is-hashset Hash-Set-Impl.hs-memb for A
  by unfold-locales

interpretation hs: bind-set-insert is-hashset Hash-Set-Impl.hs-ins for A
  by unfold-locales

interpretation hs: bind-set-delete is-hashset Hash-Set-Impl.hs-delete for A
  by unfold-locales

setup Locale-Code.open-block
interpretation hs: bind-set-iterate is-hashset hs-is-it hs-it-init hs-it-has-next hs-it-next for A
  by unfold-locales simp
setup Locale-Code.close-block

3.15.6 Open Singly Linked List (osll)

interpretation osll: bind-list os-list for A by unfold-locales
interpretation osll-empty: bind-list-empty os-list os-empty for A
  by unfold-locales

definition osll-empty ≡ op-list-empty
interpretation osll: list-custom-empty osll-assn A os-empty osll-empty
  apply unfold-locales
  apply (rule osll-empty.hnr)
  by (simp add: osll-empty-def)

interpretation osll-is-empty: bind-list-is-empty os-list os-is-empty for A
  by unfold-locales

interpretation osll-prepend: bind-list-prepend os-list os-prepend for A
  by unfold-locales

interpretation osll-hd: bind-list-hd os-list os-head for A

381
by unfold-locales

**interpretation** osll-tl: bind-list-tl os-list os-tl for A
  by unfold-locales

**interpretation** osll-rev: bind-list-rev os-list os-reverse for A
  by unfold-locales

### 3.15.7 Circular Singly Linked List (csll)

**interpretation** csll: bind-list cs-list for A by unfold-locales

**interpretation** csll-empty: bind-list-empty cs-list cs-empty for A
  by unfold-locales

**definition** csll-empty \(\equiv\) op-list-empty

**interpretation** csll: list-custom-empty csll.assn A cs-empty csll-empty
  apply unfold-locales
  apply (rule csll-empty.hnr)
  by (simp add: csll-empty-def)

**interpretation** csll-is-empty: bind-list-is-empty cs-list cs-is-empty for A
  by unfold-locales

**interpretation** csll-prepend: bind-list-prepend cs-list cs-prepend for A
  by unfold-locales

**interpretation** csll-append: bind-list-append cs-list cs-append for A
  by unfold-locales

**interpretation** csll-hd: bind-list-hd cs-list cs-head for A
  by unfold-locales

**interpretation** csll-tl: bind-list-tl cs-list cs-tail for A
  by unfold-locales

**interpretation** csll-rotate1: bind-list-rotate1 cs-list cs-rotate for A
  by unfold-locales

**schematic-goal** hn-refine (emp) (?c::?ec Heap) ?1 ?R (do {
  x \leftarrow mop-list-empty;
  RETURN (1 \in dom\ [1::nat \mapsto True, 2\mapsto False], \{1,2::nat\}, 1\#(2::nat)#x)
})
  apply (subst iam-sz fold-custom-empty[where N=10])
  apply (subst hs fold-custom-empty)
  apply (subst osll fold-custom-empty)
  by sepref

end
3.16 The Imperative Isabelle Collection Framework

theory IICF
imports
  Intf / IICF-Set
  Impl / IICF-List-SetO

  Intf / IICF-Multiset
  Intf / IICF-Prio-Bag

  Impl / IICF-List-Mset
  Impl / IICF-List-MsetO

  Impl / Heaps / IICF-Impl-Heap

  Intf / IICF-Map
  Intf / IICF-Prio-Map

  Impl / Heaps / IICF-Impl-Heapmap

  Intf / IICF-List

  Impl / IICF-Array
  Impl / IICF-HOL-List
  Impl / IICF-Array-List
  Impl / IICF-Indexed-Array-List
  Impl / IICF-MS-Array-List

  Intf / IICF-Matrix

  Impl / IICF-Array-Matrix

  Impl / IICF-Sepl-Binding

begin
  thy-deps
end
Chapter 4

User Guides

This chapter contains the available user guides.

4.1 Quickstart Guide

theory Sepref-Guide-Quickstart
imports ../IICF/IICF
begin

4.1.1 Introduction

Sepref is an Isabelle/HOL tool to semi-automatically synthesize imperative code from abstract specifications.

The synthesis works by replacing operations on abstract data by operations on concrete data, leaving the structure of the program (mostly) unchanged. Seperef proves a refinement theorem, stating the relation between the abstract and generated concrete specification. The concrete specification can then be converted to executable code using the Isabelle/HOL code generator.

This quickstart guide is best appreciated in the Isabelle IDE (currently Isabelle/jedit), such that you can use cross-referencing and see intermediate proof states.

Prerequisites

Sepref is a tool for experienced Isabelle/HOL users. So, this quickstart guide assumes some familiarity with Isabelle/HOL, and will not explain standard Isabelle/HOL techniques.

Sepref is based on Imperative/HOL (HOL—Imperative-HOL. Imperative-HOL) and the Isabelle Refinement Framework (Refine-Monadic.Refine-Monadic). It makes extensive use of the Separation logic formalization for Imperative/HOL (Separation-Logic-Imperative-HOL.Sep-Main).
For a thorough introduction to these tools, we refer to their documentation. However, we try to explain their most basic features when we use them.

4.1.2 First Example

As a first example, let’s compute a minimum value in a non-empty list, wrt. some linear order.

We start by specifying the problem:

\[
\begin{align*}
\text{definition} & \quad \text{min-of-list} :: 'a::linorder list \Rightarrow 'a nres \quad \text{where} \\
& \quad \text{min-of-list} \ l \equiv \text{ASSERT} \ (l \neq []) \gg \text{SPEC} \ (\lambda x. \ \forall y \in \text{set} \ l. \ x \leq y)
\end{align*}
\]

This specification asserts the precondition and then specifies the valid results \(x\). The \(\gg\) operator is a bind-operator on monads.

Note that the Isabelle Refinement Framework works with a set/exception monad over the type - nres, where FAIL is the exception, and RES \(X\) specifies a set \(X\) of possible results. SPEC is just the predicate-version of RES (actually SPEC \(\Phi\) is a syntax abbreviation for SPEC \(\Phi\)).

Thus, \text{min-of-list} will fail if the list is empty, and otherwise nondeterministically return one of the minimal elements.

Abstract Algorithm

Next, we develop an abstract algorithm for the problem. A natural choice for a functional programmer is folding over the list, initializing the fold with the first element.

\[
\begin{align*}
\text{definition} \ & \quad \text{min-of-list1} :: 'a::linorder list \Rightarrow 'a nres \\
& \quad \text{where} \quad \text{min-of-list1} \ l \equiv \text{ASSERT} \ (l \neq []) \gg \text{RETURN} \ (\text{fold} \ \text{min} \ (\text{tl} \ l) \ (\text{hd} \ l))
\end{align*}
\]

Note that RETURN returns exactly one (deterministic) result.

We have to show that our implementation actually refines the specification

\[
\text{lemma} \quad \text{min-of-list1-refine}: (\text{min-of-list1}, \text{min-of-list}) \in \text{Id} \rightarrow (\text{Id}) \text{nres-rel}
\]

This lemma has to be read as follows: If the argument given to \text{min-of-list1} and \text{min-of-list} are related by Id (i.e. are identical), then the result of \text{min-of-list1} is a refinement of the result of \text{min-of-list}, wrt. relation Id.

For an explanation, lets simplify the statement first:

\[
\text{apply} \ (\text{clarsimp intro!}: \text{nres-relI})
\]

The - nres type defines the refinement ordering, which is a lifted subset ordering, with FAIL being the greatest element. This means, that we can assume a non-empty list during the refinement proof (otherwise, the RHS will be FAIL, and the statement becomes trivial)
The Isabelle Refinement Framework provides various techniques to extract verification conditions from given goals, we use the standard VCG here:

\begin{verbatim}
unfolding min-of-list-def min-of-list1-def
apply (refine-vcg)
\end{verbatim}

The VCG leaves us with a standard HOL goal, which is easily provable

\begin{verbatim}
by (auto simp: neq-Nil-conv Min.set-eq-fold[symmetric])
\end{verbatim}

A more concise proof of the same lemma omits the initial simplification, which we only inserted to explain the refinement ordering:

\begin{verbatim}
lemma (min-of-list1,min-of-list) ∈ Id → ⟨Id⟩ nres-rel
unfolding min-of-list-def[abs-def] min-of-list1-def[abs-def]
apply (refine-vcg)
by (auto simp: neq-Nil-conv Min.set-eq-fold[symmetric])
\end{verbatim}

Refined Abstract Algorithm

Now, we have a nice functional implementation. However, we are interested in an imperative implementation. Ultimately, we want to implement the list by an array. Thus, we replace folding over the list by indexing into the list, and also add an index-shift to get rid of the \texttt{hd} and \texttt{tl}.

\begin{verbatim}
definition min-of-list2 :: 'a::linorder list ⇒ 'a nres
where min-of-list2 l ≡ ASSERT (l̸=[]) ⇒ RETURN (fold (λi. min (l!(i+1))))
[0..<length l − 1] (l!0))
\end{verbatim}

Proving refinement is straightforward, using the \texttt{fold-idx-conv} lemma.

\begin{verbatim}
lemma min-of-list2-refine: (min-of-list2, min-of-list1)∈Id → ⟨Id⟩ nres-rel
unfolding min-of-list2-def[abs-def] min-of-list1-def[abs-def]
apply refine-vcg
apply clarsimp-all
apply (rewrite in -=n fold-idx-conv)
by (auto simp: nth-tl hd-conv-nth)
\end{verbatim}

Imperative Algorithm

The version \texttt{min-of-list2} already looks like the desired imperative version, only that we have lists instead of arrays, and would like to replace the folding over \([0..<length l − 1]\) by a for-loop.

This is exactly what the Sepref-tool does. The following command synthesizes an imperative version \texttt{min-of-list3} of the algorithm for natural numbers, which uses an array instead of a list:

\begin{verbatim}
sepref-definition min-of-list3 is min-of-list2 :: (array-assn nat-assn)^k → a nat-assn
unfolding min-of-list2-def[abs-def]
by sepref
\end{verbatim}
The generated constant represents an Imperative/HOL program, and is executable:

- `thm min-of-list3-def`
- `export-code min-of-list3 checking SML-imp`
- `export-code min-of-list3 in SML-imp module-name Min-Of-List`

Also note that the Sepref tool applied a deforestation optimization: It recognizes a fold over \([0..<n]\), and implements it by the tail-recursive function `imp-for'`, which uses a counter instead of an intermediate list.

There are a couple of optimizations, which come in the form of two sets of simplifier rules, which are applied one after the other:

- `thm sepref-opt-simps`
- `thm sepref-opt-simps2`

They are just named theorem collections, e.g., `sepref-opt-simps add/del` can be used to modify them.

Moreover, a refinement theorem is generated, which states the correspondence between `min-of-list3` and `min-of-list2`:

- `thm min-of-list3-refine`

It states the relations between the parameter and the result of the concrete and abstract function. The parameter is related by `array-assn nat-assn`. Here, `array-assn A` relates arrays with lists, such that the elements are related `A` — in our case by `nat-assn`, which relates natural numbers to themselves. We also say that we `implement` lists of nats by arrays of nats. The result is also implemented by natural numbers.

Moreover, the parameters may be stored on the heap, and we have to indicate whether the function keeps them intact or not. Here, we use the annotation `-k` (for `keep`) to indicate that the parameter is kept intact, and `-d` (for `destroy`) to indicate that it is destroyed.

### Overall Correctness Statement

Finally, we can use transitivity of refinement to link our implementation to the specification. The `FCOMP` attribute is able to compose refinement theorems:

- `theorem min-of-list3-correct: (min-of-list3, min-of-list) ∈ (array-assn nat-assn)^k →_a nat-assn`
- `using min-of-list3-refine[FCOMP min-of-list2-refine, FCOMP min-of-list1-refine]`

While the above statement is suited to re-use the algorithm within the sepref-framework, a more low-level correctness theorem can be stated using separation logic. This has the advantage that understanding the statement depends on less definitional overhead:
The proof of this theorem has to unfold the several layers of the Sepref framework, down to the separation logic layer. An explanation of these layers is out of scope of this quickstart guide, we just present some proof techniques that often work. In the best case, the fully automatic proof will work:

\[
\text{lemma } l \neq [] \Rightarrow <\text{array-assn } \text{nat-assn } l \ a > \ \text{min-of-list3 } a <\lambda x. \ \text{array-assn } \text{nat-assn } l \ a + \gamma(\forall y \in \text{set } l. \ x \leq y)>_t
\]

If the automatic method does not work, here is a more explicit proof, that can be adapted for proving similar statements:

\[
\text{lemma } l \neq [] \Rightarrow <\text{array-assn } \text{nat-assn } l \ a > \ \text{min-of-list3 } a <\lambda x. \ \text{array-assn } \text{nat-assn } l \ a + \gamma(\forall y \in \text{set } l. \ x \leq y)>_t
\]

We inlined the definition of \textit{min-of-list}. This will yield two proof obligations later, which we discharge as auxiliary lemmas here:

\[
\text{assume } [\text{simp}]; \ l \neq []
\]

\[
\text{have } [\text{simp}]; \ \text{nofail } (\text{min-of-list } l)
\]

\[
\text{by } (\text{auto simp: min-of-list-def refine-pw-simps})
\]

\[
\text{have } 1: \ \forall x. \ \text{RETURN } x \leq \text{min-of-list } l \ \Rightarrow \ \forall y \in \text{set } l. \ x \leq y
\]

\[
\text{by } (\text{auto simp: min-of-list-def pw-le-iff refine-pw-simps})
\]

\[
\text{note } rl = \text{min-of-list3-correct}[\text{THEN hrefD, of } l \ a, \ \text{THEN hn-refineD, simplified}]
\]

This should yield a Hoare-triple for \textit{min-of-list3 }a, which can now be used to prove the desired statement via a consequence rule

\[
\text{show } ?\text{thesis}
\]

\[
\text{apply } (\text{rule cons-rule}[OF \ - \ - \ rl])
\]

The preconditions should match, however, \textit{sep-auto} is also able to discharge more complicated implications here. Be sure to simplify with \textit{pure-def}, if you have parameters that are not stored on the heap (in our case, we don’t, but include the simplification anyway.)

\[
\text{apply } (\text{sep-auto simp: pure-def})
\]

The heap-parts of the postcondition should also match. The pure parts require the auxiliary statements that we proved above.

\[
\text{apply } (\text{sep-auto simp: pure-def dest!: 1})
\]

\[
\text{done}
\]

\[
\text{qed}
\]
Using the Algorithm

As an example, we now want to use our algorithm to compute the minimum value of some concrete list. In order to use an algorithm, we have to declare both, its abstract version and its implementation to the Sepref tool.

**sepref-register min-of-list**
- This command registers the abstract version, and generates an interface type for it. We will explain interface types later, and only note that, by default, the interface type corresponds to the operation’s HOL type.

**declare min-of-list3-correct[sepref-fr-rules]**
- This declares the implementation to Sepref

Now we can define the abstract version of our example algorithm. We compute the minimum value of pseudo-random lists of a given length

```plaintext
primrec rand-list-aux :: nat ⇒ nat ⇒ nat list where
  rand-list-aux s 0 = []
  | rand-list-aux s (Suc n) = (let s = (1664525 * s + 1013904223) mod 2^32 in s # rand-list-aux s n)

definition rand-list ≡ rand-list-aux 42

definition min-of-rand-list n = min-of-list (rand-list n)
```

And use Sepref to synthesize a concrete version

We use a feature of Sepref to combine imperative and purely functional code, and leave the generation of the list purely functional, then copy it into an array, and invoke our algorithm. We have to declare the `rand-list` operation:

**sepref-register rand-list**

**lemma [sepref-import-param]: (rand-list,rand-list)∈nat-rel → ⟨nat-rel⟩list-rel by auto**

Here, we use a feature of Sepref to import parametricity theorems. Note that the parametricity theorem we provide here is trivial, as `nat-rel` is identity, and `list-rel` as well as `(→)` preserve identity. However, we have to specify a parametricity theorem that reflects the structure of the involved types.

Finally, we can invoke Sepref

**sepref-definition min-of-rand-list1 is min-of-rand-list :: nat-assn^k →_a nat-assn unfolding min-of-rand-list-def[abs-def]**

We construct a plain list, however, the implementation of `min-of-list` expects an array. We have to insert a conversion, which is conveniently done with the `rewrite` method:

```plaintext
apply (rewrite in min-of-list ⊢ array-fold-custom-of-list)
by sepref
```

In the generated code, we see that the pure `rand-list` function is invoked, its result is converted to an array, which is then passed to `min-of-list3`. 
Note that **sepref-definition** prints the generated theorems to the output on the end of the proof. Use the output panel, or hover the mouse over the by-command to see this output.

The generated algorithm can be exported

**export-code min-of-rand-list1 checking SML OCaml? Haskell? Scala**

and executed

**ML-val @\{code min-of-rand-list1\} (\{code nat-of-integer\} 100) ()**

Note that Imperative/HOL for ML generates a function from unit, and applying this function triggers execution.

### 4.1.3 Binary Search Example

As second example, we consider a simple binary search algorithm. We specify the abstract problem, i.e., finding an element in a sorted list.

**definition in-sorted-list x xs ≡ ASSERT (sorted xs) ⩵ RETURN (x∈set xs)**

And give a standard iterative implementation:

**definition in-sorted-list1-invar x xs ≡ λ (l,u,found).**

\[(l \leq u \land u \leq \text{length } xs)\]
\[(\text{found } \rightarrow x \in \text{set } xs)\]
\[(\neg\text{found } \rightarrow (x \notin \text{set } (\text{take } l\ xs) \land x \notin \text{set } (\text{drop } u\ xs))\]

**definition in-sorted-list1 x xs ≡ do**

\[\text{let } l=0;\]
\[\text{let } u=\text{length } xs;\]
\[\langle l,u,r \rangle ← \text{WHILEIT} \ (\text{in-sorted-list1-invar } x\ xs)\]
\[\langle l,u,i,found \rangle. \ l \leq u \land \neg\text{found} \ (\lambda (l,u,found).\ \text{do}\{\]
\[\text{let } i = (l+u) \div 2;\]
\[\text{ASSERT } (i < \text{length } xs); — \text{Added here to help synthesis to prove precondition for array indexing}\]
\[\text{let } xi = xs!i;\]
\[\text{if } x=xi \text{ then }\]
\[\text{RETURN } (l,u,\text{True})\]
\[\text{else if } x<xi \text{ then }\]
\[\text{RETURN } (l,i,\text{False})\]
\[\text{else }\]
\[\text{RETURN } (i+1,u,\text{False})\]
\[\}\rangle (l,u,\text{False});\]
\[\text{RETURN } r\]

Note that we can refine certain operations only if we can prove that their preconditions are matched. For example, we can refine list indexing to array
indexing only if we can prove that the index is in range. This proof has to be done during the synthesis procedure. However, such precondition proofs may be hard, in particular for automatic methods, and we have to do them anyway when proving correct our abstract implementation. Thus, it is a good idea to assert the preconditions in the abstract implementation. This way, they are immediately available during synthesis (recall, when refining an assertion, you may assume the asserted predicate $(\Phi \implies ?M \leq ?M') \implies ?M \leq \text{ASSERT } ?\Phi \leftarrow (\lambda-. ?M')$).

An alternative is to use monadic list operations that already assert their precondition. The advantage is that you cannot forget to assert the precondition, the disadvantage is that the operation is monadic, and thus, nesting it into other operations is more cumbersome. In our case, the operation would be $\text{mop-list-get}$ (Look at it’s simplified definition to get an impression what it does).

\textbf{thm} \texttt{mop-list-get-alt}

We first prove the refinement correct

\textbf{context} begin

\privatelemma \texttt{isl1-measure}: $\text{wf (measure (\lambda(l,u,f). u−l + (if f then 0 else 1))))}$ 
by \simpl

\privatelemma \texttt{neg-nlt-is-gt}:
fixes $a \ b :: 'a::linorder$
shows $a\neq b \implies \neg(a < b) \implies a > b$ by \simpl

\privatelemma \texttt{isl1-aux1}:
assumes $\text{sorted xs}$
assumes $i \ < \ \text{length} \ xs$
assumes $xs!i \ < \ x$
shows $x \notin \text{set (take i xs)}$
using \assms
by (auto \simp: \take-set leD \sorted-nth-mono)

\privatelemma \texttt{isl1-aux2}:
assumes $x \notin \text{set (take n xs)}$
shows $x \notin \text{set (drop n xs)} \iff x \notin \text{set xs}$
apply (rewrite \in \= \text{append-take-drop-id}[of n,symmetric])
using \assms
by (auto \simp: \append-take-drop-id)

\lemma \texttt{in-sorted-list1-refine}: $\text{in-sorted-list1, in-sorted-list} \in \text{Id} \rightarrow \text{Id} \rightarrow \langle \text{Id}\rangle\text{nres-rel}$
unfolding \texttt{in-sorted-list1-def}[\abs-def] \texttt{in-sorted-list-def}[\abs-def]
apply (refine-vcg \texttt{isl1-measure})
apply (vc-solve \simp: \texttt{in-sorted-list1-invar-def} \texttt{isl1-aux1} \texttt{isl1-aux2} solve: \texttt{asm-rl})
apply (auto \simp: \take-set \texttt{set-drop-conv leD \sorted-nth-mono})[]
apply (auto \simp: \take-set leD \sorted-nth-mono \dest: \texttt{neg-nlt-is-gt})[]
done
First, let’s synthesize an implementation where the list elements are natural numbers. We will discuss later how to generalize the implementation for arbitrary types.

For technical reasons, the Sepref tool works with uncurried functions. That is, every function has exactly one argument. You can use the \texttt{uncurry} function, and we also provide abbreviations \texttt{uncurry2} up to \texttt{λf. uncurry2 (uncurry2 (uncurry f))}. If a function has no parameters, \texttt{uncurry0} adds a unit parameter.

\texttt{sepref-definition in-sorted-list2 is uncurry in-sorted-list1 :: nat-assn \*\*nat-assn \texttt{array-assn --\*\*nat-assn \rightarrow\*\*nat-assn \texttt{array-assn \rightarrow\*\*bool-assn}}

\texttt{unfolding in-sorted-list1-def[abs-def] by sepref}

\texttt{export-code in-sorted-list2 checking SML lemmas in-sorted-list2-correct = in-sorted-list2.refine[FCOMP in-sorted-list1-refine]}

4.1.4 Basic Troubleshooting

In this section, we will explain how to investigate problems with the Sepref tool. Most cases where \texttt{sepref} fails are due to some missing operations, unsolvable preconditions, or an odd setup.

Example

We start with an example. Recall the binary search algorithm. This time, we forget to assert the precondition of the indexing operation.

\texttt{definition in-sorted-list1' x xs \equiv do \{ let l=0; let u=length xs; (-,-,r) \leftarrow WHILEIT (in-sorted-list1-invar x xs) (λ(l,u,found). l<u \land \neg found) (λ(l,u,found). do \{ let i = (l+u) div 2; let xi = xs!i; -- It’s not trivial to show that \(i\) is in range if \(x=xi\) then \RETURN (l,u,True) else if \(x<xi\) then \RETURN (l,i,False) else \RETURN (i+1,u,False) \}) (l,u,False); \RETURN r \}}
We try to synthesize the implementation. Note that `sepref-thm` behaves like `sepref-definition`, but actually defines no constant. It only generates a refinement theorem.

**sepref-thm in-sorted-list2** is `uncurry in-sorted-list1′ :: nat-assn ⊕₃ (array-assn nat-assn)ᵏ →ₐ bool-assn`

**unfolding** `in-sorted-list1′-def[abs-def]`

---

— If `sepref` fails, you can use `sepref-dbg-keep` to get some more information.

**apply** `sepref-dbg-keep`

— This prints a trace of the different phases of `sepref`, and stops when the first phase fails. It then returns the internal proof state of the tool, which can be inspected further.

Here, the translation phase fails. The translation phase translates the control structures and operations of the abstract program to their concrete counterparts. To inspect the actual problem, we let translation run until the operation where it fails:

**supply** `[[goals-limit=1]]` — There will be many subgoals during translation, and printing them takes very long with Isabelle :(

**apply** `sepref-dbg-trans-keep`

— Things get stuck at a goal with predicate `hn-refine`. This is the internal refinement predicate, `hn-refine Γ c Γ′ R a` means, that, for operands whose refinement is described by Γ, the concrete program `c` refines the abstract program `a`, such that, afterwards, the operands are described by Γ′, and the results are refined by R.

Inspecting the first subgoal reveals that we got stuck on refining the abstract operation `RETURN $(op-list-get b xf)`. Note that the ($) is just a constant for function application, which is used to tame Isabelle’s higher-order unification algorithms. You may use `unfolding APP-def`, or even `simp` to get a clearer picture of the failed goal.

If a translation step fails, it may be helpful to execute as much of the translation step as possible:

**apply** `sepref-dbg-trans-step-keep`

— The translation step gets stuck at proving `pre-list-get (b, xf)`, which is the precondition for list indexing.

**apply** `(sepref-dbg-side-keep)` — If you think the side-condition should be provable, this command returns the left-over subgoals after some preprocessing and applying auto

**oops**

**Internals of Sepref**

Internally, `sepref` consists of multiple phases that are executed one after the other. Each phase comes with its own debugging method, which only executes that phase. We illustrate this by repeating the refinement of `min-of-list2`. This time, we use `sepref-thm`, which only generates a refinement theorem, but defines no constants:

**sepref-thm min-of-list3′ is min-of-list2 :: (array-assn nat-assn)ᵏ →ₐ nat-assn**
— The `sepref-thm` or `sepref-definition` command assembles a schematic goal statement.

**unfolding** `min-of-list2-def[abs-def]`
— The preprocessing phase converts the goal into the `hn-refine`-form. Moreover, it adds interface type annotations for the parameters. (for now, the interface type is just the HOL type of the parameter, in our case, `nat list`)

**apply** `sepref-dbg-preproc`
— The next phase applies a consequence rule for the postcondition and result. This is mainly for technical reasons.

**apply** `sepref-dbg-cons-init`
— The next phase tries to identify the abstract operations, and inserts tag-constants for function application and abstraction. These tags are for technical reasons, working around Isabelle/HOL’s unifier idiosyncrasies.

Operation identification assigns a single constant to each abstract operation, which is required for technical reasons. Note that there are terms in HOL, which qualify as a single operation, but consists of multiple constants, for example, `{x}`, which is just syntactic sugar for `insert x `{}. In our case, the operation identification phase rewrites the assertion operations followed by a bind to a single operation `op-ASSERT-bind`, and renames some operations to more canonical names.

**apply** `sepref-dbg-id`
— Now that it is clear which operations to execute, we have to specify an execution order. Note that HOL has no notion of execution at all. However, if we want to translate to operations that depend on a heap, we need a notion of execution order. We use the `nres-monad`’s bind operation as sequencing operator, and flatten all nested operations, using left-to-right evaluation order.

**apply** `sepref-dbg-monadify`
— The next step just prepares the optimization phase, which will be executed on the translated program. It just applies the rule $[hn-refine \Gamma \ c \ \Gamma' \ ?R \ ?a;\ CNV \ c \ \Gamma] \Longrightarrow hn-refine \Gamma' \ c' \ \Gamma' \ ?R \ ?a$.

**apply** `sepref-dbg-opt-init`
— The translation phase does the main job of translating the abstract program to the concrete one. It has rules how to translate abstract operations to concrete ones. For technical reasons, it differentiates between operations, which have only first-order arguments (e.g., `length`) and combinators, which have also higher-order arguments (e.g., `fold`).

The basic idea of translation is to repeatedly apply the translation rule for the top-most combinator/operator, and thus recursively translate the whole program. The rules may produce various types of side-conditions, which are resolved by the tool.

**apply** `sepref-dbg-trans`
— The next phase applies some simplification rules to optimize the translated program. It essentially simplifies first with the rules `sepref-opt-simps`, and then with `sepref-opt-simps2`.

**apply** `sepref-dbg-opt`
— The next two phases resolve the consequence rules introduced by the `cons-init` phase.

**apply** `sepref-dbg-cons-solve`
**apply** `sepref-dbg-cons-solve`
— The translation phase and the consequence rule solvers may postpone some side conditions on yet-unknown refinement assertions. These are solved in the last phase.

apply sepref-dbgs-constraints
done

In the next sections, we will explain, by example, how to troubleshoot the various phases of the tool. We will focus on the phases that are most likely to fail.

Initialization
A common mistake is to forget the keep/destroy markers for the refinement assertion, or specify a refinement assertion with a non-matching type. This results in a type-error on the command

sepref-thm test-add-2 is \( \lambda x. \text{RETURN} \ (2 + x) : \text{nat-assn}^k \to_a \text{nat-assn} \)

by sepref

Translation Phase
In most cases, the translation phase will fail. Let’s try the following refinement:

sepref-thm test is \( \lambda l. \text{RETURN} \ (l!1 + 2) : (\text{array-assn nat-assn})^k \to_a \text{nat-assn} \)

The sepref method will just fail. To investigate further, we use sepref-dbgs-keep, which executes the phases until the first one fails. It returns with the proof state before the failed phase, and, moreover, outputs a trace of the phases, such that you can easily see which phase failed.

apply sepref-dbgs-keep
— In the trace, we see that the translation phase failed. We are presented the tool’s internal goal state just before translation. If a phase fails, the usual procedure is to start the phase in debug mode, and see how far it gets. The debug mode of the translation phase stops at the first operation or combinator it cannot translate. Note, it is a good idea to limit the visible goals, as printing goals in Isabelle can be very, very slow :

supply \([\text{goals-limit} = 1]\) apply sepref-dbgs-trans-keep
— Here, we see that translation gets stuck at op-list-get. This may have two reasons: Either there is no rule for this operation, or a side condition cannot be resolved. We apply a single translation step in debug mode, i.e., the translation step is applied as far as possible, leaving unsolved side conditions:

apply sepref-dbgs-trans-step-keep
— This method reports that the ”Apply rule” phase produced a wrong number of subgoals. This phase is expected to solve the goal, but left some unsolved side condition, which we are presented in the goal state. We can either guess what pre-list-get means and why it cannot be solved, or try to partially solve the side condition:
apply sepref-dbg-side-keep
— From the remaining subgoal, one can guess that there might be a problem with
too short lists, where index 1 does not exist.

oops

Inserting an assertion into the abstract program solves the problem:

sepref-thm test is λl. ASSERT (length l > 1) ⇒ RETURN (!!l + 2) :: (array-assn
nat-assn)^k → a nat-assn
by sepref

Here is an example for an unimplemented operation:

sepref-thm test is λl. RETURN (Min (set l)) :: (array-assn nat-assn)^k → a
nat-assn
supply [[goals-limit = 1]]
apply sepref-dbg-keep
apply sepref-dbg-trans-keep
— Translation stops at the set operation
apply sepref-dbg-trans-step-keep
— This tactic reports that the ”Apply rule” phase failed, which means that there
is no applicable rule for the set operation on arrays.

oops

4.1.5 The Isabelle Imperative Collection Framework (IICF)

The IICF provides a library of imperative data structures, and some manage-
ment infrastructure. The main idea is to have interfaces and implementa-
tions.
An interface specifies an abstract data type (e.g., list) and some operations
with preconditions on it (e.g., (@) or (!) with in-range precondition).
An implementation of an interface provides a refinement assertion from the
abstract data type to some concrete data type, as well as implementations
for (a subset of) the interface’s operations. The implementation may add
some more implementation specific preconditions.
The default interfaces of the IICF are in the folder IICF/Intf, and the stand-
ard implementations are in IICF/Impl.

Map Example

Let’s implement a function that maps a finite set to an initial segment of
the natural numbers

definition nat-seg-map s ≡
ASSERT (finite s) ⇒ SPEC (λm. dom m = s ∧ ran m = {0..<card s})

We implement the function by iterating over the set, and building the map
definition nat-seg-map1 s ≡ do 
  ASSERT (finite s);
  (m, i) ← FOREACHi (λit (m, i). dom m = s−it ∧ ran m = {0..<i} ∧ i=card (s−it))
  s (λx (m, i). RETURN (m(x→i),i+1)) (Map.empty,0);
RETURN m 
}

lemma nat-seg-map1-refine: (nat-seg-map1, nat-seg-map) ∈ Id →⟨Id⟩ nres-rel
apply (intro fun-relI)
unfolding nat-seg-map1-def[abs-def] nat-seg-map-def[abs-def]
apply (refine-vcg)
apply (vc-solve simp: it-step-insert-iff solve: asm-rl dest: domD)
done

We use hashsets hs.assn and hashmaps (hm.assn).

sepref-definition nat-seg-map2 is nat-seg-map1 :: (hs.assn id-assn)^k → a hm.assn
id-assn nat-assn
unfolding nat-seg-map1-def[abs-def]
apply sepref-dbg-keep
apply sepref-dbg-trans-keep
— We got stuck at op-map-empty. This is because Sepref is very conservative when
it comes to guessing implementations. Actually, no constructor operation will
be assigned a default operation, with some obvious exceptions for numbers
and Booleans.

oops

Assignment of implementations to constructor operations is done by rewrit-
ing them to synonyms which are bound to a specific implementation. For
hashmaps, we have op-hm-empty, and the rules hm.fold-custom-empty.

sepref-definition nat-seg-map2 is nat-seg-map1 :: (hs.assn id-assn)^k → a hm.assn
id-assn nat-assn
unfolding nat-seg-map1-def[abs-def]
— We can use the rewrite method for position-precise rewriting:
apply (rewrite in FOREACHi - - - ⊢ hm.fold-custom-empty)
by sepref

export-code nat-seg-map2 in SML module-name Test
export-code nat-seg-map2 checking SML

lemmas nat-seg-map2-correct = nat-seg-map2.refine[FCOMP nat-seg-map1-refine]

4.1.6 Specification of Preconditions

In this example, we will discuss how to specify precondition of operations,
which are required for refinement to work. Consider the following function,
which increments all members of a list by one:

definition incr-list l ≡ map ((+ 1)) l
We might want to implement it as follows

**definition incr-list1** \( l \equiv \text{fold} (\lambda i \ l. \ l[i:=1 + l[i]]) [0..<\text{length } l] \ l \)

**lemma incr-list1-refine**: \((\text{incr-list1}, \text{incr-list}) \in \text{Id} \to \text{Id}\)

**proof**

```ml
fix l :: 'a list
{ fix n m
  assume n<=m and length l = m
  hence fold (\lambda i \ l[i:=1+!i]) [n..<m] \ l = take n l @ map (((+))1) (drop n l)
    apply (induction arbitrary: l rule: inc-induct)
    apply simp
    apply (clarsimp simp: upt-conv-Cons take-Suc-conv-app-nth)
    apply (auto simp add: list-eq-iff-nth-eq nth-Cons split: nat.split)
  done
}
from this[of 0 length l] show incr-list1 l = incr-list l
  unfolding incr-list1-def incr-list1-def
  by simp
qed
```

Trying to refine this reveals a problem:

**sepref-thm incr-list2** is \( \text{RETURN o incr-list1} :: (\text{array-assn nat-assn})^d \to_a \text{array-assn nat-assn} \)

**unfolding** incr-list1-def[abs-def]

**apply** separf-dbg-keep

**apply** separf-dbg-trans-keep

**apply** separf-dbg-trans-step-keep

**apply** separf-dbg-side-keep

— We get stuck at the precondition of \( \text{op-list-get} \). Indeed, we cannot prove the generated precondition, as the translation process dropped any information from which we could conclude that the index is in range.

**oops**

Of course, the fold loop has the invariant that the length of the list does not change, and thus, indexing is in range. We only cannot prove it during the automatic synthesis.

Here, the only solution is to do a manual refinement into the nres-monad, and adding an assertion that indexing is always in range.

We use the \( \text{nfoldli} \) combinator, which generalizes \( \text{fold} \) in two directions:

1. The function is inside the nres monad
2. There is a continuation condition. If this is not satisfied, the fold returns immediately, dropping the rest of the list.

**definition incr-list2** \( l \equiv \text{nfoldli} [0..<\text{length } l] \)
(λ· True)
(λl. ASSERT (n<l) ⇒ RETURN (l[i:=l[i+1]]))

Note: Often, it is simpler to prove refinement of the abstract specification, rather than proving refinement to some intermediate specification that may have already done refinements "in the wrong direction". In our case, proving refinement of incr-list1 would require to generalize the statement to keep track of the list-length invariant, while proving refinement of incr-list directly is as easy as proving the original refinement for incr-list1.

lemma incr-list2-refine: (incr-list2, RETURN o incr-list) ∈ Id → (Id)nres-rel

proof (intro nres-relI fun-relI; simp)
fix l :: 'a list
show incr-list2 l ≤ RETURN (incr-list l)
unfolding incr-list2-def incr-list-def
— nfoldli comes with an invariant proof rule. In order to use it, we have to specify the invariant manually:
apply (refine-vcg nfoldli-rule[where I=λl1 l2 s. s=map (((+))1) (take (length l1) l) @ drop (length l1) l])
apply (vc-solve simp: upt-eq-append-conv upt-eq-Cons-conv
simp: nth-append list-update-append upd-conv-take-nth-drop take-Suc-conv-app-nth
solve: asm-rl
)
done
qed

sepref-definition incr-list3 is incr-list2 :: (array-assn nat-assn) → (array-assn nat-assn)
nat-assn
unfolding incr-list2-def[abs-def]
by sepref

lemmas incr-list3-correct = incr-list3.refine[FCOMP incr-list2-refine]

4.1.7 Linearity and Copying

Consider the following implementation of an operation to swap to list indexes. While it is perfectly valid in a functional setting, an imperative implementation has a problem here: Once the update a index i is done, the old value cannot be read from index i any more. We try to implement the list with an array:

sepref-thm swap-nonlinear is uncurry2 (λl i j. do {
ASSERT (n<i length l ∧ j<length l);
RETURN (l[i:=l[j], j:=l[i]])
}) :: (array-assn id-assn) → (array-assn id-assn)
nat-assn
supply [[goals-limit = l]]
apply sepref-dbg-keep
apply sepref-dbg-trans-keep — (1) We get stuck at an op-list-get operation

apply sepref-dbg-trans-step-keep — (2) Further inspection reveals that the "recover pure" phase fails, and we are left with a subgoal of the form CONSTRAINT is-pure (array-assn id-assn). Constraint side conditions are deferrable side conditions: They are produced as side-conditions, and if they cannot be solved immediately, they are deferred and processed later, latest at the end of the synthesis. However, definitely unsolvable constraints are not deferred, but halt the translation phase immediately, and this is what happened here: At (1) we can see that the refinement for the array we want to access is hn-invalid (array-assn id-assn). This means, the data structure was destroyed by some preceding operation. The hn-invalid only keeps a record of this fact. When translating an operation that uses an invalidated parameter, the tool tries to restore the invalidated parameter: This only works if the data structure was purely functional, i.e., not stored on the heap. This is where the is-pure constraint comes from. However, arrays are always stored on the heap, so this constraint is definitely unsolvable, and thus immediately rejected instead of being deferred.

Note: There are scenarios where a constraint gets deferred before it becomes definitely unsolvable. In these cases, you only see the problem after the translation phase, and it may be somewhat tricky to figure out the reason.

oops

The fix for our swap function is quite obvious. Using a temporary storage for the intermediate value, we write:

sepref-thm swap-with-tmp is uncurry2 (λl i j. do {
  ASSERT (i<length l ∧ j<length l);
  let tmp = l!i;
  RETURN (l[i:=l!j, j:=tmp])
}) :: (array-assn id-assn)^d *a nat-assn^k *a nat-assn^k -> a array-assn id-assn
by sepref

Note that also the argument must be marked as destroyed (j)^d. Otherwise, we get a similar error as above, but in a different phase:

sepref-thm swap-with-tmp is uncurry2 (λl i j. do {
  ASSERT (i<length l ∧ j<length l);
  let tmp = l!i;
  RETURN (l[i:=l!j, j:=tmp])
}) :: (array-assn id-assn)^k *a nat-assn^k *a nat-assn^k -> a array-assn id-assn
apply sepref-dbg-keep — We get stuck at a frame, which would require restoring an invalidated array
apply sepref-dbg-cons-solve-keep — Which would only work if arrays were pure

oops

If copying is really required, you have to insert it manually. Reconsider the example incr-list from above. This time, we want to preserve the original data (note the (j)^b annotation):

sepref-thm incr-list3-preserve is incr-list2 :: (array-assn nat-assn)^k -> a array-assn nat-assn

400
unfolding \textit{incr-list2-def[abs-def]}
— We explicitly insert a copy-operation on the list, before it is passed to the fold operation.
\texttt{apply (rewrite in nfoldli - - - \oplus op-list-copy-def[symmetric])}
\texttt{by sepref}

**4.1.8 Nesting of Data Structures**

Sepref and the IICF support nesting of data structures with some limitations:

- Only the container or its elements can be visible at the same time. For example, if you have a product of two arrays, you can either see the two arrays, or the product. An operation like \texttt{snd} would have to destroy the product, loosing the first component. Inside a case distinction, you cannot access the compound object.

These limitations are somewhat relaxed for pure data types, which can always be restored.

- Most IICF data structures only support pure component types. Exceptions are HOL-lists, and the list-based set and multiset implementations \texttt{List-MsetO} and \texttt{List-SetO} (Here, the \texttt{O} stands for \textit{own}, which means that the data-structure owns its elements.).

Works fine:

\begin{verbatim}
sepref-thm product-ex1 is uncurry0 (do {
    let p = (op-array-replicate 5 True, op-array-replicate 2 False);
    case p of (a1,a2) ⇒ RETURN (a1!2)
}) :: unit-assn\textit{k} →_α bool-assn
\texttt{by sepref}
\end{verbatim}

Fails: We cannot access compound type inside case distinction

\begin{verbatim}
sepref-thm product-ex2 is uncurry0 (do {
    let p = (op-array-replicate 5 True, op-array-replicate 2 False);
    case p of (a1,a2) ⇒ RETURN (snd p!1)
}) :: unit-assn\textit{k} →_α bool-assn
\texttt{apply sepref-dbg-keep}
\texttt{apply sepref-dbg-trans-keep}
\texttt{apply sepref-dbg-trans-step-keep}
\texttt{oops}
\end{verbatim}

Works fine, as components of product are pure, such that product can be restored inside case.

\begin{verbatim}
sepref-thm product-ex2 is uncurry0 (do {
    let p = (op-list-replicate 5 True, op-list-replicate 2 False);
    case p of (a1,a2) ⇒ RETURN (snd p!1)
}\end{verbatim}
\[
\text{unit-assn}^k \rightarrow_{a} \text{bool-assn}
\]

by \text{sepref-db-keep}

Trying to create a list of arrays, first attempt:

\text{sepref-thm set-of-arrays-ex is uncurry0 (RETURN (op-list-append [] op-array-empty))} :: \text{unit-assn}^k \rightarrow_a \text{arl-assn (array-assn nat-assn)}

unfolding \text{arl.fold-custom-empty}

apply \text{sepref-db-keep}

apply \text{sepref-db-trans-keep}

apply \text{sepref-db-trans-step-keep}

supply \text{[[goals-limit = 1, unify-trace-failure]]}

— Many IICF data structures, in particular the array based ones, requires the element types to be of \text{default}. If this is not the case, Sepref will simply find no refinement for the operations. Be aware that type-class related mistakes are hard to debug in Isabelle/HOL, above we sketched how to apply the refinement rule that is supposed to match with unifier tracing switched on. The \text{to-hnr} attribute is required to convert the rule from the relational form to the internal \text{hn-refine} form. Note that some rules are already in \text{hn-refine} form, and need not be converted, e.g., \text{hn-refine (hn-ctxt ?P1.0 \ @x1.0 \ @x1' * hn-ctxt ?P2.0 \ @x2.0 \ @x2')} (return (?x1', ?x2')) \text{hn-invalid ?P1.0 \ @x1.0 \ @x1' * hn-invalid ?P2.0 \ @x2.0 \ @x2')} (?P1.0 \times_a ?P2.0) (RETURN $(Pair $ @x1.0 $ @x2.0)).

oops

So lets choose a circular singly linked list (csll), which does not require its elements to be of default type class

\text{sepref-thm set-of-arrays-ex is uncurry0 (RETURN (op-list-append [] op-array-empty))} :: \text{unit-assn}^k \rightarrow_a \text{csll-assn (array-assn nat-assn)}

unfolding \text{csll.fold-custom-empty}

apply \text{sepref-db-keep}

apply \text{sepref-db-trans-keep}

apply \text{sepref-db-trans-step-keep}

— We end up with an unprovable purity-constraint: As many IICF types, csll only supports pure member types. We expect this restriction to be lifted in some future version.

oops

Finally, there are a few data structures that already support nested element types, for example, functional lists:

\text{sepref-thm set-of-arrays-ex is uncurry0 (RETURN (op-list-append [] op-array-empty))} :: \text{unit-assn}^k \rightarrow_a \text{list-assn (array-assn nat-assn)}

unfolding \text{HOL-list.fold-custom-empty}

by \text{sepref}

4.1.9 Fixed-Size Data Structures

For many algorithms, the required size of a data structure is already known, such that it is not necessary to use data structures with dynamic resizing.
The Sepref-tool supports such data structures, however, with some limitations.

Running Example

Assume we want to read a sequence of natural numbers in the range \( \{0::'a..<N\} \), and drop duplicate numbers. The following abstract algorithm may work:

```plaintext
definition remdup l ≡ do 
  (s, r) ← nfoldli l (λ-. True) 
  (λx (s, r), do 
    ASSERT (distinct r ∧ set r ⊆ set l ∧ s = set r); — Will be required to prove 
    that list does not grow too long 
    if x∈s then RETURN (s, r) else RETURN (insert x s, r@[x]) 
  ) 
  ({}[],[]); 
RETURN r
```

We want to use `remdup` in our abstract code, so we have to register it.

```plaintext
sepref-register remdup
```

The straightforward version with dynamic data-structures is:

```plaintext
sepref-definition remdup1 is remdup :: (list-assn nat-assn)^k →_a arl-assn nat-assn

unfolding remdup-def[abs-def]
— Lets use a bit-vector for the set
apply (rewrite in nfoldli - - - π ias.fold-custom-empty)
— And an array-list for the list
apply (rewrite in nfoldli - - - π arl.fold-custom-empty)
by sepref
```

Initializiation of Dynamic Data Structures

Now let’s fix an upper bound for the numbers in the list. Initializations and statically sized data structures must always be fixed variables, they cannot be computed inside the refined program.

TODO: Lift this restriction at least for initialization hints that do not occur in the refinement assertions.

```plaintext
context fixes N :: nat begin

sepref-definition remdup1-initiz is remdup :: (list-assn nat-assn)^k →_a arl-assn nat-assn

unfolding remdup-def[abs-def]
— Many of the dynamic array-based data structures in the IICF can be pre
initialized to a certain size. This initialization is only a hint, and has no
abstract consequences. The list data structure will still be resized if it grows
larger than the initialization size.
```

403
apply (rewrite in nfoldli - - - ias-sz.fold-custom-empty[of N])
apply (rewrite in nfoldli - - - arl-sz.fold-custom-empty[of N])
by sepref

end

To get a usable function, we may add the fixed $N$ as a parameter, effectively converting the initialization hint to a parameter, which, however, has no abstract meaning

definition remdup-initsz ($N$::nat) ≡ remdup
lemma remdup-init-hnr:
    (uncurry remdup1-initsz, uncurry remdup-initsz) ∈ nat-assn $^k_a$ (list-assn nat-assn)$^k$
→ ( distrust arl-assn nat-assn
    using remdup1-initsz.refine unfolding remdup-initsz-def[abs-def]
    unfolding hfref-def hn-refine-def
by (auto simp: pure-def)

Static Data Structures

We use a locale to hide local declarations. Note: This locale will never be interpreted, otherwise all the local setup, that does not make sense outside the locale, would become visible. TODO: This is probably some abuse of locales to emulate complex private setup, including declaration of constants and lemmas.

locale my-remdup-impl-loc =
  fixes $N$::nat
  assumes $N$>0 — This assumption is not necessary, but used to illustrate the general case, where the locale may have such assumptions

begin

For locale hierarchies, the following seems not to be available directly in Isabelle, however, it is useful when transferring stuff between the global theory and the locale

lemma my-remdup-impl-loc-this: my-remdup-impl-loc $N$ by unfold-locales

Note that this will often require to use $N$ as a usual constant, which is refined. For pure refinements, we can use the sepref-import-param attribute, which will convert a parametricity theorem to a rule for Sepref:

sepref-register $N$
lemma $N$-hnr[sepref-import-param]: ($N$,$N$)∈nat-rel by simp
thm $N$-hnr

Alternatively, we could directly prove the following rule, which, however, is more cumbersome:

lemma $N$-hnr': (uncurry0 (return $N$), uncurry0 (RETURN $N$))∈unit-assn $^k_a$ nat-assn
by sepref-to-hoare sep-auto

Next, we use an array-list with a fixed maximum capacity. Note that the capacity is part of the refinement assertion now.

sepref-definition remdup1-fixed is remdup :: (list-assn nat-assn)^k \rightarrow_a marl-assn N nat-assn

unfolding remdup-def[abs-def]
apply (rewrite in nfoldli - - - ias-sz.fold-custom-empty[of N])
apply (rewrite in nfoldli - - - marl-fold-custom-empty-sz[of N])
supply [[goals-limit = 1]]
apply sepref-dbg-keep
apply sepref-dbg-trans-keep
apply sepref-dbg-trans-step-keep

In order to append to the array list, we have to show that the size is not yet exceeded. This may require to add some assertions on the abstract level. We already have added some assertions in the definition of remdup.

oops

Moreover, we add a precondition on the list

sepref-definition remdup1-fixed is remdup :: [\lambda l. \text{set } l \subseteq \{0..<N\}]_a (list-assn nat-assn)^k \rightarrow marl-assn N nat-assn

unfolding remdup-def[abs-def]
apply (rewrite in nfoldli - - - ias-sz.fold-custom-empty[of N])
apply (rewrite in nfoldli - - - marl-fold-custom-empty-sz[of N])
supply [[goals-limit = 1]]
apply sepref-dbg-keep
apply sepref-dbg-trans-keep
apply sepref-dbg-trans-step-keep
apply sepref-dbg-side-keep

We can start from this subgoal to find missing lemmas

oops

We can prove the remaining subgoal, e.g., by auto with the following lemma declared as introduction rule:

lemma aux1[intro]: [ set l \subseteq \{0..<N\}; distinct l ] \implies \text{length } l < N

apply (simp add: distinct-card[symmetric])
apply (drule psubset-card-mono[rotated])
apply auto
done

We use some standard boilerplate to define the constant globally, although being inside the locale. This is required for code-generation.

sepref-thm remdup1-fixed is remdup :: [\lambda l. \text{set } l \subseteq \{0..<N\}]_a (list-assn nat-assn)^k \rightarrow marl-assn N nat-assn

unfolding remdup-def[abs-def]
apply (rewrite in nfoldli - - - ias-sz.fold-custom-empty[of N])
apply (rewrite in nfoldli - - - marl-fold-custom-empty-sz[of N])
by sepref

405
concrete-definition (in −) remdup1-fixed uses my-remdup-impl-loc.remdup1-fixed.refine-raw
is (uncurry ?f,−)∈−

prepare-code-thms (in −) remdup1-fixed-def
lemmas remdup1-fixed-refine[sepref-fr-rules] = remdup1-fixed.refine[OF my-remdup-impl-loc-this]

The concrete-definition command defines the constant globally, without any locale assumptions. For this, it extracts the definition from the theorem, according to the specified pattern. Note, you have to include the uncurrying into the pattern, e.g., (uncurry ?f,−)∈−.

The prepare-code-thms command sets up code equations for recursion combinators that may have been synthesized. This is required as the code generator works with equation systems, while the heap-monad works with fixed-point combinators.

Finally, the third lemma command imports the refinement lemma back into the locale, and registers it as refinement rule for Sepref.

Now, we can refine remdup to remdup1-fixed N inside the locale. The latter is a global constant with an unconditional definition, thus code can be generated for it.

Inside the locale, we can do some more refinements:

definition test-remdup ≡ do {l ← remdup [0..<N]; RETURN (length l) }

Note that the abstract test-remdup is just an abbreviation for test-remdup. Whenever we want Sepref to treat a compound term like a constant, we have to wrap the term into a PR-CONST tag. While sepref-register does this automatically, the PR-CONST has to occur in the refinement rule.

sepref-register test-remdup
sepref-thm test-remdup1 is
  uncurry0 (PR-CONST test-remdup) :: unit-assn^k → a nat-assn
  unfolding test-remdup-def PR-CONST-def
by sepref
concrete-definition (in −) test-remdup1 uses my-remdup-impl-loc.test-remdup1.refine-raw
is (uncurry0 ?f,−)∈−

prepare-code-thms (in −) test-remdup1-def
lemmas test-remdup1-refine[sepref-fr-rules] = test-remdup1.refine[of N]
end

Outside the locale, a refinement of my-remdup-impl-loc.test-remdup also makes sense, however, with an extra argument N.

thm test-remdup1.refine
lemma test-remdup1-refine-aux: (test-remdup1, my-remdup-impl-loc.test-remdup)
∈ [my-remdup-impl-loc]a nat-assn^k → nat-assn
using test-remdup1.refine
unfolding hfref-def hn-refine-def
by \((auto\ simp:\ pure-def)\)

We can also write a more direct precondition, as long as it implies the locale

**lemma** test-remdup1-refine: \((test-remdup1, my-remdup-impl-loc.test-remdup) \in [\lambda N. N>0]_a \text{nat-assn}^k \rightarrow \text{nat-assn}\)

**apply** \((\text{rule hfref-cons}\{OF test-remdup1-refine-aux - entt-refl entt-refl entt-refl\})\)

**by** unfold-locales

**export-code** test-remdup1 checking SML

We can also register the abstract constant and the refinement, to use it in further refinements

**sepref-register** my-remdup-impl-loc.test-remdup

**lemmas** \([\text{sepref-fr-rules}] = test-remdup1-refine\)

**Static Data Structures with Custom Element Relations**

In the previous section, we have presented a refinement using an array-list without dynamic resizing. However, the argument that we actually could append to this array was quite complicated.

Another possibility is to use bounded refinement relations, i.e., a refinement relation intersected with a condition for the abstract object. In our case, \(nbn\text{-assn} N\) relates natural numbers less than \(N\) to themselves.

We will repeat the above development, using the bounded relation approach:

**definition** bremdup \(l \equiv\) do {
\((s, r) \leftarrow \text{nfoldli} l \ (\lambda-. True)\)
\((\lambda x (s, r),) \) do {
\(\text{ASSERT} \ (\text{distinct} \ r \land s = \text{set} \ r)\); — Less assertions than last time
\(\text{if } x \in s \text{ then RETURN } (s, r) \text{ else RETURN } (\text{insert } x \ s, r@[x])\)
}
\(\{\{\},[]\}\): \(\text{return} \ r\}

**sepref-register** bremdup

**locale** my-bremdup-impl-loc =

**fixes** \(N::\text{nat}\)

**assumes** \(N>0\) — This assumption is not necessary, but used to illustrate the general case, where the locale may have such assumptions

**begin**

**lemma** my-bremdup-impl-loc-this: my-bremdup-impl-loc \(N\) **by** unfold-locales

**sepref-register** \(N\)

**lemma** \(N\text{-hr}[\text{sepref-import-param}]: (N,N)\in\text{nat-rel} \text{ by simp}\)

Conceptually, what we insert in our list are elements, and these are less than \(N\).
abbreviation  

lemma aux1[intro]: \[ \text{set } l \subseteq \{0..<N\}; \text{distinct } l \] \implies \text{length } l < N

apply (simp add: distinct-card[symmetric])
apply (drule psubset-card-mono[rotated])
apply auto
done

sepref-thm remdup1-fixed is remdup :: \( \lambda l. \text{set } l \subseteq \{0..<N\}_{a} \text{(list-assn elem-assn)} \)
\[ \rightarrow \text{marl-assn } N \text{ elem-assn} \]

unfolding remdup-def[abs-def]
apply (rewrite in nfoldli - - - \( \text{ias-sz.fold-custom-empty[of } N \)\))
apply (rewrite in nfoldli - - - \( \text{marl-fold-custom-empty-sz[of } N \)\))
by sepref

concrete-definition (in –) brenddup1-fixed uses my-bremdup-impl-loc.remdup1-fixed.refine-raw
is (\( \text{uncurry0} \) \text{PR-CONST} \text{test-remdup}) :: unit-assn \[ \rightarrow \text{nat-assn} \]

prepare-code-thms (in –) brenddup1-fixed-def
lemmas remdup1-fixed-refine[sepref-fr-rules] = brenddup1-fixed.refine[\text{OF my-bremdup-impl-loc-this}]

definition test-remdup \equiv \text{do } \{ l \leftarrow \text{remdup } [0..<N]; \text{RETURN } (\text{length } l) \} \}

sepref-register test-remdup

This refinement depends on the (somewhat experimental) subtyping feature
to convert from \text{nat-assn} to \text{elem-assn}, based on context information

sepref-thm test-remdup1 is
uncurry0 (PR-CONST test-remdup) :: unit-assn\[ \rightarrow \text{nat-assn} \]

unfolding test-remdup-def PR-CONST-def
by sepref

concrete-definition (in –) test-bremdup1 uses my-bremdup-impl-loc.test-bremdup1.refine-raw
is (\( \text{uncurry0} \) \text{PR-CONST}) \text{test-remdup1}

prepare-code-thms (in –) test-bremdup1-def
lemmas test-remdup1-refine[sepref-fr-rules] = test-bremdup1.refine[\text{OF my-bremdup-impl-loc-this}]

end

lemma test-bremdup1-refine-aux: (test-bremdup1, my-bremdup-impl-loc.test-bremdup1)\[ \in [\text{my-bremdup-impl-loc}]_{a} \text{nat-assn} \rightarrow \text{nat-assn} \]
using test-bremdup1.refine
unfolding hreffen-def hn-refine-def
by (auto simp: pure-def)

lemma test-bremdup1-refine: (test-bremdup1, my-bremdup-impl-loc.test-bremdup1)\[ \in [\text{hreffen}]_{a} \text{nat-assn} \rightarrow \text{nat-assn} \]
apply (rule hreffen-cons[\text{OF test-bremdup1-refine-aux - entt-refl entt-refl entt-refl}])
by unfold-locales

408
We can also register the abstract constant and the refinement, to use it in further refinements.

lemmas [sepref-fr-rules] = test-bremdup1-refine

Fixed-Value Restriction

Initialization only works with fixed values, not with dynamically computed values.

...apply sepref-dbg-keep
...apply sepref-dbg-trans-keep
...supply [[unify-trace-failure, goals-limit=1]]

— The problem manifests itself in trying to carry an abstract variable (the argument to \( \text{op-arl-empty-sz} \)) to the concrete program (the second argument of \( \text{hn-refine} \)). However, the concrete program can only depend on the concrete variables, so unification fails.

Matrix Example

We first give an example for implementing point-wise matrix operations, using some utilities from the (very prototype) matrix library.

Our matrix library uses functions \('a \text{ mtx}(\text{nat} \times \text{nat}) \Rightarrow \text{'}a\) as the abstract representation. The (currently only) implementation is by arrays, mapping points at coordinates out of range to \(0::\text{'}a\).

Pointwise unary operations are those that modify every point of a matrix independently. Moreover, a zero-value must be mapped to a zero-value. As an example, we duplicate every value on the diagonal of a matrix.

Abstractly, we apply the following function to every value. The first parameter are the coordinates.

definition \text{mtx-dup-diag-f}:: \text{nat} \times \text{nat} \Rightarrow \text{'}a::\{\text{numeral},\text{times},\text{mult-zero}\} \Rightarrow \text{'}a

where \text{mtx-dup-diag-f} \equiv \lambda(i,j) x. \text{if}\ i=j\ \text{then}\ x*\text{(2)}\ \text{else}\ x

We refine this function to a heap-function, using the identity mapping for values.
context
  fixes dummy :: 'a::{numeral,times,mult-zero}
notes [[sepref-register-adhoc PR-CONST (2::'a)]]
  — Note: The setup for numerals, like 2, is a bit subtle in that numerals are always treated as constants, but have to be registered for any type they shall be used with. By default, they are only registered for int and nat.

notes [sepref-import-param] = IdI [of PR-CONST (2::'a)]
notes [sepref-import-param] = IdI [of ( * )::'a⇒-, folded fun-rel-id-simp]
begin

sepref-definition mtx-dup-diag-f1 is uncurry (RETURN oo (mtx-dup-diag-f::-'a⇒-)) :: (prod-assn nat-assn nat-assn) k∗a id-assn k→a id-assn
  unfolding mtx-dup-diag-f-def
  by sepref

end

Then, we instantiate the corresponding locale, to get an implementation for array matrices. Note that we restrict ourselves to square matrices here:

interpretation dup-diag: amtx-pointwise-unop-impl N N mtx-dup-diag-f id-assn mtx-dup-diag-f1
  apply standard
  applyS (simp add: mtx-dup-diag-f-def) []
  applyS (rule mtx-dup-diag-f1.refine)
  done

We introduce an abbreviation for the abstract operation. Note: We do not have to register it (this is done once and for all for mtx-pointwise-unop), nor do we have to declare a refinement rule (done by amtx-pointwise-unop-impl-locale)

abbreviation mtx-dup-diag ≡ mtx-pointwise-unop mtx-dup-diag-f

The operation is usable now:

sepref-thm mtx-dup-test is λm. RETURN (mtx-dup-diag (mtx-dup-diag m)) :: (asmtx-assn N int-assn)d→a asmtx-assn N int-assn
  by sepref

Similarly, there are operations to combine to matrices, and to compare two matrices:

interpretation pw-add: amtx-pointwise-binop-impl N M (((+))::(-::monoid-add) ⇒ -) id-assn return oo (((+))
  for N M
  apply standard
  apply simp
  apply (sepref-to-hoare) apply sep-auto — Alternative to synthesize concrete operation, for simple ad-hoc refinements
  done

abbreviation mtx-add ≡ mtx-pointwise-binop ((+)
A limitation here is, that the first operand is destroyed on a coarse-grained level. Although adding a matrix to itself would be valid, our tool does not support this. (However, you may use an unary operation)

Of course, you can always copy the matrix manually:

A compare operation checks that all pairs of entries fulfill some property \( f \), and at least one entry fulfills a property \( g \).

In a final example, we store some coordinates in a set, and then use the stored coordinates to access the matrix again. This illustrates how bounded relations can be used to maintain extra information, i.e., coordinates being in range.

context

fixes \( N \cdot M \cdot \text{nat} \)
notes [sepref-register-adhoc N M]
notes [sepref-import-param] = IdI[of N] IdI[of M]

begin

We introduce an assertion for coordinates

abbreviation co-assn ≡ prod-assn (nbn-assn N) (nbn-assn M)

And one for integer matrices

abbreviation mtx-assn ≡ amtx-assn N M int-assn

definition co-set-gen ≡ do {
  nfldli [0..<N] (λi. True) (λj. nfldli [0..<M] (λ-. True) (λj s.
      if max i j − min i j ≤ 1 then RETURN (insert (i,j) s)
      else RETURN s
    )) {}
}

sepref-definition co-set-gen1 is uncurry0 co-set-gen :: unit-assnk →a hs.assn
co-assn

unfolding co-set-gen-def
apply (rewrite hs.fold-custom-empty)
apply sepref-dbg-keep
apply sepref-dbg-trans-keep
— We run into the problem that the Sepref tool uses nat-assn to refine natural
numbers, and only later tries to convert it to nbn-assn. However, at this point,
the information is already lost.

oops

We can use a feature of Sepref, to annotate the desired assertion directly
into the abstract program. For this, we use annotate-assn, which inserts
the (special) constant ASSN-ANNOT, which is just identity, but enforces
refinement with the given assertion.

sepref-definition co-set-gen1 is uncurry0 (PR-CONST co-set-gen) :: unit-assnk
→a hs.assn co-assn

unfolding co-set-gen-def PR-CONST-def
apply (rewrite hs.fold-custom-empty)
apply (rewrite in insert ⊕ annotate-assn[where A=co-assn])
— Annotate the pair as coordinate before insertion

by sepref
lemmas [sepref-fr-rules] = co-set-gen1.refine

sepref-register co-set-gen

Now we can use the entries from the set as coordinates, without any worries
about them being out of range

sepref-thm co-set-use is (λm. do {
  co ← co-set-gen;
  FOREACH co (λ(i,j) m. RETURN ( m((i,j) := 1)) ) m
} 412
end

4.1.10 Type Classes
TBD

4.1.11 Higher-Order
TBD

4.1.12 A-Posteriori Optimizations

The theorem collection \texttt{sepref-opt-simps} and \texttt{sepref-opt-simps2} contain simplifier lemmas that are applied, in two stages, to the generated Imperative/HOL program.

This is the place where some optimizations, such as deforestation, and simplifying monad-expressions using the monad laws, take place.

\texttt{thm sepref-opt-simps}
\texttt{thm sepref-opt-simps2}

4.1.13 Short-Circuit Evaluation

Consider

\texttt{sepref-thm test-sc-eval is RETURN o (\lambda l. length l > 0 \land hd l) :: (list-assn \ a \ bool-assn)k \rightarrow a bool-assn}
\texttt{apply sepref-dbg-keep}
\texttt{apply sepref-dbg-trans-keep}
\texttt{apply sepref-dbg-trans-step-keep}

— Got stuck, as the operands of \(\land\) are evaluated before applying the operator, i.e., \(hd\) is also applied to empty lists

\texttt{oops}

\texttt{sepref-thm test-sc-eval is RETURN o (\lambda l. length l > 0 \land hd l) :: (list-assn \ a \ bool-assn)k \rightarrow a bool-assn}
\texttt{unfolding short-circuit-conv} — Enables short-circuit evaluation by rewriting \(\land\), \(\lor\), and \(\rightarrow\) to \texttt{if}-expressions
\texttt{by sepref}

end
4.2 Reference Guide

theory Sepref-Guide-Reference
imports ../IICF/IICF
begin

This guide contains a short reference of the most important Sepref commands, methods, and attributes, as well as a short description of the internal working, and troubleshooting information with examples.

Note: To get an impression how to actually use the Sepref-tool, read the quickstart guide first!

4.2.1 The Sepref Method

The sepref method is the central method of the tool. Given a schematic goal of the form \( \text{hn-refine } \Gamma \ ?c \ ?\Gamma' \ ?R \ f \), it tries to synthesize terms for the schematics and prove the theorem. Note that the \( ?\Gamma' \) and \( ?R \) may also be fixed terms, in which case frame inference is used to match the generated assertions with the given ones. \( \Gamma \) must contain a description of the available refinements on the heap, the assertion for each variable must be marked with a \( \text{hn-ctxt} \) tag.

Alternatively, a term of the form \((?c.f) \in [P]_a A \to R\) is accepted, where \(A\) describes the refinement and preservation of the arguments, and \(R\) the refinement of the result. \( f \) must be in uncurried form (i.e. have exactly one argument).

We give some very basic examples here. In practice, you would almost always use the higher-level commands sepref-definition and sepref-register.

In its most primitive form, the Sepref-tool is applied like this:

schematic-goal
notes [id-rules] = itypeI[of x TYPE(nat)] itypeI[of a TYPE(bool list)]
shows hn-refine
  (hn-ctxt nat-assn x xi * hn-ctxt (array-assn bool-assn) a ai)
  (?c: ?c Heap) ?\Gamma' ?R
  (do { ASSERT (x<length a); RETURN (a!x) })
by sepref

The above command asks Sepref to synthesize a program, in a heap context where there is a natural number, refined by \(\text{nat-assn}\), and a list of booleans, refined by \(\text{array-assn bool-assn}\). The \(\text{id-rules}\) declarations declare the abstract variables to the operation identification heuristics, such that they are recognized as operands.

Using the alternative hfref-form, we can write:

schematic-goal (uncurry (?c), uncurry (\lambda x a. do \{ASSERT (x<length a); RETURN (a!x)\}))
\begin{align*}
\in \text{nat-assn}^k \ast_a (\text{array-assn bool-assn})^k \rightarrow_a \text{bool-assn} \\
\text{by sepref}
\end{align*}

This uses the specified assertions to derive the rules for operation identification automatically. For this, it uses the assertion-interface bindings declared in `intf-of-assn`. If there is no such binding, it uses the HOL type as interface type.

\textbf{thm `intf-of-assn`}

The `sepref`-method is split into various phases, which we will explain now.

\textbf{Preprocessing Phase}

This tactic converts a goal in `hhref` form to the more basic `hn-refine` form. It uses the theorems from `intf-of-assn` to add interface type declarations for the generated operands. The final result is massaged by rewriting with `to-hnr-post`, and then with `sepref-preproc`.

Moreover, this phase ensures that there is a constraint slot goal (see section on constraints).

The method `sepref-dbg-preproc` gives direct access to the preprocessing phase.

\textbf{thm `sepref-preproc`}

\textbf{thm `intf-of-assn`}

\textbf{thm `to-hnr-post` — Note: These rules are only instantiated for up to 5 arguments. If you have functions with more arguments, you need to add corresponding theorems here!}

\textbf{Consequence Rule Phase}

This phase rewrites `hn-invalid - x y` assertions in the postcondition to `hn-ctxt (\lambda\ -\ -. true) x y` assertions, which are trivial to discharge. Then, it applies `CONS-init`, to make postcondition and result relation schematic, and introduce (separation logic) implications to the originals, which are discharged after synthesis.

Use `sepref-dbg-cons-init` for direct access to this phase. The method `weaken-hnr-post` performs the rewriting of `hn-invalid` to `\lambda\ -\ -. true` postconditions, and may be useful on its own for proving combinator rules.

\textbf{Operation Identification Phase}

The purpose of this phase is to identify the conceptual operations in the given program. Consider, for example, a map `m::'k \Rightarrow 'v option`. If one writes `m(k \mapsto v)`, this is a map update. However, in Isabelle/HOL maps are encoded as functions `'(k \Rightarrow 'v option`, and the map update is just syntactic
sugar for \textit{fun-upd} \( m \ k \ (\textit{Some} \ v) \). And, likewise, map lookup is just function application.

However, the Sepref tool must be able to distinguish between maps and functions into the option type, because maps shall be refined, to e.g., hash-tables, while functions into the option type shall be not. Consider, e.g., the term \textit{Some} \( x \). Shall \textit{Some} be interpreted as the constructor of the option datatype, or as a map, mapping each element to itself, and perhaps be implemented with a hashtable.

Moreover, for technical reasons, the translation phase of Sepref expects each operation to be a single constant applied to its operands. This criterion is neither matched by map lookup (no constant, just application of the first to the second operand), nor map update (complex expression, involving several constants).

The operation identification phase uses a heuristics to find the conceptual types in a term (e.g., discriminate between map and function to option), and rewrite the operations to single constants (e.g. \textit{op-map-lookup} for map lookup). The heuristics is a type-inference algorithm combined with rewriting. Note that the inferred conceptual type does not necessarily match the HOL type, nor does it have a semantic meaning, other than guiding the heuristics.

The heuristics store a set of typing rules for constants, in \textit{id-rules}. Moreover, it stores two sets of rewrite rules, in \textit{pat-rules} and \textit{def-pat-rules}. A term is typed by first trying to apply a rewrite rule, and then applying standard Hindley-Milner type inference rules for application and abstraction. Constants (and free variables) are typed using the \textit{id-rules}. If no rule for a constant exists, one is inferred from the constant’s signature. This does not work for free variables, such that rules must be available for all free variables. Rewrite rules from \textit{pat-rules} are backtracked over, while rewrite rules from \textit{def-pat-rules} are always tried first and never backtracked over.

If typing succeeds, the result is the rewritten term.

For example, consider the type of maps. Their interface (or conceptual) type is \((\?k, \?v)\) \textit{i-map}. The \textit{id-rule} for map lookup is \textit{op-map-lookup} :: TYPE(\?a \Rightarrow (\?a, \?b) \textit{i-map} \Rightarrow \?b \textit{option}). Moreover, there is a rule to rewrite function application to map lookup (\?m $ \?k \equiv \textit{op-map-lookup} \ ?k \ ?m). It can be backtracked over, such that also functions into the option type are possible.

\begin{verbatim}
  thm op-map-lookup.iype
  thm pat-map-lookup
  thm id-rules
\end{verbatim}

The operation identification phase, and all further phases, work on a tagged version of the input term, where all function applications are replaced by the tagging constant ($), and all abstractions are replaced by \( \lambda x. (\#t x\#) \)
This is required to tame Isabelle’s higher-order unification. However, it makes tagged terms quite unreadable, and it may be helpful to unfold $\text{APP-def PROTECT2-def}$ to get back the untagged form when inspecting internal states for debugging purposes.

To prevent looping, rewrite-rules can use ($'$) on the RHS. This is a synonym for ($$), and gets rewritten to ($$) after the operation identification phase. During the operation identification phase, it prevents infinite loops of pattern rewrite rules.

Interface type annotations can be added to the term using ($$$i$) (syntax $t$$::$i$\textsc{TYPE}$($'$a'$)$).

In many cases, it is desirable to treat complex terms as a single constant, a standard example are constants defined inside locales, which may have locale parameters attached. Those terms can be wrapped into an $\text{PR-CONST}$ tag, which causes them to be treated like a single constant. Such constants must always have $\text{id-rules}$, as the interface type inference from the signature does not apply here.

**Troubleshooting Operation Identification**

If the operation identification fails, in most cases one has forgotten to register an $\text{id-rule}$ for a free variable or complex $\text{PR-CONS}$ constant, or the identification rule is malformed. Note that, in practice, identification rules are registered by the $\text{sepref-register}$ (see below), which catches many malformed rules, and handles $\text{PR-CONS}$ tagging automatically. Another frequent source of errors here is forgetting to register a constant with a conceptual type other than its signature. In this case, operation identification gets stuck trying to unify the signature’s type with the interface type, e.g., $'k$ $\Rightarrow$ $'v$ option with ($'$k', $'v'$)$ i$-map.

The method $\text{sepref-dbg-id}$ invokes the id-phase in isolation. The method $\text{sepref-dbg-id-keep}$ returns the internal state where type inference got stuck. It returns a sequence of all stuck states, which can be inspected using $\text{back}$.

The methods $\text{sepref-dbg-id-init}, \text{sepref-dbg-id-step}$, and $\text{sepref-dbg-id-solve}$ can be used to single-step the operation identification phase. Here, solve applies single steps until the current subgoal is discharged. Be aware that application of single steps allows no automatic backtracking, such that backtracking has to be done manually.

Examples for identification errors

```
context
  fixes N::nat
  notes [sepref-import-param] = IdI[of N]
begin
```

```
sepref-thm N-plus-2-example is uncurry0 (RETURN (N+2)) :: unit-assn\(^k\) \(\rightarrow_a\) nat-assn

apply sepref-dbg-keep
apply sepref-dbg-id-keep
— Forgot to register \(n\)
oops

Solution: Register \(n\), be careful not to export meaningless registrations from context!

context
notes [[sepref-register-adhoc \(N\)]]
begin
sepref-thm N-plus-2-example is uncurry0 (RETURN (N+2)) :: unit-assn\(^k\)
\(\rightarrow_a\) nat-assn by sepref
end
end

definition my-map \(\equiv\) op-map-empty

lemmas [[sepref-fr-rules] = hm.empty-hnr[folded my-map-def]]

sepref-thm my-map-example is uncurry0 (RETURN (my-map(False\(\rightarrow\)1))) :: unit-assn\(^k\)
\(\rightarrow_a\) hm.assn bool-assn nat-assn
apply sepref-dbg-keep
apply sepref-dbg-trans-keep
— Stuck at refinement for function update on map
oops

Solution: Register with correct interface type

sepref-register my-map :: (\('k',\('v')\) i-map

sepref-thm my-map-example is uncurry0 (RETURN (my-map(False\(\rightarrow\)1))) :: unit-assn\(^k\)
\(\rightarrow_a\) hm.assn bool-assn nat-assn
by sepref

Monadify Phase

The monadify phase rewrites the program such that every operation becomes visible on the monad level, that is, nested HOL-expressions are flattened. Also combinators (e.g. if, fold, case) may get flattened, if special rules are registered for that.

Moreover, the monadify phase fixes the number of operands applied to an operation, using eta-expansion to add missing operands.

Finally, the monadify phase handles duplicate parameters to an operation, by inserting a \(COPY\) tag. This is necessary as our tool expects the parameters of a function to be separate, even for read-only parameters\(^1\).

\(^1\)Using fractional permissions or some other more fine grained ownership model might lift this restriction in the future.

418
The monadify phase consists of a number of sub-phases. The method `sepref-dbg-monadify` executes the monadify phase, the method `sepref-dbg-monadify-keep` stops at a failing sub-phase and presents the internal goal state before the failing sub-phase.

**Monadify: Arity**

In the first sub-phase, the rules from `sepref-monadify-arity` are used to standardize the number of operands applied to a constant. The rules work by rewriting each constant to a lambda-expression with the desired number of arguments, and the using beta-reduction to account for already existing arguments. Also higher-order arguments can be enforced, for example, the rule for fold enforces three arguments, the function itself having two arguments: 

\[
\text{fold} \equiv \lambda x. \left((\lambda x a. \left((\lambda x b. \left(((\lambda \text{SP fold} \ (\lambda x a. \left((\lambda x b. \left((x a \#) (x b \#))\right)) (x a \#) (x b \#))\right)) (x a \#) (x b \#))\right)) (x a \#) (x b \#))\right).
\]

In order to prevent arity rules being applied infinitely often, the `SP` tag can be used on the RHS. It prevents anything inside from being changed, and gets removed after the arity step.

The method `sepref-dbg-monadify-arity` gives you direct access to this phase.

In the Sepref-tool, we use the terminology `operator/operation` for a function that only has first-order arguments, which are evaluated before the function is applied (e.g. `(+)`), and `combinator` for operations with higher-order arguments or custom evaluation orders (e.g. `fold`, `If`).

Note: In practice, most arity (and combinator) rules are declared automatically by `sepref-register` or `sepref-decl-op`. Manual declaration is only required for higher-order functions.

**Monadify: Combinators**

The second sub-phase flattens the term. It has a rule for every function into `- nres` type, that determines the evaluation order of the arguments. First-order arguments are evaluated before an operation is applied. Higher-order arguments are treated specially, as they are evaluated during executing the (combinator) operation. The rules are in `sepref-monadify-comb`.

Evaluation of plain (non-monadic) terms is triggered by wrapping them into the `EVAL` tag. The `sepref-monadify-comb` rules may also contain rewrite-rules for the `EVAL` tag, for example to unfold plain combinators into the monad (e.g. 

\[
\text{EVAL} \$ (\text{If} \ $ ?b \ $ ?t \ $ ?e) \equiv (\Rightarrow) \ $ (\text{EVAL} \ $ ?b) \ $ (\lambda x. \ (\text{EVAL} \ $ ?t) \ $ (\text{EVAL} \ $ ?e))
\]

\[
\text{EVAL} \$ (\text{case-list} \ $ ?fn \ $ (\lambda x. \ (\text{EVAL} \ $ ?fc x xa\#)) \ $ ?l) \equiv (\Rightarrow) \ $ (\text{EVAL} \ $ ?l) \ $ (\lambda x. \ (\text{EVAL} \ $ ?fc x xa\#))
\]

\[
\text{EVAL} \$ (\text{case-list} \ $ ?fn \ $ (\lambda x. \ (\text{EVAL} \ $ ?fc x xa\#))) \ $ ?l) \equiv (\Rightarrow) \ $ (\text{EVAL} \ $ ?l) \ $ (\lambda x. \ (\text{EVAL} \ $ ?fc x xa\#))
\]

419
EVAL $(\text{case-prod}$(\lambda x. (\#x.a. (\#fp x xa\#))$)$ ?p) \equiv (\Rightarrow) (EVAL \ ?p)$(\lambda x. (\#case-prod$(\lambda x. (\#EVAL \ ?fp x xa\#))$)$ x\#))$

EVAL $(\text{case-option}$(\lambda x. (\#fs x\#))$ ?ov) \equiv (\Rightarrow) (EVAL \ ?ov)$(\lambda x. (\#case-option$(EVAL \ ?fn)$(\lambda x. (\#EVAL \ ?fs x\#))$)$ x\#))$

EVAL $(\text{Let}$(\lambda x. (\#f x\#))$)$ \equiv (\Rightarrow) (EVAL \ ?v)$(\lambda x. (\#EVAL \ ?f x\#))$

If no such rule applies, the default method is to interpret the head of the term as a function, and recursively evaluate the arguments, using left-to-right evaluation order. The head of a term inside EVAL must not be an abstraction. Otherwise, the EVAL tag remains in the term, and the next sub-phase detects this and fails.

The method sepref-dbq-monadify-comb executes the combinator-phase in isolation.

**Monadify: Check-Eval**

This phase just checks for remaining EVAL tags in the term, and fails if there are such tags. The method sepref-dbq-monadify-check-EVAL gives direct access to this phase.

Remaining EVAL tags indicate higher-order functions without an appropriate setup of the combinator-rules being used. For example:

**definition** my-fold \equiv fold

**lemma** my-fold-test is \lambda l. do { RETURN (my-fold (\lambda x y. x+y*2) l 0) } :: (list-assn nat-assn)k -> nat-assn

by sepref-dbq-keep

by sepref-dbq-monadify-keep

— An EVAL-tag with an abstraction remains. This is b/c the default heuristics tries to interpret the function inside the fold as a plain value argument.

**oops**

Solution: Register appropriate arity and combinator-rules

**lemma** my-fold-arity[sepref-monadify-arity]: my-fold \equiv \lambda_2 f l s. SP my-fold$(\lambda_2 x s. f$x$s)$ by auto

The combinator-rule rewrites to the already existing and set up combinator nfoldli:

**lemma** monadify-plain-my-fold[sepref-monadify-comb]: EVAL$(my-fold$(\lambda_2 x s. f x s)$s)s) \equiv (\Rightarrow)$EVAL$(\lambda_2 l. (\Rightarrow)$EVAL$s)$(\lambda_2 s. nfoldli$s(l(\lambda_2 x s. EVAL$(f x s))s))

by (simp add: fold-eq-nfoldli my-fold-def)

**sepref-thm** my-fold-test is \lambda l. do { RETURN (my-fold (\lambda x y. x+y*2) l 0) } :: (list-assn nat-assn)k -> nat-assn

by sepref
Monadify: Dup

The last three phases, mark-params, dup, remove-pass are to detect duplicate parameters, and insert COPY tags. The first phase, mark-params, adds PASS tags around all parameters. Parameters are bound variables and terms that have a refinement in the precondition.

The second phase detects duplicate parameters and inserts COPY tags to remove them. Finally, the last phase removes the PASS tags again.

The methods sepref-dbg-monadify-mark-params, sepref-dbg-monadify-dup, and sepref-dbg-monadify-remove-pass gives you access to these phases.

Monadify: Step-Through Example

We give an annotated example of the monadify phase. Note that the program utilizes a few features of monadify:

- The fold function is higher-order, and gets flattened
- The first argument to fold is eta-contracted. The missing argument is added.
- The multiplication uses the same argument twice. A copy-tag is inserted.

sepref-thm monadify-step-thru-test is \( \lambda l. \) do {
  let i = length l;
  RETURN (fold (\( x. (+) (x\times x) \)) l i)
} :: (list-assn nat-assn) \( ^{\text{\( a \)}} \) nat-assn
apply sepref-dbg-preproc
apply sepref-dbg-cons-init
apply sepref-dbg-id
apply sepref-dbg-arity — Second operand of fold-function is added
apply sepref-dbg-monadify-comb — Flattened. fold rewritten to monadic-nfoldli.
apply sepref-dbg-monadify-check-EVAL — No EVAL tags left
apply sepref-dbg-monadify-mark-params — Parameters marked by PASS. Note the multiplication \( x\times x \).
apply sepref-dbg-monadify-dup — COPY tag inserted.
apply sepref-dbg-monadify-remove-pass — PASS tag removed again
apply sepref-dbg-opt-init
apply sepref-dbg-trans
apply sepref-dbg-opt

421
apply sepref-dbg-cons-solve
apply sepref-dbg-cons-solve
apply sepref-dbg-cons-solve
done

Optimization Init Phase

This phase, accessed by sepref-dbg-opt-init, just applies the rule \[ [\text{hn-refine } \Gamma ?c ?\Gamma' ?R ?a; \text{CNV } ?c ?c'] \Rightarrow \text{hn-refine } ?\Gamma' ?c' ?\Gamma' ?R ?a \] to set up a subgoal for a-posteriori optimization.

Translation Phase

The translation phase is the main phase of the Sepref tool. It performs the actual synthesis of the imperative program from the abstract one. For this, it integrates various components, among others, a frame inference tool, a semantic side-condition solver and a monotonicity prover.

The translation phase consists of two major sub-phases: Application of translation rules and solving of deferred constraints.

The method sepref-dbg-trans executes the translation phase, sepref-dbg-trans-keep executes the translation phase, presenting the internal goal state of a failed sub-phase.

The translation rule phase repeatedly applies translation steps, until the subgoal is completely solved.

The main idea of the translation phase is, that for every abstract variable \( x \) in scope, the precondition contains an assertion of the form \( \text{hn-ctxt } A x x_i \), indicating how this variable is implemented. Common abbreviations are \( \text{hn-val } R x x_i \equiv \text{hn-val } R x x_i \) and \( \text{hn-invalid } A x x_i \equiv \text{hn-invalid } A x x_i \).

Translation: Step

A translation step applies a single synthesis step for an operator, or solves a deferred side-condition.

There are two types of translation steps: Combinator steps and operator steps. A combinator step consists of applying a rule from sepref-comb-rules to the goal-state. If no such rule applies, the rules are tried again after rewriting the precondition with sepref-frame-normrel-eqs (see frame-inference). The premises of the combinator rule become new subgoals, which are solved by subsequent steps. No backtracking is applied over combinator rules. This restriction has been introduced to make the tool more deterministic, and hence more manageable.

An operator step applies an operator rule (from sepref-fr-rules) with frame-inference, and then tries to solve the resulting side conditions immediately.
If not all side-conditions can be solved, it backtracks over the application of the operator rule.

Note that, currently, side conditions to operator rules cannot contain synthesis goals themselves. Again, this restriction reduces the tool’s complexity by avoiding deep nesting of synthesis. However, it hinders the important feature of generic algorithms, where an operation can issue synthesis subgoals for required operations it is built from (E.g., set union can be implemented by insert and iteration). Our predecessor tool, Autoref, makes heavy use of this feature, and we consider dropping the restriction in the near future.

An operator-step itself consists of several sub-phases:

**Align goal** Splits the precondition into the arguments actually occurring in the operation, and the rest (called frame).

**Frame rule** Applies a frame rule to focus on the actual arguments. Moreover, it inserts a subgoal of the form $RECOVER-PURE \Gamma \Gamma'$, which is used to restore invalidated arguments if possible. Finally, it generates an assumption of the form $\textit{vassn-tag} \Gamma'$, which means that the precondition holds on some heap. This assumption is used to extract semantic information from the precondition during side-condition solving.

**Recover pure** This phase tries to recover invalidated arguments. An invalidated argument is one that has been destroyed by a previous operation. It occurs in the precondition as $\textit{hn-invalid} A x x_i$, which indicates that there exists a heap where the refinement holds. However, if the refinement assertion $A$ does not depend on the heap (is pure), the invalidated argument can be recovered. The purity assumption is inserted as a constraint (see constraints), such that it can be deferred.

**Apply rule** This phase applies a rule from $\textit{sepref-fr-rules}$ to the subgoal. If there is no matching rule, matching is retried after rewriting the precondition with $\textit{sepref-frame-normrel-eqs}$. If this does not succeed either, a consequence rule is used on the precondition. The implication becomes an additional side condition, which will be solved by the frame inference tool.

To avoid too much backtracking, the new precondition is massaged to have the same structure as the old one, i.e., it contains a (now schematic) refinement assertion for each operand. This excludes rules for which the frame inference would fail anyway.

If a matching rule is found, it is applied and all new subgoals are solved by the side-condition solver. If this fails, the tool backtracks over the application of the $\textit{sepref-fr-rules}$-rules. Note that direct matches prevent precondition simplification, and matches after precondition simplification prevent the consequence rule to be applied.
The method \texttt{sepref-dbgt-trans-step} performs a single translation step. The method \texttt{sepref-dbgt-trans-step-keep} presents the internal goal state on failure. If it fails in the \texttt{apply-rule} phase, it presents the sequence of states with partially unsolved side conditions for all matching rules.

**Translation: Side Conditions**

The side condition solver is used to discharge goals that arise as side-conditions to the translation rules. It does a syntactic discrimination of the side condition type, and then invokes the appropriate solver. Currently, it supports the following side conditions:

- **Merge** ($\forall A \implies t$). These are used to merge postconditions from different branches of the program (e.g. after an if-then-else). They are solved by the frame inference tool (see section on frame inference).

- **Frame** ($t$). Used to match up the current precondition against the precondition of the applied rule. Solved by the frame inference tool (see section on frame inference).

- **Independence** ($\text{INDEP} (\text{?R} x_1 \ldots x_n)$). Deprecated. Used to instantiate a schematic variable such that it does not depend on any bound variables any more. Originally used to make goals more readable, we are considering of dropping this.

- **Constraints** ($\text{CONSTRAINT} - -$) Apply solver for deferrable constraints (see section on constraints).

- **Monotonicity** ($\text{mono-Heap} -$) Apply monotonicity solver. Monotonicity subgoals occur when translating recursion combinators. Monadic expressions are monotonic by construction, and this side-condition solver just forwards to the monotonicity prover of the partial function package, after stripping any preconditions from the subgoal, which are not supported by the case split mechanism of the monotonicity prover (as of Isabelle2016).

- **Prefer/Defer** ($\text{PREFER-tag} / \text{DEFER-tag}$). Deprecated. Invoke the tagged solver of the Autoref tool. Used historically for importing refinements from the Autoref tool, but as Sepref becomes more complete imports from Autoref are not required any more.

- **Resolve with Premise** $\text{RPREM} -$ Resolve subgoal with one of its premises. Used for translation of recursion combinators.
Generic Algorithm  **GEN-ALGO** - - Triggers resolution with a rule from
`sepref-gen-algo-rules`. This is a poor-man’s version of generic algo-
rithm, which is currently only used to synthesize to-list conversions
for foreach-loops.

**Fallback**  (Any pattern not matching the above, nor being a `hn-refine` goal).
Unfolds the application and abstraction tagging, as well as `bind-ref-tag`
tags which are inserted by several translation rules to indicate the value
a variable has been bound to, and then tries to solve the goal by `auto`,
after freezing schematic variables. This tactic is used to discharge
semantic side conditions, e.g., in-range conditions for array indexing.

Methods: `sepref-dbg-side` to apply a side-condition solving step, `sepref-dbg-side-unfold`
to apply the unfolding of application and binding tags and `sepref-dbg-side-keep`
to return the internal state after failed side-condition solving.

**Translation: Constraints**

During the translation phase, the refinement of operands is not always known
immediately, such that schematic variables may occur as refinement asser-
tions. Side conditions on those refinement assertions cannot be discharged
until the schematic variable gets instantiated.

Thus, side conditions may be tagged with `CONSTRAINT`. If the side con-
dition solver encounters a constraint side condition, it first removes the con-
straint tag (`?P ?x =⇒ CONSTRAINT ?P ?x`) and freezes all schematic
variables to prevent them from accidentally getting instantiated. Then it
simplifies with `constraint-simps` and tries to solve the goal using rules from
`safe-constraint-rules` (no backtracking) and `constraint-rules` (with backtrack-
ning).

If solving the constraint is not successful, only the safe rules are applied,
and the remaining subgoals are moved to a special `CONSTRAINT-SLOT`
subgoal, that always is the last subgoal, and is initialized by the prepro-
cessing phase of Sepref. Moving the subgoal to the constraint slot looks for
Isabelle’s tacticals like the subgoal has been solved. In reality, it is only
deferred and must be solved later.

Constraints are used in several phases of Sepref, and all constraints are solved
at the end of the translation phase, and at the end of the Sepref invocation.

Methods:

- `solve-constraint` to apply constraint solving, the `CONSTRAINT-tag`
is optional.

- `safe-constraint` to apply safe rules, the `CONSTRAINT-tag` is optional.
print-slot to print the contents of the constraint slot.

Translation: Merging and Frame Inference

Frame inference solves goals of the form $\Gamma \Rightarrow t$. For this, it matches $hn$-ctxt components in $\Gamma'$ with those in $\Gamma$. Matching is done according to the refined variables. The matching pairs and the rest is then treated differently: The rest is resolved by repeatedly applying the rules from $\forall P \Rightarrow t$

\[
\begin{align*}
?F & \Rightarrow_t ?F' \Rightarrow ?F \ast hn$-ctxt $?A ?x ?y \Rightarrow_t ?F'
\end{align*}
\]

$\forall P \Rightarrow_t \text{emp}$. The matching pairs are resolved by repeatedly applying rules from $\forall P \Rightarrow_t ?P$

\[
\begin{align*}
[?P & \Rightarrow_t ?P'; ? F \Rightarrow_t ? F'] \Rightarrow ?F \ast ?P \Rightarrow_t ?F' \ast ?P' \\
? F & \Rightarrow_t \text{hn-invalid} ?R ?x ?y
\end{align*}
\]

and $\forall P \Rightarrow_t ?P \Rightarrow_t \text{hn-ctxt} (\lambda \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \·

This is essentially a subtyping mechanism on the level of refinement assertions, which is quite useful for maintaining natural side conditions on operands. A standard example is to maintain a list of array indices: The refinement assertion for array indices is nat-assn restricted to indices that are in range: nbn-assn N. When inserting natural numbers into this list, one has to prove that they are actually in range (conversion from nat-assn to nbn-assn). Elements of the list can be used as natural numbers (conversion from nbn-assn to nat-assn). Additionally, the side condition solver can derive that the predicate holds on the abstract variable (via the vassn-tag inserted by the operator steps).

Translation: Annotated Example

context
fixes N::nat
notes [[sepref-register-adhoc N]]
notes [sepref-import-param] = IdI[of N]

begin

This worked example utilizes the following features of the translation phase:

- We have a fold combinator, which gets translated by its combinator rule
- We add a type annotation which enforces converting the natural numbers inserted into the list being refined by nbn-assn N, i.e., smaller than N.
- We can only prove the numbers inserted into the list to be smaller than N because the combinator rule for If inserts congruence assumptions.
- By moving the elements from the list to the set, they get invalidated. However, as nat-assn is pure, they can be recovered later, allowing us to mark the list argument as read-only.

sepref-thm filter-N-test is λl. RETURN (fold (λx s. if x<N then insert (ASSN-ANNOT (nbn-assn N) x) s else s) l op-hs-empty) :: (list-assn nat-assn) →ₜ hs.assn (nbn-assn N)

apply sepref-dbgs-preproc
apply sepref-dbgs-cons-init
apply sepref-dbgs-id
apply sepref-dbgs-monadify
apply sepref-dbg-opt-init

apply sepref-dbg-trans-step — Combinator rule for bind, generating two hn-refine goals, and a frame rule to separate the bound variable from the rest.
apply sepref-dbg-trans-step — Rule for empty hashset, solves goal
apply sepref-dbg-trans-step — Combinator rule for nfoldli ($\text{INDEP} ?Rk; \text{INDEP} ?R\sigma; \text{TERM} \text{monadic-nfoldli} \Rightarrow ?\Gamma c \sigma \Rightarrow ?\Gamma * \text{hn-ctxt} ?Rk ?s ?s' ?s\sigma ?\sigma'; \wedge x' x' x' \in \text{set} ?s' \Rightarrow \text{hn-refine} (?\Gamma * \text{hn-ctxt} ?Rk x' x * \text{hn-ctxt} ?R\sigma ?\sigma' ?\sigma) (?f x) (?\Gamma f x' x' \sigma' ?\sigma) ?R\sigma (?f' x' ?\sigma); \wedge x' x' x' x' \text{hn-ctxt} ?R k' x' x \vee A \text{hn-ctxt} ?Rk' x' x \Rightarrow ?\Gamma * \text{hn-ctxt} ?Rk' x' x = ?\Gamma * \text{hn-ctxt} ?Rk' x' x \Rightarrow ?\Gamma * \text{hn-ctxt} ?Rk' x' x \Rightarrow ?\Gamma * \text{hn-ctxt} ?Rk' x' x \Rightarrow ?\Gamma * \text{hn-ctxt} ?Rk' x' x \Rightarrow ?\Gamma * \text{hn-ctxt} ?Rk' x' x \Rightarrow \text{hn-refine} ?P (\text{imp-nfoldli} ?s ?c ?f ?\sigma) (?\Gamma * \text{hn-ctxt} (\text{list-assn} ?Rk' ?s') ?s' ?s * \text{hn-invalid} ?R\sigma ?\sigma' ?\sigma) ?R\sigma (\text{monadic-nfoldli} $ ?s' (\lambda x. (#?c x)?#)) (\lambda x. (#?x a?#)) $ (#)?a))
apply sepref-dbg-trans-step — INDEP
apply sepref-dbg-trans-step — INDEP
apply sepref-dbg-trans-step — Frame to get list and initial state
apply sepref-dbg-trans-step — Refinement of continuation condition
apply sepref-dbg-trans-step — Frame to recover state after continuation condition

— Loop body
apply sepref-dbg-trans-step
apply sepref-dbg-trans-step
apply sepref-dbg-trans-step
apply sepref-dbg-trans-step
apply sepref-dbg-trans-step
apply sepref-dbg-trans-step
apply sepref-dbg-trans-step
apply sepref-dbg-trans-step
apply sepref-dbg-trans-step
apply sepref-dbg-trans-step
apply sepref-dbg-trans-step
apply sepref-dbg-trans-step
apply sepref-dbg-trans-step
apply sepref-dbg-trans-step — At this point, we arrived at the nbn-rel annotation. There is enough information to show $x' a < N$
apply sepref-dbg-trans-step
apply sepref-dbg-trans-step
apply sepref-dbg-trans-step
apply sepref-dbg-trans-step
apply sepref-dbg-trans-step
apply sepref-dbg-trans-step
apply sepref-dbg-trans-step
apply sepref-dbg-trans-step — At this point, we have to merge the postconditions from the two if branches. nat-rel gets merged with invalid-assn (nbn-assn n), yielding invalid-assn nat-assn
apply sepref-dbg-trans-step
apply sepref-dbg-trans-step — Frame rule separating bound variable from rest
apply sepref-dbg-trans-step — Frame rule separating fold-state from rest
apply sepref-dbg-trans-step — Merging elements of list before body with elements of list after body, to get refinement for resulting list
apply sepref-dbg-trans-step — Frame rule from initial bind, separating bound variable from the rest
apply sepref-dbg-opt
apply sepref-dbg-cons-solve — Frame rule, recovering the invalidated list or pure elements, propagating recovery over the list structure
apply sepref-dbg-cons-solve — Trivial frame rule
apply sepref-dbg-constraints
done

end

Optimization Phase

The optimization phase simplifies the generated program, first with sepref-opt-simps, and then with sepref-opt-simps2. For simplification, the tag CNV is used, which is discharged with CNV ?x ?x after simplification.

Method sepref-dbg-opt gives direct access to this phase. The simplification is used to beautify the generated code. The most important simplifications collapse code that does not depend on the heap to plain expressions (using the monad laws), and apply certain deforestation optimizations.

Consider the following example:

sepref-thm opt-example is \( \lambda n. \) do { let \( r = \text{fold} (+) [1..<n] 0 \); RETURN (\( n*n+2 \)) }

:: nat-assn^k a nat-assn
apply sepref-dbg-preproc
apply sepref-dbg-cons-init
apply sepref-dbg-id
apply sepref-dbg-monadify
apply sepref-dbg-opt-init
apply sepref-dbg-trans
— The generated program contains many superfluous binds, moreover, it actually generates a list and then folds over it
supply [[show-main-goal]]
apply sepref-dbg-opt
— The superfluous binds have been collapsed, and the fold over the list has been replaced by \textit{imp-for}', which uses a counter.
apply sepref-dbg-cons-solve
apply sepref-dbg-cons-solve
apply sepref-dbg-constraints
done

Cons-Solve Phases

These two phases, accessible via sepref-dbg-cons-solve, applies the frame inference tool to solve the two implications generated by the consequence rule phase.
Constraints Phase

This phase, accessible via `sepref-dbg-constraints`, solve the deferred constraints that are left, and then removes the `CONSTRAINT-SLOT` subgoal.

4.2.2 Refinement Rules

There are two forms of specifying refinement between an Imperative/HOL program and an abstract program in the `nres`-monad. The `hn-refine` form (also `hnr-form`) is the more low-level form. The term $P \implies hn-refine \Gamma \ c \ \Gamma' \ R \ a$ states that, under precondition $P$, for a heap described by $\Gamma$, the Imperative/HOL program $c$ produces a heap described by $\Gamma'$ and the result is refined by $R$. Moreover, the abstract result is among the possible results of the abstract program $a$.

This low-level form formally enforces no restrictions on its arguments, however, there are some assumed by our tool:

- $\Gamma$ must have the form $hn-ctxt \ A_1 \ x_1 \ * \ ... \ * \ hn-ctxt \ A_n \ x_n \ x_i$.
- $\Gamma'$ must have the form $hn-ctxt \ B_1 \ x_1 \ * \ ... \ * \ hn-ctxt \ B_n \ x_n \ x_i$, where either $B_i = A_i$ or $B_i = invalid-assn \ A_i$. This means that each argument to the program is either preserved or destroyed.
- $R$ must not contain a $hn-ctxt$ tag.
- $a$ must be in protected form (($) and `PROTECT2` tags).

The high-level `hhref` form formally enforces these restrictions. Moreover, it assumes $c$ and $a$ to be presented as functions from exactly one argument. For constants or functions with more arguments, you may use `uncurry0` and `uncurry`. (Also available `uncurry2` to `\lambda. uncurry2 (uncurry2 (uncurry f)))`.

The general form is $PC \implies (uncurry_x \ f, uncurry_x \ g) \in [P]_a A_1^{k_1} *_a \ ... \ *_a A_n^{k_n} \rightarrow R$, where $ki$ is $k$ if the argument is preserved (kept) or $d$ is it is destroyed. $PC$ are preconditions of the rule that do not depend on the arguments, usually restrictions on the relations. $P$ is a predicate on the single argument of $g$, representing the precondition that depends on the arguments.

Optionally, $g$ may be of the form `RETURN o...o g'`, in which case the rule applies to a plain function.

If there is no precondition, there is a shorter syntax: $Args \rightarrow_a R \equiv Args \rightarrow_a R$.

For example, consider `arl-swap-hnr [unfolded pre-list-swap-def]`. It reads `CONSTRAINT is-pure A \implies (uncurry2 arl-swap, uncurry2 (RETURN ooo op-list-swap)) \in [\lambda (l, i, j). i < length l \land j < length l]_a (arl-assn A)^d *_a nat-assn_k *_a nat-assn_k \rightarrow arl-assn A`
We have three arguments, the list and two indexes. The refinement assertion $A$ for the list elements must be pure, and the indexes must be in range. The original list is destroyed, the indexes are kept.

```thm```
```
```

Converting between hfref and hnr form

A subgoal in hfref form is converted to hnr form by the preprocessing phase of Sepref (see there for a description).

Theorems with hnr/hfref conclusions can be converted using `to-hfref/to-hnr`. This conversion is automatically done for rules registered with `sepref-fr-rules`, such that this attribute accepts both forms.

Conversion to hnr-form can be controlled by specifying `to-hnr-post` unfold-rules, which are applied after the conversion.

Note: These currently contain hard-coded rules to handle `RETURN o...o -` for up to six arguments. If you have more arguments, you need to add corresponding rules here, until this issue is fixed and the tool can produce such rules automatically.

Similarly, `to-hfref-post` is applied after conversion to hfref form.

```thm```
```
```

Importing Parametricity Theorems

For pure refinements, it is sometimes simpler to specify a parametricity theorem than a hnr/hfref theorem, in particular as there is a large number of parametricity theorems readily available, in the parametricity component or Autoref, and in the Lifting/Transfer tool.

Autoref uses a set-based notation for parametricity theorems (e.g. $((@), (@)) \in \langle A \rangle list-rel \rightarrow \langle A \rangle list-rel \rightarrow \langle A \rangle list-rel$), while lifting/transfer uses a predicate based notation (e.g. `rel-fun (list-all2 A) (rel-fun (list-all2 A) (list-all2 A)) (@) (@))`.

Currently, we only support the Autoref style, but provide a few lemmas that ease manual conversion from the Lifting/Transfer style.

Given a parametricity theorem, the attribute `sepref-param` converts it to a hfref theorem, the attribute `sepref-import-param` does the conversion and registers the result as operator rule. Relation variables are converted to assertion variables with an `is-pure` constraint.

The behaviour can be customized by `sepref-import-rewrite`, which contains rewrite rules applied in the last but one step of the conversion, before converting relation variables to assertion variables. These theorems can be used to convert relations to there corresponding assertions, e.g., `pure`
\((\forall R)\text{list-rel}\) = \text{list-assn} (\text{pure} \ ?R)\) converts a list relation to a list assertion.

For debugging purposes, the attribute \text{sepref-dbg-import-rl-only} converts a parametricity theorem to a \text{hnr-theorem}. This is the first step of the standard conversion, followed by a conversion to \text{hfref} form.

\text{thm sepref-import-rewrite}
\text{thm param-append} — Parametricity theorem for append
\text{thm param-append[sepref-param]} — Converted to \text{hfref-form}. \text{list-rel} is rewritten to \text{list-assn}, and the relation variable is replaced by an assertion variable and a \text{is-pure} constraint.

\text{thm param-append[sepref-dbg-import-rl-only]}

For re-using Lifting/Transfer style theorems, the constants \text{p2rel} and \text{rel2p} may be helpful, however, there is no automation available yet.

Usage examples can be found in, e.g., \text{Refine-Imperative-HOL.IICF-Multiset}, where we import parametricity lemmas for multisets from the Lifting/Transfer package.

\text{thm p2rel} — Simp rules to convert predicate to relational style
\text{thm rel2p} — Simp rules to convert relational to predicate style

4.2.3 Composition

Fref-Rules

In standard parametricity theorems as described above, one cannot specify preconditions for the parameters, e.g., \text{hd} is only parametric for non-empty lists.

As of Isabelle2016, the Lifting/Transfer package cannot specify such preconditions at all.

\text{Autoref’s} parametricity tool can specify such preconditions by using first-order rules, (cf. \([\forall l \neq []; (\forall l', l) \in (\forall A)\text{list-rel}] \implies (\text{hd} \ l', \text{hd} \ l) \in \forall A\)). However, currently, \text{sepref-import-param} cannot handle these first-order rules.

Instead, \text{Sepref} supports the fref-format for parametricity rules, which resembles the \text{hfref-format}: Abstract and concrete objects are functions with exactly one parameter, uncurried if necessary. Moreover, there is an explicit precondition. The syntax is \(\text{(uncurry}_x f, \text{uncurry}_x g) \in [P]_f (\ldots (R_1 \times_r R_2) \times_r \ldots) \times_r R_n \implies R\). and without precondition, we have \(\ldots (R_1 \times_r R_2) \times_r \ldots) \times_r R_n \implies R\).

Note the left-branching of the tuples, which is non-standard in Isabelle. As we currently have no syntax for a left-associative product relation, we use the right-associative syntax \(\times_r\) and explicit brackets.

The attribute \text{to-fref} can convert (higher-order form) parametricity theorems to the fref-form.
Composition of href and fref theorems

fref and hhref theorems can be composed, if the abstract function or the first theorem equals the concrete function of the second theorem. Currently, we can compose an hhref with an fref theorem, yielding a hhref theorem, and two fref-theorems, yielding an fref theorem. As we do not support refinement of heap-programs, but only refinement into heap programs, we cannot compose two hhref theorems.

The attribute FCOMP does these compositions and normalizes the result. Normalization consists of precondition simplification, and distributing composition over products, such that composition can be done argument-wise. For this, we unfold with fcomp-norm-unfold, and then simplify with fcomp-norm-simps.

The FCOMP attribute tries to convert its arguments to hhref/fref form, such that it also accepts hnr-rules and parametricity rules.

The standard use-case for FCOMP is to compose multiple refinement steps to get the final correctness theorem. Examples for this are in the quickstart guide.

Another use-case for FCOMP is to compose a refinement theorem of a container operation, that refines the elements by identity, with a parametricity theorem for the container operation, that adds a (pure) refinement of the elements. In practice, the high-level utilities sepref-decl-op and sepref-decl-impl are used for this purpose. Internally, they use FCOMP.

4.2.4 Registration of Interface Types

An interface type represents some conceptual type, which is encoded to a more complex type in HOL. For example, the interface type \((\mathcal{K}, \mathcal{V})\) i-map represents maps, which are encoded as \(\mathcal{K} \Rightarrow \mathcal{V} \text{ option}\) in HOL. New interface types must be registered by the command sepref-decl-intf.
sepref-decl-intf \(\langle a, b \rangle\) \(i\)-my-intf is \(\text{'}a\ast\text{'}a \Rightarrow \text{'}b\) option

— Declares \(\langle a, b \rangle\) \(i\)-my-intf as new interface type, and registers it to correspond to \(\text{'}a \times \text{'}a \Rightarrow \text{'}b\) option. Note: For HOL, the interface type is just an arbitrary new type, which is not related to he corresponding HOL type.

sepref-decl-intf \(\langle a, b \rangle\) \(i\)-my-intf2 \((\text{infix} \ast\rightarrow\text{'}0)\) is \(\text{'}a\ast\text{'}a \Rightarrow \text{'}b\) option

— There is also a version that declares infix-syntax for the interface type. In this case we have \(\text{'a} \ast\rightarrow \text{'b}\). \(\text{'a} \Rightarrow \text{'b}\) Be aware of syntax space pollution, as the syntax for interface types and HOL types is the same.

4.2.5 Registration of Abstract Operations

Registering a new abstract operation requires some amount of setup, which is automated by the \texttt{sepref-register} tool. Currently, it only works for operations, not for combinators.

The \texttt{sepref-register} command takes a list of terms and registers them as operators. Optionally, each term can have an interface type annotation.

If there is no interface type annotation, the interface type is derived from the terms HOL type, which is rewritten by the theorems from \texttt{map-type-eqs}.

This rewriting is useful for bulk-setup of many constants with conceptual types different from there HOL-types. Note that the interface type must correspond to the HOL type of the registered term, otherwise, you’ll get an error message.

If the term is not a single constant or variable, and does not already start with a \texttt{PR-CONST} tag, such a tag will be added, and also a pattern rule will be registered to add the tag on operator identification.

If the term has a monadic result type (\(\text{- nres}\)), also an arity and combinator rule for the monadify phase are generated.

There is also an attribute version \texttt{sepref-register-adhoc}. It has the same syntax, and generates the same theorems, but does not give names to the theorems. It’s main application is to conveniently register fixed variables of a context. Warning: Make sure not to export such an attribute from the context, as it may become meaningless outside the context, or worse, confuse the tool.

Example for bulk-registration, utilizing type-rewriting

\texttt{definition map-op1} \(m\ n \equiv m(n\rightarrow n+1)\)
\texttt{definition map-op2} \(m\ n \equiv m(n\rightarrow n+2)\)
\texttt{definition map-op3} \(m\ n \equiv m(n\rightarrow n+3)\)
\texttt{definition map-op-to-map} (\(m::\text{'}a\rightarrow\text{'}b\)) \(\equiv m\)

\texttt{context}
\texttt{notes} \([\text{map-type-eqs}] = \text{map-type-eqI}[\text{of TYPE(\text{'a} \rightarrow \text{'b}) TYPE((\text{'}a, \text{'b})i-map)]}\)
\texttt{begin}
\texttt{sepref-register map-op1 map-op2 map-op3}
Registered interface types use \( i\text{-}\text{map} \)

\textbf{sepref-register} \( \text{map-op-to-map} :: (\'a \rightarrow \'b) \Rightarrow (\'a,\'b) \text{\( i\text{-}\text{map} \))}

Explicit type annotation is not rewritten

Example for insertion of \( PR\text{-}\text{CONST} \) tag and attribute-version

\begin{verbatim}
context
    fixes \( N :: \text{nat} \) and \( D :: \text{int} \)
end
\end{verbatim}

Notes:

\begin{verbatim}
[sepref-import-param] = \text{IdI[of \( N \)] \text{IdI[of \( D \)]}}
\end{verbatim}

For declaring refinement rules, the \( \text{sepref-import-param} \) attribute comes in handy here. If this is not possible, you have to work with nested contexts, proving the refinement lemmas in the first level, and declaring them as \( \text{sepref-fr-rules} \) on the second level.

\begin{verbatim}
begin
    definition \( \text{newlist} \equiv \text{replicate} \( N \) \( D \))

    \text{sepref-register} \( \text{newlist} \)
    \text{print-theorems}
    — \( PR\text{-}\text{CONST} \) tag is added, pattern rule is generated

    \text{sepref-register} \( \text{other-basename-newlist}: \text{newlist} \)
    \text{print-theorems}
    — The base name for the generated theorems can be overridden

    \text{sepref-register} \( \text{yet-another-basename-newlist}: \text{\( PR\text{-}\text{CONST} \) newlist} \)
    \text{print-theorems}
    — If \( PR\text{-}\text{CONST} \) tag is specified, no pattern rule is generated automatically

end
\end{verbatim}

Example for \( \text{mcomb/arity} \) theorems

\begin{verbatim}
definition \( \text{select-a-one} \equiv \text{SPEC} \( \lambda i. \text{length} \( l \) \land \text{\( l\![i] \equiv \{1::\text{nat}\} \))}

\text{sepref-register} \( \text{select-a-one} \)
\text{print-theorems}
    — Arity and \( \text{mcomb} \) theorem is generated
\end{verbatim}

The following command fails, as the specified interface type does not correspond to the HOL type of the term: \textbf{sepref-register} \( \text{hd} :: (\text{nat,nat}) \text{\( i\text{-}\text{map} \))}
4.2.6 High-Level tools for Interface/Implementation Declaration

The Imperative Isabelle Collections Framework (IICF), which comes with Sepref, has a concept of interfaces, which specify a set of abstract operations for a conceptual type, and implementations, which implement these operations.

Each operation may have a natural precondition, which is established already for the abstract operation. Many operations come in a plain version, and a monadic version which asserts the precondition. Implementations may strengthen the precondition with implementation specific preconditions.

Moreover, each operation comes with a parametricity lemma. When registering an implementation, the refinement of the implementation is combined with the parametricity lemma to allow for (pure) refinements of the element types.

The command \texttt{sepref-decl-op} declares an abstract operation. It takes a term defining the operation, and a parametricity relation. It generates the monadic version from the plain version, defines constants for the operations, registers them, and tries to prove parametricity lemmas automatically. Parametricity must be proved for the operation, and for the precondition. If the automatic parametricity proofs fail, the user gets presented goals that can be proven manually.

Optionally, a basename for the operation can be specified. If none is specified, a heuristics tries to derive one from the specified term.

A list of properties (separated by space and/or \texttt{and}) can be specified, which get constraint-preconditions of the relation.

Finally, the following flags can be specified. Each flag can be prefixed by
no- to invert its meaning:

**mop** (default: true) Generate monadic version of operation

**ismop** (default: false) Indicate that given term is the monadic version

**rawgoals** (default: false) Present raw goals to user, without attempting to prove them

**def** (default: true) Define a constant for the specified term. Otherwise, use the specified term literally.

The **sepref-decl-impl** command declares an implementation of an interface operation. It takes a refinement theorem for the implementation, and combines it with the corresponding parametricity theorem. After *uses*, one can override the parametricity theorem to be used. A heuristics is used to merge the preconditions of the refinement and parametricity theorem. This heuristics can be overridden by specifying the desired precondition inside [...] . Finally, the user gets presented remaining subgoals that cannot be solved by the heuristics. The command accepts the following flags:

**mop** (default: true) Generate implementation for monadic version

**ismop** (default: false) Declare that the given theorems refer to the monadic version

**transfer** (default: true) Try to automatically transfer the implementation’s precondition over the argument relation from the parametricity theorem.
4.2.7 Defining synthesized Constants

The **sepref-definition** allows one to specify a name, an abstract term and a desired refinement relation in hhref-form. It then sets up a goal that can be massaged (usually, constants are unfolded and annotations/implementation specific operations are added) and then solved by **sepref**. After the goal is solved, the command extracts the synthesized term and defines it as a constant with the specified name. Moreover, it sets up code equations for the constant, correctly handling recursion combinators. Extraction of code equations is controlled by the **prep-code** flag. Examples for this command can be found in the quickstart guide.

4.3 General Purpose Utilities

theory Sepref-Guide-General-Util
imports ../IICF/IICF
begin

This userguide documents some of the general purpose utilities that come with the Sepref tool, but are useful in other contexts, too.

4.3.1 Methods

Resolve with Premises

The **rprems** resolves the current subgoal with one of its premises. It returns a sequence of possible resolvents. Optionally, the number of the premise to resolve with can be specified.

First-Order Resolution

The **fo-rule** applies a rule with first-order matching. It is very useful to be used with theorems like \(?x = ?y \implies \?f \ ?x = \?f \ ?y\).

**notepad begin**

```
have card \{x. \exists<x \land x<(7::nat)\} = card \{x. \exists\leq x \land x\leq(6::nat)\}
  apply (fo-rule arg-cong)
  apply auto
  done
```

**end**
— While the first goal could also have been solved with rule arg-cong[where
\( f = \text{card} \)], things would be much more verbose for the following goal. (Such
goals actually occur in practice!)

\[
\begin{align*}
\text{fix } f &: \text{nat set } \Rightarrow \text{nat set } \Rightarrow \text{bool} \\
\text{have } \forall a. f \{ x. x*2 + a + 3 < 10 \} \{ x. 3 < x \land x < (7::\text{nat}) \} = f \{ x. x*2 + a \\
\leq 6 \} \{ x. 4 \leq x \land x \leq (6::\text{nat}) \} \\
\text{apply } (\text{fo-rule arg-cong fun-cong cong})+ \\
\text{apply } \text{auto} \\
\text{done}
\end{align*}
\]

\textbf{Clarsimp all goals}

\textit{clarsimp-all} is a \textit{clarsimp} on all goals. It takes the same arguments as
\textit{clarsimp}.

\textbf{VCG solver}

\textit{vc-solve} clarsimps all subgoals. Then, it tries to apply a rule specified in the
\textit{solve:} argument, and tries to solve the result by \textit{auto}. If the goal cannot be
solved this way, it is not changed.

This method is handy to be applied after verification condition generation.
If \textit{auto} shall be tried on all subgoals, specify \textit{solve: asm-rl}.

\subsection*{4.3.2 Structured Apply Scripts (experimental)}

A few variants of the apply command, that document the subgoal structure
of a proof. They are a lightweight alternative to \textit{subgoal}, and fully support
schematic variables.

\textbf{applyS} applies a method to the current subgoal, and fails if the subgoal is
not solved.

\textbf{apply1} applies a method to the current subgoal, and fails if the goal is
solved or additional goals are created.

\textbf{focus} selects the current subgoal, and optionally applies a method.

\textbf{applyF} selects the current subgoal and applies a method.

\textbf{solved} enforces no subgoals to be left in the current selection, and unselects.

Note: The selection/unselection mechanism is a primitive version of focusing
on a subgoal, realized by inserting protect-tags into the goal-state.
4.3.3 Extracting Definitions from Theorems

The **concrete-definition** can be used to extract parts of a theorem as a constant. It is documented at the place where it is defined (ctrl-click to jump there).
Chapter 5

Examples

This chapter contains practical examples of using the IRF and IICF. Moreover it contains some snippets that illustrate how to solve common tasks like setting up custom datatypes or higher-order combinators.

5.1 Imperative Graph Representation

theory Sepref-Graph
imports
  ../Sepref
  ../Sepref-ICF-Bindings
  ../IICF/IICF
begin

Graph Interface

sepref-decl-intf 'a i-graph is ('a×'a) set

definition op-graph-succ :: ('v×'v) set ⇒ 'v ⇒ 'v set
  where [simp]: op-graph-succ E u ≡ E''{u}

sepref-register op-graph-succ :: 'a i-graph ⇒ 'a ⇒ 'a set

thm intf-of-assnI

lemma [pat-rules]: ((""))$E$ (insert$u${}) ≡ op-graph-succ$E$u by simp

definition [to-relAPP]: graph-rel A ≡ ⟨A×, A⟩ set-rel

Adjacency List Implementation

lemma param-op-graph-succ[param];
  [IS-LEFT-UNIQUE A; IS-RIGHT-UNIQUE A] ⇒ (op-graph-succ, op-graph-succ)
  ∈ ⟨A⟩ graph-rel → A → ⟨A⟩ set-rel
  unfolding op-graph-succ-def[abs-def] graph-rel-def
  by parametricity

441
context begin

private definition graph-α1 l ≡ { (i,j). i<length l ∧ j∈l!i } 

private definition graph-rel1 ≡ br graph-α1 (λ-. True)

private definition succ1 l i ≡ if i<length l then l!i else {}

private lemma succ1-refine : (succ1, op-graph-succ) ∈ graph-rel1 → Id → ⟨Id⟩ set-rel
  by (auto simp: graph-rel1-def graph-α1-def br-def succ1-def split: if-split-asm intro!: ext)

private definition assn2 ≡ array-assn (pure ⟨Id⟩ list-set-rel)

definition adjg-assn A ≡ hr-comp (hr-comp assn2 graph-rel1) ((the-pure A) graph-rel)

context

notes [fcomp-norm-unfold] = adjg-assn-def [symmetric]

begin

sepref-definition succ2 is (uncurry (RETURN oo succ1)) :: (assn2 k∗a id-assn k → a pure ⟨Id⟩ list-set-rel)

unfolding succ1-def [abs-def] assn2-def by sepref

  ⇒ (uncurry succ2, uncurry (RETURN oo op-graph-succ)) ∈ (adjg-assn A) k∗a A k → a pure ⟨(Id) list-set-rel⟩
  using succ2.refine [FCOMP succ1-refine, FCOMP param-op-graph-succ, simplified, of A]
  by (simp add: IS-PURE-def list-set-rel-compp)

end

end

lemma [intf-of-assn]:
  intf-of-assn A (i::'I itself) ⇒ intf-of-assn (adjg-assn A) TYPE('I i-graph) by simp

definition cr-graph
  :: nat ⇒ (nat × nat) list ⇒ nat list Heap.array Heap

where
cr-graph numV Es ≡ do {
  a ← Array.new numV [];
  a ← imp-nfoldli Es (λ-. return True) (λ(u,v) a. do {
    l ← Array.nth a u;
    let l = v#l;
  }) a
}
export-code cr-graph checking SML-imp

5.2 Simple DFS Algorithm

theory Sepref-DFS
imports
  ../Sepref
  Sepref-Graph
begin

We define a simple DFS-algorithm, prove a simple correctness property, and
do data refinement to an efficient implementation.

5.2.1 Definition

Recursive DFS-Algorithm. $E$ is the edge relation of the graph, $vd$ the node
to search for, and $v0$ the start node. Already explored nodes are stored in
$V$.

context
  fixes $E :: \forall v rel$ and $v0 :: v$ and $tgt :: v \Rightarrow bool$
begin
  definition $dfs :: (\forall v set \times bool) nres$ where
  $dfs \equiv do \{$
    $(V,r) \leftarrow RECT (\lambda dfs (V,v)).$
    if $v \in V$ then RETURN $(V,\text{False})$
    else do {
      let $V = insert v V$;
      if tgt $v$ then
        RETURN $(V,\text{True})$
      else
        FOREACH$_C (E^* \{v\}) (\lambda (-,b). \neg b) \ (\lambda v' (V,\_). \ dfs (V,v')) (V,\text{False})$
    }
  ) (({},v0);
  RETURN $(V,r)\}$

  definition reachable $\equiv \{ v. (v0,v) \in E^* \}$
definition dfs-spec ≡ SPEC (λ(V,r). (r ←→ reachable∩Collect tgt≠{}) ∧ (¬r → V=reachable))

lemma dfs-correct:
  assumes fr: finite reachable
  shows dfs ≤ dfs-spec
proof −
  have F: ∀v. v∈reachable ⇒ finite (E''{v})
    using fr
  apply (auto simp: reachable-def)
  by (metis (mono-tags) Image-singleton Image-singleton-iff
    finite-subset rtrancl.rtrancl-into-rtrancl subsetI)

define rpre where rpre = (λS (V,v).
  v∈reachable
  ∧ V⊆reachable
  ∧ S⊆V
  ∧ (V ∩ Collect tgt = {})
  ∧ E"(V−S) ⊆ V)

define rpost where rpost = (λS (V,v) (V',r).
  (r←→V'∩Collect tgt ≠ {})
  ∧ V⊆V'
  ∧ v∈V'
  ∧ V'⊆reachable
  ∧ (¬r → (E"(V'−S) ⊆ V')))}

define fe-inv where fe-inv = (λS V v it (V',r).
  (r←→V∩Collect tgt ≠ {})
  ∧ insert v V⊆V'
  ∧ E"{v} − it ⊆ V'
  ∧ V'⊆reachable
  ∧ S⊆insert v V
  ∧ (¬r → E"(V'−S) ⊆ V' ∪ it ∧ E"(V'−insert v S) ⊆ V'))

have vc-pre-initial: rpre {} ({{}, v0)
  by (auto simp: rpre-def reachable-def)

{ fix S V v
  assume rpre S (V,v)
  and v∈V
  hence rpost S (V,v) (V,False)
  unfolding rpre-def rpost-def
  by auto
} note vc-node-visited = this
\{ 
fix S V v 
assume tgt v 
and rpre S (V, v) 
hence rpost S (V, v) (insert v V, True) 
  unfolding rpre-def rpost-def 
by auto 
\} note vc-node-found = this 

\{ 
fix S V v 
assume rpre S (V, v) 
hence finite (E'' \{ v \}) 
  unfolding rpre-def using F by (auto) 
\} note vc-foreach-finite = this 

\{ 
fix S V v 
assume A: v \notin V \neg tgt v 
  and PRE: rpre S (V, v) 
hence fe-inv S V v (E'' \{ v \}) (insert v V, False) 
  unfolding fe-inv-def rpre-def by (auto) 
\} note vc-enter-foreach = this 

\{ 
fix S V v v' it V' 
assume A: v \notin V \neg tgt v v' \in it it \subseteq E'' \{ v \} 
  and FEI: fe-inv S V v it (V', False) 
  and PRE: rpre S (V, v) 

from A have v' \in E'' \{ v \} by auto 
moreover from PRE have v \in reachable by (auto simp: rpre-def) 
hence E'' \{ v \} \subseteq reachable by (auto simp: reachable-def) 
ultimately have [simp]: v' \in reachable by blast 

have rpre (insert v S) (V', v') 
  unfolding rpre-def 
  using FEI PRE by (auto simp: fe-inv-def rpre-def) [] 
\} note vc-rec-pre = this 

\{ 
fix S V V' v v' it Vr'' 
assume fe-inv S V v it (V', False) 
  and rpost (insert v S) (V', v') Vr'' 

\}
hence \text{fe-inv } S \ V \ v \ (it - \{v’\}) \ V’’

unfolding \text{rpre-def rpost-def fe-inv-def}
by clarsimp blast

} \text{ note vc-iterate-foreach = this}

\{
fix S \ V \ v \ V’
assume \text{PRE: } rpre \ S \ (V, v)
assume A: v \notin \ V \neg \text{tgt } v
assume \text{FEI: } \text{fe-inv } S \ V \ v \ \{\} \ (V’, \ False)
have \text{rpost } S \ (V, v) \ (V’, \ False)
unfolding \text{rpost-def}
using \text{FEI} by (auto simp: \text{fe-inv-def})

} \text{ note vc-foreach-completed-imp-post = this}

\{
fix S \ V \ v \ V’ \ it
assume \text{PRE: } rpre \ S \ (V, v)
and A: v \notin \ V \neg \text{tgt } v \ it \subseteq E'' \{v\}
and \text{FEI: } \text{fe-inv } S \ V \ v \ it \ (V’, \ True)
hence \text{rpost } S \ (V, v) \ (V’, \ True)
by (auto simp add: \text{rpre-def rpost-def fe-inv-def})

} \text{ note vc-foreach-interrupted-imp-post = this}

\{
fix V \ r
assume \text{rpost } \{\} \ (\{\}, v0) \ (V, \ r)
hence (r \leftrightarrow \text{reachable} \cap \text{Collect tgt } \neq \{\}) \land (\neg r \rightarrow V = \text{reachable})
by (auto
simp: \text{rpost-def reachable-def}
dest: \text{Image-closed-trancl}
intro: rev-ImageI)

} \text{ note vc-rpost-imp-spec = this}

show \text{?thesis}
unfolding \text{dfs-def dfs-spec-def}
apply (refine-reg refine-reg)
apply (rule order-trans)
apply(rule RECT-rule-arb
where
pre=\text{rpre}
and M=\lambda a \ x. \ \text{SPEC } (rpost a \ x)
and V=\text{finite-psupset reachable } \langle *lex* \rangle \ \{\}
)
apply refine-mono
apply (blast intro: fr)
apply (rule vc-pre-initial)

446
apply (refine-rcg refine-vcg
   FOREACHc-rule [where I = fe-inv S v s for S v s]
   )
apply (simp-all add: vc-node-visited vc-node-found)
apply (simp add: vc-foreach-finite)
apply (auto intro: vc-enter-foreach)
apply (rule order-trans)
apply (rprems)
apply (erule (5) vc-rec-pre)
   apply (auto simp add: fe-inv-def finite-psupset-def)
apply (auto simp add: vc-iterate-foreach)
apply (auto simp add: vc-foreach-completed-imp-post)
apply (auto simp add: vc-foreach-interrupted-imp-post)
apply (auto dest: vc-rpost-imp-spec)
qed
end

lemma dfs-correct': (uncurry2 dfs, uncurry2 dfs-spec)
   ∈ [λ((E,s),t). finite (reachable E s)]f ((Id × r Id) × r Id) → (Id)nres-rel
apply (intro frefl nres-refl; clarsimp)
by (rule dfs-correct)

5.2.2 Refinement to Imperative/HOL

We set up a schematic proof goal, and use the sepref-tool to synthesize the
implementation.

sepref-definition dfs-impl is
   uncurry2 dfs :: (adjg-assn nat-assn)k*a nat-assnk*a(pure (nat-rel → bool-rel))k
   a prod-assn (ias_assn nat-assn) bool-assn
unfolding dfs-def [abs-def] — Unfold definition of DFS
using [[goals-limit = 1]]
apply (rewrite in RECT - (Π,-) ias.fold-custom-empty) — Select impls
apply (rewrite in if ⊥ then RETURN (⊥,True) else - fold-pho-apply)
apply sepref — Invoke sepref-tool
done
export-code dfs-impl checking SML-imp
— Generate SML code with Imperative/HOL

export-code dfs-impl in Haskell module-name DFS
Finally, correctness is shown by combining the generated refinement theorem with the abstract correctness theorem.

**lemmas** dfs-impl-correct' = dfs-impl.refine[FCOMP dfs-correct']

**corollary** dfs-impl-correct:

finite (reachable E s) \implies
<adjg-assn nat-assn E Ei>

dfs-impl Ei s tgt

<λ(Vi,r). \exists_A V. adjg-assn nat-assn E Ei * ias.assn nat-assn V Vi * \Uparrow((r \leftrightarrow reachable E s \cap Collect tgt ≠ {})) ∧ (¬r \rightarrow V = reachable E s)) \Uparrow_t

using dfs-impl-correct'[THEN hrefD, THEN hn-refineD, of ((E,s),tgt) ((Ei,s),tgt), simplified]

apply (rule cons-rule[rotated −1])

apply (sep-auto intro!: ent-ex-preI simp: dfs-spec-def pure-def)+

done

end

5.3 Imperative Implementation of Dijkstra’s Shortest Paths Algorithm

theory Sepref-Dijkstra

imports
../IICF/IICF
../Sepref-ICF-Bindings
Dijkstra-Shortest-Path.Dijkstra
Dijkstra-Shortest-Path.Test
HOL-Library.Code-Target-Numeral

Sepref-WGraph

begin

instantiation infy :: (heap) heap

begin

instance

apply standard

apply (rule-tac x=\lambdaInfy ⇒ 0 | Num a ⇒ to-nat a + 1 in exI)

apply (rule injI)

apply (auto split: infy.splits)

done

end

fun infy-assn where

infy-assn A (Num x) (Num y) = A x y
\[ \text{infty-assn } \text{A } \text{Infty} \text{ Infty} = \text{emp} \]
\[ \text{infty-assn } - - - = \text{false} \]

Connection with infty-rel

\textbf{lemma} \text{infty-assn-pure-conv}: \text{infty-assn } \text{(pure } \text{A} \text{)} = \text{pure } \text{(}(\text{A})\text{infty-rel})

\text{apply} (\text{intro ext})

\text{subgoal for } x \text{ y by} (\text{cases } x; \text{ cases } y; \text{ simp add: pure-def})

\text{done}

\textbf{lemmas} \text{[sepref-import-rewrite, fcomp-norm-unfold, sepref-frame-normrel-eqs]} = \text{infty-assn-pure-conv}\text{[symmetric]}

\textbf{lemmas} \text{[constraint-simps]} = \text{infty-assn-pure-conv}

\textbf{lemma} \text{infty-assn-pure} \text{[safe-constraint-rules]}: \text{is-pure } \text{A} = \text{⇒} \text{is-pure } \text{(infty-assn } \text{A})

\text{by} (\text{auto simp: is-pure-conv infty-assn-pure-conv})

\textbf{lemma} \text{infty-assn-id} \text{[simp]}: \text{infty-assn } \text{id-assn} = \text{id-assn}

\text{by} (\text{simp add: infty-assn-pure-cone})

\textbf{lemma} \text{[safe-constraint-rules]}: \text{IS-BELOW-ID } \text{R} = \text{⇒} \text{IS-BELOW-ID } \text{(}(\text{R})\text{infty-rel})

\text{by} (\text{auto simp: infty-rel-def IS-BELOW-ID-def})

\textbf{sepref-register} \text{Num } \text{Infty}

\textbf{lemma} \text{Num-hnr} \text{[sepref-fr-rules]}: \text{(}\text{return } \text{Num,RETURN } \text{o } \text{Num}\text{)}\in A^d \rightarrow_\alpha \text{ infty-assn } \text{A}

\text{by} \text{sepref-to-hoare sep-auto}

\textbf{lemma} \text{Infty-hnr} \text{[sepref-fr-rules]}: \text{((uncurry0 } \text{ (return } \text{Infty),uncurry0 } \text{(RETURN } \text{Infty))}\in \text{unit-assn}^k \rightarrow_\alpha \text{ infty-assn } \text{A}

\text{by} \text{sepref-to-hoare sep-auto}

\textbf{sepref-register} \text{case-infty}

\textbf{lemma} \text{[sepref-monadify-arity]}: \text{case-infty } \equiv \lambda_2 \text{f1 f2 x. SP case-infty}\$1$$(\lambda_2 x. f2\$x)\$x$

\text{by simp}

\textbf{lemma} \text{[sepref-monadify-comb]}: \text{case-infty}\$1$$(\lambda_2 x. f2\$x)$

\equiv (\equiv)$(\text{EVAL}\$x)$(\lambda_2 x. SP \text{case-infty}$1$(\lambda_2 x. f2\$x)\$x)$

\text{by simp}

\textbf{lemma} \text{[sepref-monadify-comb]}: \text{EVAL}\$(\text{case-infty}$1$(\lambda_2 x. f2\$x)\$x)$

\equiv (\equiv)$(\text{EVAL}\$x)$(\lambda_2 x. SP \text{case-infty}$1$(\text{EVAL } f1)$(\lambda_2 x. EVAL $ f1)$$(\text{EVAL } f2\$x)\$x)$

\text{apply} (\text{rule eq-reflection})

\text{by} (\text{simp split: infty.splits})

\textbf{lemma} \text{infty-assn-ctxt} \text{: infty-assn } \text{A x y = z } \equiv \text{hn-ctxt} \text{ (infty-assn } \text{A) x y = z}

\text{by} (\text{simp add: hn-ctxt-def})

\textbf{lemma} \text{infty-cases-hnr} \text{[sepref-prep-comb-rule, sepref-comb-rules]}:

\text{fixes } \text{A e e'}
defines simp: \( \text{INVe} \equiv \text{hn-invalid} (\text{infty-assn} \ A) \ e \ e' \)
assumes FR: \( \Gamma \vdash (\text{hn-context} (\text{infty-assn} \ A) \ e \ e' * F) \)
assumes Infty: \( [\ e = \text{Infty}; \ e' = \text{Infty}] \implies \text{hn-refine} (\text{hn-context} (\text{infty-assn} \ A) \ e \ e' * F) \)
assumes Num: \( [\ x1 \ x1a. \ [\ e = \text{Num} \ x1; \ e' = \text{Num} \ x1a] \implies \text{hn-refine} (\text{hn-context} \ A \ x1 \ x1a * \text{INVe} * F) \ ) (\text{hn-context} \ A' \ x1 \ x1a) \ ) \)
shows \( \text{hn-refine} \ \Gamma \ (\text{case-infty} \ f1' \ f2' \ e' \ (\text{hn-context} \ XX1 \ e \ e' * F)) \ ) \ ) \)
assumes MERGE2[unfolded hn-context-def]: \( \Gamma_1' \lor A \Gamma_2' \implies \Gamma' \)
apply (rule hn-refine-cons-pre[OF FR])
apply1 extract-hnr-invalids
apply (cases e; cases e'; simp add: infty-assn.simps[THEN infty-assn-context])
subgoal
apply (rule hn-refine-cons[OF - Infty - entt-refl]; assumption?)
applyS (simp add: hn-context-def)
apply (subst mult.commute, rule entt-fr-drop)
apply (rule entt-trans[OF - MERGE2])
apply (simp add:)
done
subgoal
apply (rule hn-refine-cons[OF - Num - entt-refl]; assumption?)
applyS (simp add: hn-context-def)
apply (rule entt-star-mono)
apply1 (rule entt-fr-drop)
applyS (simp add: hn-context-def)
apply1 (rule entt-trans[OF - MERGE2])
applyS (simp add:)
done
done

lemma hnr-val[sepref-fr-rules]: \( (\text{return} \ o \ \text{Weight.val}, \text{RETURN} \ o \ \text{Weight.val}) \in \ \\
\{x. \ x \neq \text{Infty}\}_a (\text{infty-assn} \ A)^d \rightarrow A \)
apply sepref-to-hoare
subgoal for x y by (cases x; cases y; sep-auto)
done

context
fixes A :: 'a::weight \Rightarrow 'b \Rightarrow \text{assn}
fixes plusi
assumes GA[unfolded GEN-ALGO-def, sepref-fr-rules]: \( \text{GEN-ALGO} \ \text{plusi} (\lambda f. \ ) \ (\text{uncurry} \ f, \text{uncurry} \ (\text{RETURN} oo (+)))(A^k * A^k \to A) \)
begin
sepref-thm infty-plus-impl is uncurry (\text{RETURN} oo (+)) \:: ((\text{infty-assn} A)^k *_a (\text{infty-assn} A)^k \to_\text{a} \text{infty-assn} A) \\
\text{unfolding infty-plus-\text{eq}+\text{plus}[\text{symmetric}] \text{infty-plus-def}[\text{abs-def}] \text{by sepref} \\
\text{end}
concrete-definition infty-plus-impl uses infty-plus-impl.refine-raw is (uncurry
lemmas [sepref-fr-rules] = infty-plus-impl.refine

definition infty-less where
  infty-less lt a b ≡ case (a, b) of (Num a, Num b) ⇒ lt a b | (Num -, Infty) ⇒ True | - ⇒ False

lemma infty-less-param [param] :
  (infty-less, infty-less) ∈ (R → R → bool-rel) → ⟨ R ⟩ infty-rel → ⟨ R ⟩ infty-rel → bool-rel
  unfolding infty-less-def [abs-def]
  by parametricity

lemma infty-less-eq-less : infty-less (<) = (<)
  unfolding infty-less-def [abs-def]
  apply (clarsimp intro ! : ext)
  subgoal for a b by (cases a ; cases b ; auto)
  done

context
  fixes A :: 'a::weight ⇒ 'b ⇒ assn
  fixes lessi
  assumes GA [unfolded GEN-ALGO-def, sepref-fr-rules] :
    GEN-ALGO lessi (λf.
      (uncurry f, uncurry (RETURN oo (<)))) ∈ A⁺ A⁺ A⁺ ∗ A⁺ A⁺ → A⁺ A⁺ bool-assn)
begin
  sepref-thm infty-less-impl is uncurry (RETURN oo (<)) :: ((infty-assn A)⁺ A⁺ → A⁺ bool-assn)
  unfolding infty-less-eq-less [symmetric] infty-less-def [abs-def]
  by sepref
end

concrete-definition infty-less-impl uses infty-less-impl.refine-raw is (uncurry ?impl,-) ∈-

lemmas [sepref-fr-rules] = infty-less-impl.refine

lemma param-mpath' : (mpath', mpath') ∈ ⟨ ⟨ A × r B × r A ⟩ list-rel × r ⟨ B ⟩ list-rel ⟩ option-rel
  option-rel
proof –
  have 1 : mpath' = map-option fst
  apply (intro ext, rename-tac x)
  apply (case-tac x)
  apply simp
  apply (rename-tac a)
  apply (case-tac a)
  apply simp
  done
  show ?thesis
  unfolding 1
  by parametricity
qed

lemmas (in −) [sepref-import-param] = param-mpath'
lemma param-mpath-weight':
  (mpath-weight', mpath-weight) ∈ ⟨⟨A×r × r, B⟩list-rel × r, B⟩option-rel → (B)infty-rel
by (auto elim!: option-relE simp: infty-rel-def top-infty-def)
lemmas [sepref-import-param] = param-mpath-weight'

context Dijkstra begin
lemmas impl-aux = mdijkstra-def[unfolded mdinit-def mpop-min-def mupdate-def]

lemma mdijkstra-correct:
  (mdijkstra, SPEC (is-shortest-path-map v0)) ∈ ⟨br α r res-invar⟩nres-rel
proof
  note mdijkstra-refines
  also note dijkstra'-refines
  also note dijkstra-correct
finally show ?thesis
  by (rule nres-relI)
qed

locale Dijkstra-Impl =
fixes w-dummy :: 'W:: {weight, heap}
begin
Weights

sepref-register 0::'W
lemmas [sepref-import-param] = IdI[of 0::'W]
abbreviation weight-assn ≡ id-assn :: 'W ⇒ id

lemma w-plus-param: ((+), (+)::'W⇒) ∈ Id ⇒ Id ⇒ Id by simp
lemma w-less-param: ((<), (<)::'W⇒) ∈ Id ⇒ Id ⇒ Id by simp
lemmas [sepref-import-param] = w-plus-param w-less-param
lemma [sepref-gen-algo-rules]:
  GEN-ALGO (return oo (λf. (uncurry f, uncurry (RETURN oo (+))))) ∈ id-assn * a id-assn ⇒ id-assn
  GEN-ALGO (return oo (λf. (uncurry f, uncurry (RETURN oo (<))))) ∈ id-assn * a id-assn ⇒ id-assn
  by (sep-auto simp: GEN-ALGO-def pure-def intro!: hrefI hn-refineI)+

lemma conv-prio-pop-min: prio-pop-min m = do { ASSERT (dom m ≠ {});
  ((k,v),m) ← mop-pm-pop-min id m;
  RETURN (k,v,m)
}
unfolding prio-pop-min-def mop-pm-pop-min-def
by (auto simp: pw-eq-iff refine-pw-simps ran-def)

end
interpretation Dijkstra-Impl w-dummy.

definition drmap-assn2 ≡ IICF-Sepl-Binding.iam-assn
(pure (node-rel N))
(prod-assn
  (list-assn (prod-assn (pure (node-rel N)) (prod-assn weight-assn (pure (node-rel N))))))
weight-assn)

concrete-definition mdijkstra uses Dijkstra.impl-aux

sepref-definition dijkstra-imp is uncurry mdijkstra'
:: (is-graph N (Id::(′W::{heap,weight} set)) k → a
  drmap-assn2
unfolding mdijkstra'-def
apply (subst conv-prio-pop-min)
apply (rewrite in RETURN (∧, , ) iam.fold-custom-empty)
apply (rewrite hm-fold-custom-empty-sz [of N])
unfolding drmap-assn2-def
using [[id-debug, goals-limit = 1]]
by sepref
export-code dijkstra-imp checking SML-imp

The main correctness theorem

thm Dijkstra.mdijkstra-correct

lemma mdijkstra'-aref: (uncurry mdijkstra', uncurry (SPEC oo weighted-graph.is-shortest-path-map))
∈ [λ(G, v0). Dijkstra G v0] Id×,Id → (br Dijkstra.or Dijkstra.res-invarm)res-rel
using Dijkstra.mdijkstra-correct
by (fastforce intro!::frefI simp::mdijkstra'.refine[symmetric])

definition drmap-assn N ≡ hr-comp (drmap-assn2 N) (br Dijkstra.or Dijkstra.res-invarm)

context notes [fcomp-norm-unfold] = drmap-assn-def[symmetric] begin

theorem dijkstra-imp-correct: (uncurry (dijkstra-imp N), uncurry (SPEC oo weighted-graph.is-shortest-path-map))
∈ [λ(G, v0). v0 ∈ nodes G ∧ (∀(v, w, v') ∈ edges G. 0 ≤ w)]a (is-graph N Id)k
*a (node-assn N)k → drmap-assn N
apply (rule href-weaken-pre[OF - dijkstra-imp.refine[FCOMP mdijkstra'-aref]]]
proof clarsimp
fix G :: (nat, w::{weight,heap}) graph and v0
assume v0-is-node: v0 ∈ nodes G
  and nonneg-weights: ∀ (v, w, v') ∈ edges G. 0 ≤ w

453
and \( v_0 < N \)
and \( \text{RDOM: rdomp (is-graph } N \text{ Id) } G \)

from \( \text{RDOM interpret valid-graph } G \text{ unfolding is-graph-def rdomp-def by auto} \)

from \( \text{RDOM have [simp]: finite } V \text{ unfolding is-graph-def rdomp-def by auto} \)

from \( \text{RDOM have } \forall v \in V. \{(w, v'). (v, w, v') \in E\} \in \text{Range } (\{\text{Id } \times_r \text{ node-rel } N\})\text{list-set-rel} \)
by (auto simp: succ-def is-graph-def rdomp-def)

hence \( \forall v \in V. \text{finite } \{(w, v'). (v, w, v') \in E\} \)
unfolding list-set-rel-range by simp
hence finite \( (\Sigma V (\lambda v. \{(w, v'). (v, w, v') \in E\})) \)
by auto
also have \( E \subseteq (\Sigma V (\lambda v. \{(w, v'). (v, w, v') \in E\})) \)
using \( \text{E-valid} \)
by auto
finally \( (\text{finite-subset[rotated]}) \) have [simp]: finite \( E \).

show \( \text{Dijkstra } G \) \( v_0 \)
apply (unfold-locales)
unfolding is-graph-def using \( \text{v0-is-node nonneg-weights} \)
by auto
qed

end

corollary \( \text{dijkstra-imp-rule:} \)
\[ <\text{is-graph } N \text{ Id } G \text{ Gi } * \uparrow (v_0 \in \text{nodes } G \land (\forall (v, w, v') \in \text{edges } G. 0 \leq w))> \]
dijkstra-imp \( n \) \( G \text{ v0} \)
\[ <\lambda m. \text{(is-graph } N \text{ Id) } G \text{ Gi} \]
\[ * (\exists \lambda m. \text{drmap-assn } n \text{ m } m \text{ } \uparrow (\text{weighted-graph.is-shortest-path-map } G \text{ v0 m}) >_t \]
using dijkstra-imp-correct[to-hnr, of \( v_0 \text{ G } n \text{ v0 Gi} \]
unfolding hn-refine-def
apply (clarsimp)
apply (erule cons-rule[rotated - 1])
apply (sep-auto simp: hn-context-def pure-def is-graph-def)
apply (sep-auto simp: hn-context-def)
done

end

454
5.4 Imperative Implementation of of Nested DFS (HPY-Improvement)

theory Sepref-NDFS
imports .. / Sepref Collections-Examples.Nested-DFS Sepref-Graph HOL-Library.Code-Target-Numeral

begin

sepref-decl-intf 'v i-red-witness is 'v list * 'v

lemma id-red-witness[id-rules]:
  red-init-witness :: TYPE('v ⇒ 'v ⇒ 'v i-red-witness option)
  prep-wit-red :: TYPE('v ⇒ 'v i-red-witness option ⇒ 'v i-red-witness option)
  by simp-all

definition red-witness-rel-def-internal: red-witness-rel R ≡ ⟨⟨ R ⟩ ⟩ list-rel prod-rel

lemma red-witness-rel-def: (R)red-witness-rel ≡ ⟨⟨ R ⟩ ⟩ list-rel prod-rel
  unfolding red-witness-rel-def-internal[abs-def] by (simp add: relAPP-def)

lemma red-witness-rel-sv[constraint-rules]:
  single-valued R ⇒ single-valued ⟨⟨ R ⟩ ⟩ red-witness-rel
  unfolding red-witness-rel-def
  by tagged-solver

lemma [sepref-fr-rules]: hn-refine
  (hn-val R u u' * hn-val R v v')
  (return (red-init-witness u' v'))
  (hn-val R u u' * hn-val R v v')
  (option-assn (pure ⟨⟨ R ⟩ ⟩ red-witness-rel)))
  (RETURN$(red-init-witness$u$v))
  apply simp
  unfolding red-init-witness-def
  apply rule
  apply (sep-auto simp: hn-ctxt-def pure-def red-witness-rel-def)
  done

lemma [sepref-fr-rules]: hn-refine
  (hn-val R u u' * hn-ctxt (option-assn (pure ⟨⟨ R ⟩ ⟩ red-witness-rel))) w w')
  (return (prep-wit-red u' w'))
  (hn-val R u u' * hn-ctxt (option-assn (pure ⟨⟨ R ⟩ ⟩ red-witness-rel))) w w')
  (option-assn (pure ⟨⟨ R ⟩ ⟩ red-witness-rel)))
  (RETURN$(prep-wit-red$u$w))
  apply rule
  apply (cases w)
apply (sep-auto simp: hn-ctxt-def pure-def red-witness-rel-def)
apply (cases w)
apply (sep-auto simp: hn-ctxt-def pure-def red-witness-rel-def)
done

term red-dfs

sepref-definition red-dfs-impl is
(uncurry2 (uncurry red-dfs))
:: (adjg-assn nat-assn)k *a (ias.assn nat-assn)k *a (ias.assn nat-assn)d *a nat-assnk
→a UNSPEC
unfolding red-dfs-def[abs-def]
using [[goals-limit = 1]]
by sepref
export-code red-dfs-impl checking SML-imp

declare red-dfs-impl.refine[sepref-fr-rules]

sepref-register red-dfs :: 'a i-graph ⇒ 'a set ⇒ 'a set ⇒ 'a
⇒ ('a set * 'a i-red-witness option) nres

lemma id-init-wit-blue[sepref-import-param]:
init-wit-blue :: TYPE('a ⇒ 'a i-red-witness option ⇒ 'a blue-witness)
by simp

lemma hn-blue-wit[sepref-import-param]:
(NO-CYC,NO-CYC) ∈ blue-wit-rel
(prep-wit-blue,prep-wit-blue) ∈ nat-rel → blue-wit-rel → blue-wit-rel
((=), (=)) ∈ blue-wit-rel → blue-wit-rel → bool-rel
by simp-all

lemma hn-init-wit-blue[sepref-fr-rules]: hn-refine
(hn-val nat-rel v v' * hn-ctxt (option-assn (pure ((nat-rel)red-witness-rel))) w w')
(return (init-wit-blue v' w'))
(hn-val nat-rel v v' * hn-ctxt (option-assn (pure ((nat-rel)red-witness-rel))) w w')
(pure blue-wit-rel)
(RETURN$(init-wit-blue$v$w))
apply rule
apply (sep-auto simp: hn-ctxt-def pure-def)
apply (case-tac w, sep-auto)
apply (case-tac w', sep-auto, sep-auto simp: red-witness-rel-def)
done

lemma hn-extract-res[sepref-import-param]:
(extract-res, extract-res) ∈ blue-wit-rel → Id
by simp
We tweak the initialization vector of the outer DFS, to allow pre-initialization of the size of the array-lists. When set to the number of nodes, array-lists will never be resized during the run, which saves some time.
lemma testsuite-blue-dfs-modify:
\[(\{\} \cdot \text{nat set}, \{\} \cdot \text{nat set}, \{\} \cdot \text{nat set}, s) = (\text{op-ias-empty-sz } N, \text{op-ias-empty-sz } N, \text{op-ias-empty-sz } N, s)\]
by simp

sepref-definition blue-dfs-impl-sz is uncurry2 blue-dfs :: ((adjg-assn nat-assn)^k \cdot (ias.assn) \cdot (adjg-assn nat-assn)^k \cdot id-assn)
unfolding blue-dfs-def[abs-def]
apply (rewrite in RECT - \Sigma testsuite-blue-dfs-modify)
using [[goals-limit = 1]]
by sepref
export-code blue-dfs-impl-sz checking SML-imp
end

lemmas blue-dfs-impl-sz-correct' = blue-dfs-impl-sz.refine[FCOMP blue-dfs-correct']

term blue-dfs-impl-sz

theorem blue-dfs-impl-sz-correct:
fixes E
assumes finite (E^* \cdot \{v0\})
shows <\text{ias.assn id-assn } A \text{ adjg-assn id-assn } E \text{ succ-impl}>
\text{blue-dfs-impl-sz } N \text{ succ-impl } A \text{ adjg-assn } v0
\langle\lambda v. \text{ias.assn id-assn } A \text{ adjg-assn id-assn } E \text{ succ-impl} \rangle
* \uparrow\)
  \text{case } r \text{ of None } \Rightarrow \text{ahas-acc-cycle } E \text{ A } v0
  \mid \text{Some } (v,pc,pv) \Rightarrow \text{is-acc-cycle } E \text{ A } v0 \text{ v } pv \text{ pc}
using blue-dfs-impl-sz-correct[THEN horefD, THEN hn-refineD, of (E,A,v0)]
((\text{succ-impl },A\text{-impl},v0), simplified]
apply (rule cons-rule[rotated -1])
using assms
by (sep-auto simp: blue-dfs-spec-def pure-def)
end

5.5 Generic Worklist Algorithm with Subsumption

theory Worklist-Subsumption
  imports ../Sepref
begin

5.5.1 Utilities

definition take-from-set where
take-from-set s = ASSERT (s \neq \{\}) \Rightarrow SPEC (\lambda (x, s'). x \in s \land s' = s - \{x\)

458


lemma take-from-set-correct:
  assumes $s \neq \emptyset$
  shows $\text{take-from-set } s \leq \text{SPEC } (\lambda (x, s'). x \in s \land s' = s - \{x\})$
  using assms unfolding take-from-set-def by simp

lemmas [refine-vcg] = take-from-set-correct[THEN order.trans]

definition take-from-mset where
take-from-mset $s = \text{ASSERT } (s \neq \#\emptyset) \Rightarrow \text{SPEC } (\lambda (x, s'). x \in\# s \land s' = s - \{\#x\})$

lemma take-from-mset-correct:
  assumes $s \neq \#\emptyset$
  shows $\text{take-from-mset } s \leq \text{SPEC } (\lambda (x, s'). x \in\# s \land s' = s - \{\#x\})$
  using assms unfolding take-from-mset-def by simp

lemmas [refine-vcg] = take-from-mset-correct[THEN order.trans]

lemma set-mset-mp: set-mset $m \subseteq \{s \Rightarrow n < \text{count } m x \Rightarrow x \in s$
by (meson count-greater-zero-iff le-less-trans subsetCE zero-le)

lemma pred-not-lt-is-zero: ($\neg n - \text{Suc 0} < n)$ $\iff$ $n = 0$
by auto

5.5.2 Search Spaces

A search space consists of a step relation, a start state, a final state predicate, and a subsumption preorder.

locale Search-Space-Defs =
  fixes $E : 'a \Rightarrow 'a \Rightarrow \text{bool}$ — Step relation
  and $a_0 : 'a$ — Start state
  and $F : 'a \Rightarrow \text{bool}$ — Final states
  and subsumes : $'a \Rightarrow 'a \Rightarrow \text{bool}$ (infix $\leq$ 50) — Subsumption preorder
begin

definition reachable where
  reachable $= E^{**} a_0$

definition $F$-reachable $\equiv \exists a. \text{reachable } a \land F a$
end

The set of reachable states must be finite, subsumption must be a preorder, and be compatible with steps and final states.

locale Search-Space = Search-Space-Defs +
  assumes finite-reachable: finite $\{a. \text{reachable } a\}$

459
assumes refl[intro!, simp]: \( a \leq a \)
and trans[trans]: \( a \leq b \implies b \leq c \implies a \leq c \)

assumes mono: \( a \leq b \implies E a a' \implies \text{reachable } a \implies \text{reachable } b \implies \exists b'. E b b' \land a' \leq b' \)
and F-mono: \( a \leq a' \implies F a \implies F a' \)

begin

lemma start-reachable[intro!, simp]:
reachable \( a_0 \)

unfolding reachable-def by simp

lemma step-reachable:
assumes reachable \( a \) \( E a a' \)
shows reachable \( a' \)
using assms unfolding reachable-def by simp

lemma finitely-branching:
assumes reachable \( a \)
shows finite \( (\text{Collect } (E a)) \)
by (metis assms finite-reachable finite-subset mem-Collect-eq step-reachable subsetI)

end

5.5.3 Worklist Algorithm

term card

custom Search-Space-Defs begin
definition worklist-var = inv-image (finite-psupset (Collect reachable) \(<lex>\) measure size) \((\lambda (a, b, c). (a, b))\)

definition worklist-inv-frontier passed wait =
\( (\forall a \in \text{passed}. \forall a'. E a a' \implies (\exists b' \in \text{passed} \cup \text{set-mset wait}. a' \leq b')) \)

definition start-subsumed passed wait = \( (\exists a \in \text{passed} \cup \text{set-mset wait}. a_0 \leq a) \)

definition worklist-inv \equiv \lambda (\text{passed}, \text{wait}, \text{brk}).
passed \subseteq \text{Collect reachable} \land
(brk \implies (\exists f. \text{reachable } f \land F f)) \land
(\neg brk \implies \text{worklist-inv-frontier passed wait} \land (\forall a \in \text{passed} \cup \text{set-mset wait}. \neg F a))
∧ start-subsumed passed wait
∧ set-mset wait ⊆ Collect reachable)

**definition** add-succ-spec wait a ≡ SPEC (λ(wait’,brk).
if ∃ a’, E a a’ ∧ F a’ then
  brk
else set-mset wait’ = set-mset wait ∪ {a’. E a a’} ∧ ¬brk
)

**definition** worklist-algo where
worklist-algo = do
{ if F a₀ then RETURN True
else do 
  let passed = {};
  let wait = {#a₀#};
  (passed, wait, brk) ← WHILEIT worklist-inv (λ (passed, wait, brk). ¬ brk
∧ wait ≠ {#})
  (λ (passed, wait, brk). do 
  { (a, wait) ← take-from-mset wait;
    ASSERT (reachable a);
    if (∃ a’ ∈ passed. a ⪯ a’) then RETURN (passed, wait, brk) else 
    do 
    { (wait, brk) ← add-succ-spec wait a;
      let passed = insert a passed;
      RETURN (passed, wait, brk)
    }
  }
  )
  (passed, wait, False);
  RETURN brk
}

end

Correctness Proof

**context** Search-Space begin

**lemma** wf-worklist-var:
wf worklist-var

unfolding worklist-var-def by (auto simp: finite-reachable)

**context**
begin

private lemma aux1:
  assumes \( \forall x \in \text{passed.} \neg a \preceq x \)
  and \( \text{passed} \subseteq \text{Collect reachable} \)
  and \( \text{reachable} a \)
  shows
  \[(\text{insert} a \text{ passed}, \text{wait}', \text{brk}'), \text{passed, wait, brk}) \in \text{worklist-var} \]
proof –
  from assms have \( a \notin \text{passed} \) by auto
  with assms(2,3) show ?thesis
  by (auto simp: worklist-inv-def worklist-var-def finite-psupset-def)
qed

private lemma aux2:
  assumes \( a' \in \text{passed} \)
  \( a \preceq a' \)
  \( a \in \# \text{ wait} \)
  worklist-inv-frontier \( \text{passed} \)
  shows worklist-inv-frontier \( \text{passed} (\text{wait} - \{\# a\}) \)
using assms unfolding worklist-inv-frontier-def
using trans
apply clarsimp
by (metis Un-iff insert-DiffM2 local.trans mset-right-cancel-elem)

private lemma aux5:
  assumes \( a' \in \text{passed} \)
  \( a \preceq a' \)
  \( a \in \# \text{ wait} \)
  start-subsumed \( \text{passed} \)
  shows start-subsumed \( \text{passed} (\text{wait} - \{\# a\}) \)
using assms unfolding start-subsumed-def apply clarsimp
by (metis Un-iff insert-DiffM2 local.trans mset-right-cancel-elem)

private lemma aux3:
  assumes \( \text{set-mset} \text{ wait} \subseteq \text{Collect reachable} \)
  \( a \in \# \text{ wait} \)
  \( \text{set-mset} \text{ wait}' = \text{set-mset} (\text{wait} - \{\# a\}) \cup \text{Collect} (E a) \)
  worklist-inv-frontier \( \text{passed} \)
  shows worklist-inv-frontier \( \text{insert} a \text{ passed} \text{ wait}' \)
proof –
  from assms(1,2) have reachable \( a' \)
  by (simp add: subset-iff)
with finitely-branching have [simp, intro!]: finite (Collect (E a)) .

from assms(2,3,4) show thesis unfolding worklist-inv-frontier-def
by (metis Un-iff insert-DiffM insert-iff local refl mem-Collect-eq set-mset-add-mset-insert)
qed

private lemma aux6:
assumes
  a ∈ # wait
  start-subsumed passed wait
  set-mset wait' = set-mset (wait − {#a#}) ∪ Collect (E a)
shows start-subsumed (insert a passed) wait'
using assms unfolding start-subsumed-def
by (metis Un-iff insert-DiffM insert-iff set-mset-add-mset-insert)

lemma aux4:
assumes worklist-inv-frontier passed {#} reachable x start-subsumed passed {#}
passed ⊆ Collect reachable
shows ∃ x' ∈ passed. x ≤ x'
proof −
from ⟨reachable x⟩ have E** a0 x by (simp add: reachable-def)
from assms(3) obtain b where a0 ≤ b ∈ passed unfolding start-subsumed-def
by auto
  have ∃ x'. ∃ x''. E** b x' x ≤ x' x'' ∈ passed if
    E** a a ≤ b b ≤ b' b' ∈ passed
    reachable a reachable b for a b b'
  using that proof (induction arbitrary: b b' rule: converse-rtranclp-induct)
case base
  then show ?case by auto
next
  case ⟨step a a1 b b'⟩
  from ⟨E a a1⟩ ⟨a ≤ b⟩ ⟨reachable a⟩ ⟨reachable b⟩ obtain b1 where
    E b b1 a1 ≤ b1
  using mono by blast
  then obtain b1' where E b' b1' b1 ≤ b1' using assms(4) mono step.prems
by blast
  with ⟨b' ∈ passed⟩ assms(1) obtain b1'' where b1'' ∈ passed b1' ≤ b1''
unfolding worklist-inv-frontier-def by auto
  with ⟨b1 ≤ a⟩ have b1 ≤ b1'' using trans by blast
  with step.IH[OF ⟨a1 ≤ b1⟩ this ⟨b1'' ∈ passed⟩] ⟨reachable a⟩ ⟨E a a1⟩ ⟨reachable b'′⟩ ⟨E b b1''⟩
  obtain x' x'' where
    E'' b1 x' x ≤ x' x'' x'' ∈ passed
by (auto intro: step-reachable)
moreover from ⟨E b b1′⟩ ⟨E** b1' x'⟩ have E** b x' by auto
ultimately show ?case by auto
qed
from this[OF ⟨E** a0 x⟩ ⟨a0 ≤ b⟩ refl ⟨b ∈ -⟩] assms(4) ⟨b ∈ passed⟩ show
\textbf{theorem} worklist-algo-correct:
\[\text{worklist-algo} \leq \text{SPEC} (\lambda \text{brk}. \text{brk} \iff \text{F-reachable})\]
\textbf{proof} –
\begin{itemize}
  \item \textbf{note} [\text{simp}] = size-Diff-submset pred-not-lt-is-zero
  \item \textbf{note} [\text{dest}] = set-mset-mp
\end{itemize}
\textbf{show} \(?\text{thesis}\)
\textbf{unfolding} worklist-algo-def add-succ-spec-def F-reachable-def
\begin{itemize}
  \item \textbf{apply} (\text{refine-vcg wf-worklist-var})
  \item \textbf{apply} (auto; fail) []
  \item \textbf{apply} (auto simp: worklist-inv-def worklist-inv-frontier-def start-subsumed-def; fail)
  \item \textbf{apply} (simp; fail)
  \item \textbf{apply} (auto simp: worklist-inv-def; fail)
  \item \textbf{apply} (auto simp: worklist-inv-def aux2 aux5
    \begin{itemize}
      \item \textit{dest}: in-diffD
      \item \textit{split}: if-split-asm; fail) []
    \end{itemize}
  \item \textbf{apply} (auto simp: worklist-inv-def worklist-var-def intro; finite-subset[OF - finite-reachable]; fail)
  \item \textbf{apply} (clarsimp split: if-split-asm)
  \item \textbf{apply} (clarsimp simp: worklist-inv-def; blast intro: step-reachable; fail)
\end{itemize}
\begin{itemize}
  \item \textbf{apply} (auto
    \begin{itemize}
      \item \textit{simp}: worklist-inv-def step-reachable aux3 aux6 finitely-branching
      \item \textit{dest}: in-diffD; fail)
    \end{itemize}
  \item \textbf{apply} (auto simp: worklist-inv-def aux1; fail)
  \item \textbf{using} F-mono \textbf{apply} (fastforce simp: worklist-inv-def dest!: aux4)
\end{itemize}
\textbf{done}
\textbf{qed}
\begin{itemize}
  \item \textbf{lemmas} [\text{refine-vcg}] = worklist-algo-correct[THEN order-trans]
\end{itemize}
\textbf{end} — Context
5.5.4 Towards an Implementation

locale Worklist1-Defs = Search-Space-Defs +
  fixes succs :: 'a ⇒ 'a list

locale Worklist1 = Worklist1-Defs + Search-Space +
  assumes succs-correct: reachable a ⇒ set (succs a) = Collect (E a)

begin

definition add-succ1 wait a ≡ nfoldli (succs a) (λ(-, brk). ¬brk) (λa (wait, brk).
  if F a then RETURN (wait, True) else RETURN (wait + {#a#}, False)) (wait, False)

lemma add-succ1-ref [refine]: [(wait, wait')∈Id; (a,a')∈b-rel Id reachable] ⇒
  add-succ1 wait a ≤⇓ (Id ×, bool-rel) (add-succ-spec wait' a')
  apply simp
  unfolding add-succ-spec-def add-succ1-def
  apply (refine-vec nfoldli-rule[where I = λI - (wait', brk). if brk then ∃ a'. E a
    a' ∧ F a' else set-mset wait' = set-mset wait ∪ set I I ∧ set I I ∩ Collect F = {})]
  apply (auto; fail)
  using succs-correct[of a] apply (auto; fail)
  using succs-correct[of a] apply (auto; fail)
  apply (auto; fail)
  using succs-correct[of a] apply (auto; fail)
  done

definition worklist-algo1 where
  worklist-algo1 = do
  { if F a₀ then RETURN True
    else do 
      let passed = {};
      let wait = {#a₀#};
      (passed, wait, brk) ← WHILEIT worklist-inv (λ (passed, wait, brk). ¬ brk ∧ wait ≠ {#})
      (λ (passed, wait, brk). do
      { (a, wait) ← take-from-mset wait;
        if (∃ a' ∈ passed. a ≤ a') then RETURN (passed, wait, brk) else
        do 
        { (wait,brk) ← add-succ1 wait a;
          let passed = insert a passed;
          RETURN (passed, wait, brk)
        }
      })
  }

end — Search Space
(passed, wait, False);
RETURN brk
}
}

lemma worklist-algo1-ref[refine]: worklist-algo1 ≤ ⇓ Id worklist-algo

unfolding worklist-algo1-ref worklist-algo-def
apply (refine-rcg)
apply refine-dref-type
unfolding worklist-inv-def
apply auto
done

end

end

— Theory
theory Worklist-Subsumption-Impl
imports ../IICF/IICF Worklist-Subsumption
begin

locale Worklist2-Defs = Worklist1-Defs +
  fixes A :: 'a ⇒ 'ai ⇒ assn
  fixes succsi :: 'ai ⇒ 'ai list Heap
  fixes a0i :: 'ai Heap
  fixes Fi :: 'ai ⇒ bool Heap
  fixes Lei :: 'ai ⇒ 'ai ⇒ bool Heap

locale Worklist2 = Worklist2-Defs + Worklist1 +

  assumes [sepref-fr-rules]: (uncurry0 a0i, uncurry0 (RETURN (PR-CONST a0))) ∈ unit-assn
  assumes [sepref-fr-rules]: (Fi,RETURN o PR-CONST F) ∈ A^k →_a bool-assn
  assumes [sepref-fr-rules]: (uncurry Lei,uncurry (RETURN oo PR-CONST (≤))) ∈ A^k ×_a A^k →_a bool-assn
  assumes [sepref-fr-rules]: (succsi,RETURN o PR-CONST succs) ∈ A^k →_a list-assn

begin
  sepref-register PR-CONST a0 PR-CONST F PR-CONST (≤) PR-CONST succs

  lemma [def-pat-rules]:
  a0 ≡ UNPROTECT a0 F ≡ UNPROTECT F (≤) ≡ UNPROTECT (≤)
  succs ≡ UNPROTECT succs
  by simp-all

  lemma take-from-mset-as-mop-mset-pick: take-from-mset = mop-mset-pick
apply (intra ext)
unfolding take-from-mset-def[abs-def]
by (auto simp: pw-eq-iff refine-pw-simps)

lemma [safe-constraint-rules]: CN-FALSE is-pure A ⇒ is-pure A by simp

sepref-thm worklist-algo2 is uncurry0 worklist-algo1 :: unit-assnk →a bool-assn
unfolding worklist-algo1-def add-succ1-def
supply [[goals-limit = 1]]
apply (rewrite in Let ☩ lso-fold-custom-empty)
apply (rewrite in {#a0#} lms0-fold-custom-empty)
unfolding take-from-mset-as-mop-mset-pick fold-lso-bex
by sepref

end

concrete-definition worklist-algo2
for Lei a0i Fi succsi
uses Worklist2 worklist-algo2.refine-raw is (uncurry0 $?f,-)∈-
thm worklist-algo2-def

context Worklist2 begin
lemma Worklist2-this: Worklist2 E a0 F (≤) succ A succsi a0i Fi Lei
by unfold-locales

lemma hnr-F-reachable: (uncurry0 (worklist-algo2 Lei a0i Fi succsi), uncurry0 (RETURN F-reachable)) ∈ unit-assnk →a bool-assn
using worklist-algo2.refine[OF Worklist2-this,
FCOMP worklist-algo1-ref[THEN nres-relf],
FCOMP worklist-algo-correct[THEN Id-SPEC-refine, THEN nres-relf]]
by (simp add: RETURN-def)

end

context Worklist1 begin
sepref-decl-op F-reachable :: bool-rel .
lemma [def-pat-rules]: F-reachable ≡ op-F-reachable by simp

lemma hnr-op-F-reachable:
  assumes GEN-ALGO a0i (λa0i. (uncurry0 a0i, uncurry0 (RETURN a0)) ∈ unit-assn →a A)
  assumes GEN-ALGO Fi (λFi. (RETURN o F) ∈ Ak →a bool-assn)
  assumes GEN-ALGO Lei (λLei. (uncurry Lei,uncurry (RETURN oo (≤))) ∈ Ak *a Ak →a bool-assn)
  assumes GEN-ALGO succsi (λsuccsi. (succsi,RETURN o succs) ∈ Ak →a list-assn A)
  shows (uncurry0 (worklist-algo2 Lei a0i Fi succsi), uncurry0 (RETURN
(PR-CONST op-F-reachable)))
∈ unit-assn^k \to_a bool-assn
proof -
from assms interpret Worklist2 E a0 F (\subset) succ A succsi a0i Fi Lei
by (unfold-locales; simp add: GEN-ALGO-def)

from hnr-F-reachable show thesis by simp
qed

sepref-decl-impl hnr-op-F-reachable .
end

end

5.6 Non-Recursive Algebraic Datatype

theory Sepref-Snip-Datatype
imports ../../IICF/IICF
begin

We define a non-recursive datatype

datatype 'a enum = E1 'a | E2 'a | E3 | E4 'a | E5 bool 'a

5.6.1 Refinement Assertion

fun enum-assn where
enum-assn A (E1 x) (E1 x') = A x x'
| enum-assn A (E2 x) (E2 x') = A x x'
| enum-assn A (E3) (E3) = emp
| enum-assn A (E4 x y) (E4 x' y') = A x x' * A y y'
| enum-assn A (E5 x y) (E5 x' y') = bool-assn x x' * A y y'
| enum-assn - - - = false

You might want to prove some properties

A pure-rule is required to enable recovering of invalidated data that was not stored on the heap

lemma enum-assn-pure[safe-constraint-rules]: is-pure A \implies is-pure (enum-assn A)
apply (auto simp: is-pure-iff-pure-assn)
apply (rename-tac x x')
apply (case-tac x; case-tac x'; simp add: pure-def)
done

An identitiy rule is required to easily prove trivial refinement theorems

lemma enum-assn-id[simp]: enum-assn id-assn = id-assn
apply (intro ext)
subgoal for x y by (cases x; cases y; simp add: pure-def)

468
Structural rules.

Without congruence condition

**Lemma** `enum-match-nocong`:

\[
\forall x y. \text{hn-ctxt } A x y \Rightarrow \text{hn-ctxt } A' x y
\]

\[
\text{hn-ctxt } (\text{enum-assn } A) e e' \Rightarrow \text{hn-ctxt } (\text{enum-assn } A') e e'
\]

by (cases e; cases e'; simp add: hn-ctxt-def entt-star-mono)

**Lemma** `enum-merge-nocong`:

**Assumes** \[ \forall x y. \text{hn-ctxt } A x y \lor A \text{hn-ctxt } A' x y \Rightarrow A \text{hn-ctxt } A m x y \]

**Shows** \[ \text{hn-ctxt } (\text{enum-assn } A) e e' \lor A \text{hn-ctxt } (\text{enum-assn } A') e e' \Rightarrow A \text{hn-ctxt } (\text{enum-assn } A m) e e' \]

by (cases e; cases e'; simp add: hn-ctxt-def ent-disj-star-mono)

With congruence condition

**Lemma** `enum-match-cong`:

**Assumes** \[ \forall x y. x \in \text{set-enum } e \land y \in \text{set-enum } e' \Rightarrow \text{hn-ctxt } A x y \Rightarrow \text{hn-ctxt } A' x y \]

**Shows** \[ \text{hn-ctxt } (\text{enum-assn } A) e e' \Rightarrow \text{hn-ctxt } (\text{enum-assn } A') e e' \]

by (cases e; cases e'; simp add: hn-ctxt-def entt-star-mono)

**Lemma** `enum-merge-cong`:

**Assumes** \[ \forall x y. x \in \text{set-enum } e \land y \in \text{set-enum } e' \Rightarrow \text{hn-ctxt } A x y \lor A \text{hn-ctxt } A' x y \]

**Shows** \[ \text{hn-ctxt } (\text{enum-assn } A) e e' \lor A \text{hn-ctxt } (\text{enum-assn } A') e e' \Rightarrow \text{hn-ctxt } (\text{enum-assn } A m) e e' \]

apply (blast intro: entt-disjE enum-match-cong entt-disjD1[OF assms] entt-disjD2[OF assms])

Propagating invalid

**Lemma** `entt-invalid-enum`:

**Assumes** \[ \forall x y. \text{hn-invalid } (\text{enum-assn } A) e e' \Rightarrow \text{hn-ctxt } (\text{enum-assn } A) e e' \]

**Shows** \[ \text{hn-ctxt } (\text{enum-assn } A) e e' \Rightarrow (\text{invalid-assn } A) e e' \]

apply (simp add: hn-ctxt-def invalid-assn-def[abs-def])

apply (rule enttI)

apply clarsimp

apply (cases e; cases e'; auto simp: mod-star-conv pure-def)

done

**Lemmas** `invalid-enum-merge`:

**Assumes** `sepref-frame-merge-rules` = `gen-merge-cons`[OF `entt-invalid-enum`]

5.6.2 Constructors

Constructors need to be registered

**Sepref-register** `E1 E2 E3 E4 E5`
Refinement rules can be proven straightforwardly on the separation logic level (method \textit{sepref-to-hoare})

\textbf{Lemma} \texttt{[sepref-fr-rules]}: \( \text{(return o E1,RETURN o E1)} \in A^d \rightarrow_a \text{enum-assn A} \) by \textit{sepref-to-hoare sep-auto} \n
\textbf{Lemma} \texttt{[sepref-fr-rules]}: \( \text{(return o E2,RETURN o E2)} \in A^d \rightarrow_a \text{enum-assn A} \) by \textit{sepref-to-hoare sep-auto} \n
\textbf{Lemma} \texttt{[sepref-fr-rules]}: \( \text{uncurry0 (return E3),uncurry0 (RETURN E3)} \in \text{unit-assn k \rightarrow_a \text{enum-assn A}} \) by \textit{sepref-to-hoare sep-auto} \n
\textbf{Lemma} \texttt{[sepref-fr-rules]}: \( \text{uncurry (return oo E4),uncurry (RETURN oo E4)} \in A^{d*} \rightarrow_a \text{enum-assn A} \) by \textit{sepref-to-hoare sep-auto} \n
\textbf{Lemma} \texttt{[sepref-fr-rules]}: \( \text{uncurry (return oo E5),uncurry (RETURN oo E5)} \in \text{bool-assn k \rightarrow_a \text{enum-assn A}} \) by \textit{sepref-to-hoare (sep-auto simp: pure-def)}

\subsection{5.6.3 Destructor}

There is currently no automation for destructors, so all the registration boilerplate needs to be done manually

Set ups operation identification heuristics 

\textit{sepref-register case-enum} \n
In the monadify phase, this eta-expands to make visible all required arguments

\textbf{Lemma} \texttt{[sepref-monadify-arity]}: \( \text{case-enum \equiv } \lambda_{2f1 f2 f3 f4 f5 x}. SP \text{ case-enum}$(\lambda_{2x}. f1 x)$(\lambda_{2x}. f2 x)$(\lambda_{2x}. f3 x)$(\lambda_{2x}. f4 x)$(\lambda_{2x}. f5 x)$x \) by \textit{simp} \n
This determines an evaluation order for the first-order operands

\textbf{Lemma} \texttt{[sepref-monadify-comb]}: \( \text{EVAL$(\text{case-enum}\$(\lambda_{2x}. f1 x)$\$(\lambda_{2x}. f2 x)$\$(\lambda_{2x}. f3 x)$\$(\lambda_{2x}. f4 x)$\$(\lambda_{2x}. f5 x)$)} \equiv (\Rightarrow) \text{EVAL$x$}$(\lambda_{2x}. SP \text{ case-enum}$(\lambda_{2x}. \text{EVAL } f1 x)$$(\lambda_{2x}. \text{EVAL } f2 x)$$(\lambda_{2x}. \text{EVAL } f3 x)$$(\lambda_{2x}. \text{EVAL } f4 x)$$(\lambda_{2x}. \text{EVAL } f5 x)$) \) by \textit{simp} \n
This enables translation of the case-distinction in a non-monadic context.

\textbf{Lemma} \texttt{[sepref-monadify-comb]}: \( \text{EVAL$(\text{case-enum}$(\lambda_{2x}. f1 x)$\$(\lambda_{2x}. f2 x)$\$(\lambda_{2x}. f3 x)$\$(\lambda_{2x}. f4 x)$\$(\lambda_{2x}. f5 x)$)} \equiv (\Rightarrow) \text{EVAL$x$}$(\lambda_{2x}. SP \text{ case-enum}$(\lambda_{2x}. \text{EVAL } f1 x)$$(\lambda_{2x}. \text{EVAL } f2 x)$$(\lambda_{2x}. \text{EVAL } f3 x)$$(\lambda_{2x}. \text{EVAL } f4 x)$$(\lambda_{2x}. \text{EVAL } f5 x)$) \) apply (rule eq-reflection) \n
by \textit{(simp split: enum.splits)}

Auxiliary lemma, to lift simp-rule over \textit{hn-ctxt}

\textbf{Lemma} \texttt{enum-assn-ctxt}: \( \text{enum-assn A x y = z \implies hn-ctxt (enum-assn A) x y = z} \) by \textit{(simp add: hn-ctxt-def)}
The cases lemma first extracts the refinement for the datatype from the precondition. Next, it generate proof obligations to refine the functions for every case. Finally the postconditions of the refinement are merged. Note that we handle the destructed values separately, to allow reconstruction of the original datatype after the case-expression. Moreover, we provide (invalidated) versions of the original compound value to the cases, which allows access to pure compound values from inside the case.

**lemma** enum-cases-hnr:

fixes $A \ e \ e'$

**defines** [simps]: $INVe \equiv \text{hn-invalid} (\text{enum-assn} \ A) \ e \ e'$

**assumes** $FR: \Gamma \Longrightarrow (\text{hn-cxtxt} (\text{enum-assn} \ A) \ e \ e' \ast F)$

**assumes** $E1: \forall x1 \ x1a. \ [e = E1 \ x1; \ e' = E1 \ x1a] \Longrightarrow \text{hn-refine} (\text{hn-cxtxt} \ A \ x1 \ x1a \ast INVe \ast F) (f1' \ x1a) (\text{hn-cxtxt} \ A1' \ x1 \ x1a \ast \text{hn-cxtxt} \ XX1 \ e \ e' \ast \Gamma1') R (f1 \ x1)

**assumes** $E2: \forall x2 \ x2a. \ [e = E2 \ x2; \ e' = E2 \ x2a] \Longrightarrow \text{hn-refine} (\text{hn-cxtxt} \ A \ x2 \ x2a \ast INVe \ast F) (f2' \ x2a) (\text{hn-cxtxt} \ A2' \ x2 \ x2a \ast \text{hn-cxtxt} \ XX2 \ e \ e' \ast \Gamma2') R (f2 \ x2)

**assumes** $E3: \ [e = E3; \ e' = E3] \Longrightarrow \text{hn-refine} (\text{hn-cxtxt} (\text{enum-assn} \ A) \ e \ e' \ast F) f3' (\text{hn-cxtxt} \ XX3 \ e \ e' \ast \Gamma3') R f3$

**assumes** $E4: \forall x41 \ x41a \ x42a. \ [e = E4 \ x41 \ x41a; \ e' = E4 \ x41a \ x42a] \Longrightarrow \text{hn-refine} (\text{hn-cxtxt} \ A \ x41 \ x41a \ast \text{hn-cxtxt} \ A \ x42a \ast INVe \ast F) (f4' \ x41a \ x42a) (\text{hn-cxtxt} \ A4a' \ x41 \ x41a \ast \text{hn-cxtxt} \ A4b' \ x42a \ast \text{hn-cxtxt} \ XX4 \ e \ e' \ast \Gamma4') R (f4 \ x41 \ x42)

**assumes** $E5: \forall x51 \ x52 \ x51a \ x52a. \ [e = E5 \ x51 \ x52; \ e' = E5 \ x51a \ x52a] \Longrightarrow \text{hn-refine} (\text{hn-cxtxt} \ bool-assn \ x51 \ x51a \ast \text{hn-cxtxt} \ A \ x52 \ x52a \ast INVe \ast F) (f5' \ x51a \ x52a) (\text{hn-cxtxt} \ bool-assn \ x51 \ x51a \ast \text{hn-cxtxt} \ A5' \ x52 \ x52a \ast \text{hn-cxtxt} \ XX5 \ e \ e' \ast \Gamma5') R (f5 \ x51 \ x52)$

**assumes** $MERGE1[unfolded \ hn-cxtxt-def]: \forall x \ x'. \ \text{hn-cxtxt} \ A1' \ x \ x' \lor A \ \text{hn-cxtxt} \ A2' \ x \ x' \lor A \ \text{hn-cxtxt} \ A4a' \ x \ x' \lor A \ \text{hn-cxtxt} \ A4b' \ x \ x' \lor A \ \text{hn-cxtxt} \ A5' \ x \ x' \Longrightarrow (\text{hn-cxtxt} \ A' \ x \ x')$

**assumes** $MERGE2[unfolded \ hn-cxtxt-def]: \Gamma1' \lor A \ \Gamma2' \lor A \ \Gamma3' \lor A \ \Gamma4' \lor A \ \Gamma5' \Longrightarrow (\Gamma')$

**shows** $\text{hn-refine} \ (\text{case-enum} \ f1' \ f2' \ f3' \ f4' \ f5' \ e') (\text{hn-cxtxt} \ (\text{enum-assn} \ A') \ e \ e' \ast \Gamma') R (\text{case-enum} \ $\lambda x2. \ f1 \ x)(\lambda x2. \ f2 \ x)$\$\lambda x y. \ f4 \ x \ y)(\lambda x y. \ f5 \ x y)$\$e)$

**apply** (rule hn-refine-cons-pre[OF FR])

**apply**1 extract-hnr-invalids

**apply** (cases e; cases e'; simp add: enum-assn.simps[THEN enum-assn-cxtxt])

**subgoal**

**apply** (rule hn-refine-cons[OF - E1 - enttl-refl]; assumption?)

**applyS** (simp add: hn-cxtxt-def) — Match precondition for case, get enum-assn from assumption generated by extract-hnr-invalids

**apply** (rule enttl-star-mono) — Split postcondition into pairs for compounds

471
and frame, drop hn-ctxt XX

apply1 (rule entt-fr-drop)
apply1 (rule entt-trans[OF - MERGE1])
applyS (simp add: hn-ctxt-def entt-disjI1' entt-disjI2')
apply1 (rule entt-trans[OF - MERGE2])
applyS (simp add: entt-disjI1' entt-disjI2')
done

subgoal
apply (rule hn-refine-cons[OF - E2 - entt-refl]; assumption?)
applyS (simp add: hn-ctxdef)
apply (rule entt-star-mono)
apply1 (rule entt-fr-drop)
apply1 (rule entt-trans[OF - MERGE1])
applyS (simp add: hn-ctxdef entt-disjI1' entt-disjI2')
apply1 (rule entt-trans[OF - MERGE2])
applyS (simp add: entt-disjI1' entt-disjI2')
done

subgoal
apply (rule hn-refine-cons[OF - E3 - entt-refl]; assumption?)
applyS (simp add: hn-ctxdef)
apply (rule subst mule, rule entt-fr-drop)
apply (rule entt-trans[OF - MERGE2])
apply (simp add: entt-disjI1' entt-disjI2')
done

subgoal
apply (rule hn-refine-cons[OF - E4 - entt-refl]; assumption?)
applyS (simp add: hn-ctxdef)
apply1 (rule entt-star-mono)
apply (rule entt-fr-drop)
apply (rule entt-star-mono)
apply1 (rule entt-trans[OF - MERGE1])
applyS (simp add: hn-ctxdef entt-disjI1' entt-disjI2')
apply1 (rule entt-trans[OF - MERGE1])
applyS (simp add: hn-ctxdef entt-disjI1' entt-disjI2')
apply1 (rule entt-trans[OF - MERGE2])
applyS (simp add: entt-disjI1' entt-disjI2')
done

subgoal
apply (rule hn-refine-cons[OF - E5 - entt-refl]; assumption?)
applyS (simp add: hn-ctxdef)
apply (rule entt-star-mono)
apply1 (rule entt-fr-drop)
apply (rule entt-star-mono)
apply1 (rule ent-imp-entt)
applyS (simp add: hn-ctxdef)
After some more preprocessing (adding extra frame-rules for non-atomic postconditions, and splitting the merge-terms into binary merges), this rule can be registered

\[
\textit{lemmas}\ [\textit{sepref-comb-rules}] = \textit{enum-cases-hnr}[\textit{sepref-prep-comb-rule}]
\]

### 5.6.4 Regression Test

**definition** \(\text{test1} (\varepsilon::\text{bool enum}) \equiv \text{RETURN } \varepsilon\)

**sepref-definition** \(\text{test1-impl} \text{ is } \text{test1} :: (\text{enum-assn bool-assn})^i \rightarrow_a \text{enum-assn bool-assn}\)

**unfolding** \(\text{test1-def}[\text{abs-def}]\) by \textit{sepref}

**sepref-register** \(\text{test1}\)

**lemmas** \([\textit{sepref-fr-rules}] = \textit{test1-impl.refine}\)

**definition** \(\text{test} \equiv \text{do}\{
\)

\[\text{let } x = E1 \text{ True};\]

\[\text{- \leftarrow case } x \text{ of } E1 \ a \Rightarrow \text{RETURN } (\text{Some } a) \quad \text{— Access and invalidate compound inside case}\]

\[| \ 	ext{- \Rightarrow } \text{RETURN } (\text{Some True});\]

\[\text{- \leftarrow } \text{test1 } x; \quad \text{— Rely on structure being there, with valid compound}\]

\[\text{— Same thing again, with merge}\]

\[\text{- \leftarrow if True then } \text{case } x \text{ of } E1 \ a \Rightarrow \text{RETURN } (\text{Some } a) \quad \text{— Access and invalidate compound inside case}\]

\[| \ 	ext{- \Rightarrow } \text{RETURN } (\text{Some True})\]

\[\text{else } \text{RETURN None};\]

\[\text{- \leftarrow test1 } x; \quad \text{— Rely on structure being there, with valid compound}\]

\[\text{— Now test with non-pure}\]

\[\text{let } a = \text{op-array-replicate } 4 \ (3::\text{nat});\]

\[\text{let } x = E5 \text{ False } a;\]

\[\text{- \leftarrow case } x \text{ of } E1 \ - \Rightarrow \text{RETURN } (0::\text{nat})\]

\[| \ E2 \ - \Rightarrow \text{RETURN } 1\]

\[| \ E3 \Rightarrow \text{RETURN } 0\]

\[| \ E4 \ - \Rightarrow \text{RETURN } 0\]

473
| $E5$ - $a \Rightarrow mop\text{-}list\text{-}get\ a\ 0$;

— Rely on that compound still exists (it’s components are only read in the case above)

\[
\text{case } x \text{ of}
\begin{align*}
& E1\ a \Rightarrow \text{do} \{ \text{mop\text{-}list\text{-}set}\ a\ 0\ 0; \text{RETURN } (0\mathcolon\text{nat})\} \\
& E2\ - \Rightarrow \text{RETURN } 1 \\
& E3\ a \Rightarrow \text{RETURN } 0 \\
& E4\ - \Rightarrow \text{RETURN } 0 \\
& E5\ - \Rightarrow \text{RETURN } 0
\end{align*}
\]

lemmas $[\text{safe-constraint-rules}] = \text{CN\text{-}FALSEI}[@\text{is\text{-}pure invalid-assn } A \text{ for } A]$
fixes \( B :: 'b \Rightarrow 'b'i \Rightarrow \text{assn} \)

fixes \( F :: \text{assn} \) — Symbolic frame, representing all heap content the map-function body may access

notes \([\text{sepref-register-adhoc \( f \ l \)}]\) — Register for operation id

assumes \( f-rl: \text{hn-refine} (\text{hn-ctxt} A \ x \ xi * F) (f'i \ xi) (\text{hn-ctxt} A' \ x \ xi * F) \ B (f'\$x) \)
— Refinement for \( f \)

begin

We implement our combinator using the monadic refinement framework.

definition \( \text{mmap} \equiv \text{RECT} (\lambda \text{mmap}. \lambda [] \Rightarrow \text{RETURN} []) \)

| \( x \#xs \Rightarrow \text{do} \{ x \leftarrow f'x; \ xs \leftarrow \text{mmap} \ xs; \text{RETURN} (x\#xs) \} \) l

5.7.2 Synthesis of Implementation

In order to propagate the frame \( F \) during synthesis, we use a trick: We wrap the frame into a dummy refinement assertion. This way, sepref recognizes the frame just as another context element, and does correct propagation.

definition \( \text{F-assn} (x::\text{unit}) (y::\text{unit}) \equiv F \)

lemma \( \text{F-unf}: \text{hn-ctxt} \ F\text{-assn} x y = F \)
by (auto simp: \( \text{F-assn-def} \) \( \text{hn-ctxt-def} \))

We build a combinator rule to refine \( f \). We need a combinator rule here, because \( f \) does not only depend on its formal arguments, but also on the frame (represented as dummy argument).

lemma \( f-rl':: \text{hn-refine} (\text{hn-ctxt} A \ x \ xi * \text{hn-ctxt} (\text{F-assn}) \ dx \ dxi) (f'i \ xi) (\text{hn-ctxt} A' \ x \ xi * \text{hn-ctxt} (\text{F-assn}) \ dx \ dxi) \ B (f'\$x) \)
unfolding \( \text{F-unf} \) by (rule \( f-rl' \))

Then we use the Sepref tool to synthesize an implementation of \( \text{mmap} \).

schematic-goal \( \text{mmap-impl}: \)

notes \([\text{sepref-comb-rules}] = \text{hn-refine-frame}[\text{OF} f-rl']\)

shows \( \text{hn-refine} (\text{hn-ctxt} (\text{list-assn} A) \ l \ li * \text{hn-ctxt} (\text{F-assn}) \ dx \ dxi) (\ ?c::'?c) \text{Heap} \)
?\Gamma' \ ?R \text{mmap}

unfolding \( \text{mmap-def} \) \( \text{HOL-list.fold-custom-empty} \)
apply \( \text{sepref-dbq-keep} \)

done

We unfold the wrapped frame

lemmas \( \text{mmap-impl'} = \text{mmap-impl}[\text{unfolded} \ F\text{-unf}] \)

end
5.7.3 Setup for Sepref

Outside the context, we extract the synthesized implementation as a new constant, and set up code theorems for the fixed-point combinators.

```plaintext
concrete-definition mmap-impl uses mmap-impl'
prepare-code-thms mmap-impl-def
```

Moreover, we have to manually declare arity and monadify theorems. The arity theorem ensures that we always have a constant number of operators, and the monadify theorem determines an execution order: The list-argument is evaluated first.

```plaintext
lemma mmap-arity[sepref-monadify-arity]: mmap ≡ λ2 l. SP mmap$(λ2x. f$x)$l
by simp
```

We can massage the refinement theorem \((\bigwedge x i. \text{hn-refine} (\text{hn-ctxt ?A x i} * ?F) (\text{fi x i}) (\text{hn-ctxt ?A' x i} * ?F) ?B (\text{?f x})) \implies \text{hn-refine} (\text{hn-ctxt (list-assn ?A) ?l ?li} * ?F) (\text{mmap-impl ?fi ?li}) (\text{hn-ctxt (list-assn ?A') ?l ?li} * ?F) (\text{list-assn ?B}) (\text{mmap ?f ?l})\) a bit, to get a valid combinator rule

```plaintext
print-statement
```

```plaintext
lemma mmap-comb-rl[sepref-comb-rules]:
assumes P \implies, \text{hn-ctxt (list-assn A) l li} * F
— Initial frame
and \(\bigwedge x i. \text{hn-refine} (\text{hn-ctxt A x i} * F) (\text{fi x i}) (\text{Q x xi}) B (f x)
— Refinement of map-function
and \(\bigwedge x i. Q x xi \implies, \text{hn-ctxt A' x xi} * F
— Recover refinement for list-element and original frame from what map-function produced
shows \(\text{hn-refine P (mmap-impl fi li) (hn-ctxt (list-assn A') l li F) (list-assn B) (mmap$(\lambda2x. f x)$l)}\)
unfolding APP-def PROTECT2-def
using \(\text{hn-refine-cons-pre[OF - mmap-impl.refine, sepref-prep-comb-rule, no-vars]}\)
```

Using `assms` by `simp`

5.7.4 Example

Finally, we can test our combinator. Note how the map-function accesses the array on the heap, which is not among its arguments. This is only possible as we passed around a frame.

```plaintext
sepref-thm test-mmap
is λ. do { let a = op-array-of-list [True, True, False]; mmap (λx. do { mop-list-get a (x mod 3) }) l }
```
:: (list-assn nat-assn)^k \rightarrow_a list-assn bool-assn

unfolding HOL-list.fold-custom-empty
by sepref

5.7.5 Limitations

Currently, the major limitation is that combinator rules are fixed to specific data types. In our example, we did an implementation for HOL lists. We cannot come up with an alternative implementation, for, e.g., array-lists, but have to use a different abstract combinator.

One workaround is to use some generic operations, as is done for foreach-loops, which require a generic to-list operation. However, in this case, we produce unwanted intermediate lists, and would have to add complicated a-posteriori deforestation optimizations.

end
Chapter 6

Benchmarks

Contains the benchmarks of the IRF/IICF. See the README file in the benchmark folder for more information on how to run the benchmarks.

theory Heapmap-Bench
imports
  ../../../IICF/Impl/Heaps/IICF-Impl-Heapmap
  ../../../Sepref-ICF-Bindings
begin

definition rrand :: uint32 ⇒ uint32
  where rrand s ≡ (s * 1103515245 + 12345) AND 0x7FFFFFFF

definition rand :: uint32 ⇒ nat ⇒ (uint32 * nat)
  where
    rand s m ≡ let
      s = rrand s;
      r = nat-of-uint32 s;
      r = (r * m) div 0x80000000
    in (s,r)

partial-function (heap) rep where rep i N f s = {
  if i<N then do {
    s ← f s i;
    rep (i+1) N f s
  } else return s
}

declare rep.simps[code]

term hm-insert-op-impl

definition testsuite N ≡ do {
  let s=0;
  let N2=efficient-nat-div2 N;
  hm ← hm-empty-op-impl N;
}
(hm,s) ← rep 0 N (λ(hm,s) i. do {
  let (s,v) = rand s N2;
  hm ← hm-insert-op-impl N id i v hm;
  return (hm,s)
}) (hm,s);

(hm,s) ← rep 0 N (λ(hm,s) i. do {
  let (s,v) = rand s N2;
  hm ← hm-change-key-op-impl id i v hm;
  return (hm,s)
}) (hm,s);

hm ← rep 0 N (λhm i. do {
  (_,hm) ← hm-pop-min-op-impl id hm;
  return hm
}) hm;

return ()
}

export-code rep in SML-imp

partial-function (tailrec) drep where drep i N f s = ( if i<N then drep (i+1) N f (f s i) else s )

declare drep.simps[code]

term aluprioi.insert
term aluprioi.empty
term aluprioi.pop
definition ftestsuite N ≡ do {
  let s=0;
  let N2=efficient-nat-div2 N;
  let hm= aluprioi.empty ();

  let (hm,s) = drep 0 N (λ(hm,s) i. do {
    let (s,v) = rand s N2;
    let hm = aluprioi.insert hm i v;
    (hm,s)
  }) (hm,s);

  let (hm,s) = drep 0 N (λ(hm,s) i. do {

let (s,v) = rand s N2;
let hm = aluprioi.insert hm i v;
(hm,s)
}} (hm,s);

let hm = drep 0 N (λhm i. do {
  let (-,-,hm) = aluprioi.pop hm;
  hm
}) hm;

()}

export-code
testsuite ftestsuite
nat-of-integer integer-of-nat
in SML-imp module-name Heapmap
file heapmap-export.sml

end
theory Dijkstra-Benchmark
imports ../../../Examples/Sepref-Dijkstra
  Dijkstra-Shortest-Path.Test
begin

definition nat-cr-graph-imp :: nat ⇒ (nat × nat × nat) list ⇒ nat graph-impl Heap
  where nat-cr-graph-imp ≡ cr-graph

concrete-definition nat-dijkstra-imp uses dijkstra-imp-def[where 'W=nat]
prepare-code-thms nat-dijkstra-imp-def

lemma nat-dijkstra-imp-eq: nat-dijkstra-imp = dijkstra-imp
unfolding dijkstra-imp-def[abs-def] nat-dijkstra-imp-def[abs-def]
by simp

definition nat-cr-graph-fun nn es ≡ hlg-from-list-nat ([0..<nn], es)

export-code
  integer-of-nat nat-of-integer

  ran-graph

  nat-cr-graph-fun nat-dijkstra
locale bm-fun begin

schematic-goal succ-of-list-impl:
  notes [autoref-tyrel] =
  ty-REL[where 'a=nat→nat set and R=⟨nat-rel,R⟩dflt-rm-rel for R]
  ty-REL[where 'a=nat set and R=⟨nat-rel⟩list-set-rel]

  shows (?f::'?c,succ-of-list) ∈ ?R
  unfolding succ-of-list-def[abs-def]
  apply (autoref (keep-goal))
  done

concrete-definition succ-of-list-impl uses succ-of-list-impl

schematic-goal acc-of-list-impl:
  notes [autoref-tyrel] =
  ty-REL[where 'a=nat set and R=⟨nat-rel⟩dflt-rs-rel for R]

  shows (?f::'?c,acc-of-list) ∈ ?R
  unfolding acc-of-list-def[abs-def]
  apply (autoref (keep-goal))
  done

concrete-definition acc-of-list-impl uses acc-of-list-impl

schematic-goal red-dfs-impl-refine-aux:

  fixes u::'nat and V::'nat set
  notes [autoref-tyrel] =
  ty-REL[where 'a=nat set and R=⟨nat-rel⟩dflt-rs-rel]
  assumes [autoref-rules]:
  (u,u)∈nat-rel
  (V,V')∈⟨nat-rel⟩dflt-rs-rel
\[(onstack, onstack') \in \langle \text{nat-rel} \rangle \text{dflt-rs-rel} \]
\[(E, E') \in \langle \text{nat-rel} \rangle \text{slg-rel} \]

shows \( \text{RETURN} \ (\ ?f \ :: \ ?'c), \ \text{red-dfs} \ E' \ onstack' \ V' \ u') \in \ ?R \)
apply –
unfolding red-dfs-def
apply (autoref-monadic)
done

concrete-definition red-dfs-impl uses red-dfs-impl-refine-aux
prepare-code-thms red-dfs-impl-def
declare red-difs-impl.refine[autoref-higher-order-rule, autoref-rules]

schematic-goal ndfs-impl-refine-aux:
fixes \( s :: \text{nat} \) and \( \text{succi} \)
notes [autoref-tyrel] =
ty-REL[where 'a=\text{nat set} \) \text{and} \ R=\langle \text{nat-rel} \rangle \text{dflt-rs-rel}]
assumes [autoref-rules]:
\((\text{succi}, E)\in \langle \text{nat-rel} \rangle \text{slg-rel} \)
\((A_i, A)\in \langle \text{nat-rel} \rangle \text{dflt-rs-rel} \)
notes [autoref-rules] = IdI[of s]
shows \( \text{RETURN} \ (\ ?f \ :: \ ?'c), \ \text{blue-dfs} \ E \ A \ s) \in \ (\ ?R) \text{nres-rel} \)
unfolding blue-dfs-def
apply (autoref-monadic (trace))
done

concrete-definition fun-ndfs-impl for \( \text{succi} \ A_i \ s \) uses ndfs-impl-refine-aux
prepare-code-thms fun-ndfs-impl-def

definition fun-succ-of-list \( \equiv \)
\text{succ-of-list-impl o map} \ (\lambda (u, v). \ (\text{nat-of-integer} u, \ \text{nat-of-integer} v))

definition fun-acc-of-list \( \equiv \)
\text{acc-of-list-impl o map} \ \text{nat-of-integer}

end

interpretation fun: \( \text{bm-fun} \).

locale bm-funs begin

schematic-goal succ-of-list-impl:
notes [autoref-tyrel] =
ty-REL[where 'a=\text{nat} \rightarrow \text{nat set} \) \text{and} \ R=(\text{nat-rel}, R) \text{iam-map-rel for} \ R]
ty-REL[where 'a=\text{nat set} \) \text{and} \ R=\langle \text{nat-rel} \rangle \text{list-set-rel}]

shows \( (\ ?f \ :: \ ?'c, \text{succ-of-list}) \in \ ?R \)
unfolding succ-of-list-def[abs-def]
apply (autoref (keep-goal))
done

concrete-definition succ-of-list-impl uses succ-of-list-impl

schematic-goal acc-of-list-impl:
  notes [autoref-tyrel] =
  ty-REL[where 'a=nat set and R=(nat-rel)iam-set-rel for R]

shows (?f::'?c,acc-of-list) ∈ ?R
unfolding acc-of-list-def[abs-def]
apply (autoref (keep-goal))
done

concrete-definition acc-of-list-impl uses acc-of-list-impl

schematic-goal red-dfs-impl-refine-aux:

  fixes u':nat and V':nat set
  notes [autoref-tyrel] =
  ty-REL[where 'a=nat set and R=(nat-rel)iam-set-rel]
  assumes [autoref-rules]:
  (u,u')∈nat-rel
  (V,V')∈(nat-rel)iam-set-rel
  (onstack,onstack')∈(nat-rel)iam-set-rel
  (E,E')∈(nat-rel)slg-rel
  shows (RETURN (?f::'?c), red-dfs E' onstack' V' u') ∈ ?R
  apply –
  unfolding red-dfs-def
  apply (autoref-monadic)
done

concrete-definition red-dfs-impl uses red-dfs-impl-refine-aux
prepare-code-thms red-dfs-impl-def
declare red-dfs-impl.refine[autoref-higher-order-rule, autoref-rules]

schematic-goal ndfs-impl-refine-aux:

  fixes s::nat and succi
  notes [autoref-tyrel] =
  ty-REL[where 'a=nat set and R=(nat-rel)iam-set-rel]
  assumes [autoref-rules]:
  (succi,E)∈(nat-rel)slg-rel
  (Ai,A)∈(nat-rel)iam-set-rel
  notes [autoref-rules] = Idf[of s]
  shows (RETURN (?f::'?c), blue-dfs E A s) ∈ (?R)nres-rel
  unfolding blue-dfs-def
  apply (autoref-monadic (trace))
done

483
concrete-definition funs-ndfs-impl for succi Ai s uses ndfs-impl-refine-aux
prepare-code-thms funs-ndfs-impl-def

definition funs-succ-of-list ≡
  succ-of-list-impl o map (\(u,v\). (nat-of-integer u, nat-of-integer v))

definition funs-acc-of-list ≡
  acc-of-list-impl o map nat-of-integer
end

interpretation funs: bm-funs .

definition imp-ndfs-impl ≡ blue-dfs-impl
definition imp-ndfs-sz-impl ≡ blue-dfs-impl-sz
definition imp-acc-of-list l ≡ From-List-GA.ias-from-list (map nat-of-integer l)
definition imp-graph-of-list n l ≡ cr-graph (nat-of-integer n) (map (pairself nat-of-integer) l)

export-code
  nat-of-integer integer-of-nat
  fun.fun-ndfs-impl fun.fun-succ-of-list fun.fun-acc-of-list
  funs.funs-ndfs-impl funs.funs-succ-of-list funs.funs-acc-of-list
  imp-ndfs-impl imp-ndfs-sz-impl imp-acc-of-list imp-graph-of-list
in SML-imp module-name NDFS-Benchmark file NDFS-Benchmark-export.sml

ML-val ⟨open Time⟩
end

484