

# The Imperative Refinement Framework

Peter Lammich

March 17, 2025

## 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 (ICF), 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++.

[ITP-2015] Peter Lammich: Refinement to Imperative/HOL. ITP 2015: 253–269

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

# Contents

<b>1</b>	<b>The Sepref Tool</b>	<b>2</b>
1.1	Operation Identification Phase . . . . .	2
1.1.1	Proper Protection of Term . . . . .	2
1.1.2	Operation Identification . . . . .	3
1.1.3	ML-Level code . . . . .	4
1.1.4	Default Setup . . . . .	4
1.2	Basic Definitions . . . . .	5
1.2.1	Values on Heap . . . . .	5
1.2.2	Constraints in Refinement Relations . . . . .	7
1.2.3	Heap-Nres Refinement Calculus . . . . .	7
1.2.4	Product Types . . . . .	9
1.2.5	Convenience Lemmas . . . . .	10
1.2.6	ML-Level Utilities . . . . .	13
1.3	Monadify . . . . .	13
1.4	Frame Inference . . . . .	16
1.5	Refinement Rule Management . . . . .	20
1.5.1	Assertion Interface Binding . . . . .	20
1.5.2	Function Refinement with Precondition . . . . .	21
1.5.3	Heap-Function Refinement . . . . .	22
1.5.4	Automation . . . . .	31
1.6	Setup for Combinators . . . . .	34
1.6.1	Interface Types . . . . .	34
1.6.2	Rewriting Inferred Interface Types . . . . .	34
1.6.3	ML-Code . . . . .	35
1.6.4	Obsolete Manual Setup Rules . . . . .	35
1.7	Translation . . . . .	35
1.7.1	Import of Parametricity Theorems . . . . .	43
1.7.2	Purity . . . . .	44
1.8	Sepref-Definition Command . . . . .	45
1.8.1	Setup of Extraction-Tools . . . . .	45
1.8.2	Synthesis setup for sepref-definition goals . . . . .	45
1.9	Utilities for Interface Specifications and Implementations . . . . .	46
1.9.1	Relation Interface Binding . . . . .	46

1.9.2	Operations with Precondition . . . . .	47
1.9.3	Constraints . . . . .	48
1.9.4	Composition . . . . .	48
1.9.5	Protected Constants . . . . .	51
1.9.6	Rule Collections . . . . .	51
1.9.7	ML-Level Declarations . . . . .	51
1.9.8	Obsolete Manual Specification Helpers . . . . .	52
1.10	Sepref Tool . . . . .	54
1.10.1	Sepref Method . . . . .	54
1.10.2	Debugging Methods . . . . .	55
1.10.3	Utilities . . . . .	55
<b>2</b>	<b>Basic Setup</b>	<b>58</b>
2.1	HOL Setup . . . . .	58
2.1.1	Assertion Annotation . . . . .	58
2.1.2	Shortcuts . . . . .	58
2.1.3	Identity Relations . . . . .	59
2.1.4	Inverse Relation . . . . .	59
2.1.5	Single Valued and Total Relations . . . . .	60
2.1.6	Bounded Assertions . . . . .	62
2.1.7	Tool Setup . . . . .	66
2.1.8	HOL Combinators . . . . .	66
2.1.9	Basic HOL types . . . . .	67
2.1.10	Product . . . . .	67
2.1.11	Option . . . . .	70
2.1.12	Lists . . . . .	73
2.1.13	Sum-Type . . . . .	76
2.1.14	String Literals . . . . .	79
2.2	Setup for Foreach Combinator . . . . .	80
2.2.1	Foreach Loops . . . . .	80
2.2.2	For Loops . . . . .	92
2.3	Ad-Hoc Solutions . . . . .	94
2.3.1	Pure Higher-Order Functions . . . . .	94
<b>3</b>	<b>The Imperative Isabelle Collection Framework</b>	<b>96</b>
3.1	Set Interface . . . . .	96
3.1.1	Operations . . . . .	96
3.1.2	Patterns . . . . .	97
3.2	Sets by Lists that Own their Elements . . . . .	98
3.3	Multiset Interface . . . . .	100
3.3.1	Additions to Multiset Theory . . . . .	100
3.3.2	Parametricity Setup . . . . .	101
3.3.3	Operations . . . . .	102
3.3.4	Patterns . . . . .	102

3.4	Priority Bag Interface . . . . .	103
3.4.1	Operations . . . . .	103
3.4.2	Patterns . . . . .	104
3.5	Multisets by Lists . . . . .	104
3.5.1	Abstract Operations . . . . .	105
3.5.2	Declaration of Implementations . . . . .	106
3.5.3	Swap two elements of a list, by index . . . . .	108
3.5.4	Operations . . . . .	109
3.5.5	Patterns . . . . .	110
3.6	Heap Implementation On Lists . . . . .	111
3.6.1	Basic Definitions . . . . .	112
3.6.2	Basic Operations . . . . .	114
3.6.3	Auxiliary operations . . . . .	117
3.6.4	Operations . . . . .	122
3.6.5	Operations as Relator-Style Refinement . . . . .	124
3.7	Implementation of Heaps with Arrays . . . . .	132
3.7.1	Setup of the Sepref-Tool . . . . .	133
3.7.2	Synthesis of operations . . . . .	134
3.7.3	Regression Test . . . . .	135
3.8	Map Interface . . . . .	136
3.8.1	Parametricity for Maps . . . . .	136
3.8.2	Interface Type . . . . .	137
3.8.3	Operations . . . . .	137
3.8.4	Patterns . . . . .	137
3.8.5	Parametricity . . . . .	138
3.9	Priority Maps . . . . .	138
3.9.1	Additional Operations . . . . .	139
3.10	Priority Maps implemented with List and Map . . . . .	140
3.10.1	Basic Setup . . . . .	140
3.10.2	Basic Operations . . . . .	142
3.10.3	Auxiliary Operations . . . . .	146
3.10.4	Operations . . . . .	149
3.11	Plain Arrays Implementing List Interface . . . . .	158
3.11.1	Empty . . . . .	165
3.11.2	Swap . . . . .	166
3.11.3	Length . . . . .	167
3.11.4	Index . . . . .	167
3.11.5	Butlast . . . . .	167
3.11.6	Append . . . . .	168
3.11.7	Get . . . . .	169
3.11.8	Contains . . . . .	169
3.12	Implementation of Heaps by Arrays . . . . .	170
3.12.1	Implement Basic Operations . . . . .	174
3.12.2	Auxiliary Operations . . . . .	176

3.12.3	Interface Operations . . . . .	177
3.12.4	Manual fine-tuning of code-lemmas . . . . .	180
3.13	Matrices . . . . .	183
3.13.1	Relator and Interface . . . . .	183
3.13.2	Operations . . . . .	183
3.13.3	Patterns . . . . .	184
3.13.4	Pointwise Operations . . . . .	184
3.14	Matrices by Array (Row-Major) . . . . .	190
3.14.1	Pointwise Operations . . . . .	195
3.14.2	Regression Test and Usage Example . . . . .	198
3.15	Sepref Bindings for Imp/HOL Collections . . . . .	200
3.15.1	Binding Locales . . . . .	203
3.15.2	Array Map (iam) . . . . .	211
3.15.3	Array Set (ias) . . . . .	211
3.15.4	Hash Map (hm) . . . . .	212
3.15.5	Hash Set (hs) . . . . .	213
3.15.6	Open Singly Linked List (osll) . . . . .	214
3.15.7	Circular Singly Linked List (csll) . . . . .	214
3.16	The Imperative Isabelle Collection Framework . . . . .	215
<b>4</b>	<b>User Guides</b>	<b>217</b>
4.1	Quickstart Guide . . . . .	217
4.1.1	Introduction . . . . .	217
4.1.2	First Example . . . . .	218
4.1.3	Binary Search Example . . . . .	222
4.1.4	Basic Troubleshooting . . . . .	223
4.1.5	The Isabelle Imperative Collection Framework (IICF) . . . . .	225
4.1.6	Specification of Preconditions . . . . .	226
4.1.7	Linearity and Copying . . . . .	227
4.1.8	Nesting of Data Structures . . . . .	228
4.1.9	Fixed-Size Data Structures . . . . .	229
4.1.10	Type Classes . . . . .	238
4.1.11	Higher-Order . . . . .	238
4.1.12	A-Posteriori Optimizations . . . . .	238
4.1.13	Short-Circuit Evaluation . . . . .	238
4.2	Reference Guide . . . . .	238
4.2.1	The Sepref Method . . . . .	239
4.2.2	Refinement Rules . . . . .	252
4.2.3	Composition . . . . .	255
4.2.4	Registration of Interface Types . . . . .	256
4.2.5	Registration of Abstract Operations . . . . .	257
4.2.6	High-Level tools for Interface/Implementation Declaration . . . . .	258
4.2.7	Defining synthesized Constants . . . . .	260

4.3	General Purpose Utilities . . . . .	261
4.3.1	Methods . . . . .	261
4.3.2	Structured Apply Scripts (experimental) . . . . .	262
4.3.3	Extracting Definitions from Theorems . . . . .	262
<b>5</b>	<b>Examples</b>	<b>263</b>
5.1	Imperative Graph Representation . . . . .	263
5.2	Simple DFS Algorithm . . . . .	265
5.2.1	Definition . . . . .	265
5.2.2	Refinement to Imperative/HOL . . . . .	266
5.3	Imperative Implementation of Dijkstra’s Shortest Paths Algorithm . . . . .	266
5.4	Imperative Implementation of of Nested DFS (HPY-Improvement)	271
5.5	Generic Worklist Algorithm with Subsumption . . . . .	274
5.5.1	Utilities . . . . .	274
5.5.2	Search Spaces . . . . .	275
5.5.3	Worklist Algorithm . . . . .	276
5.5.4	Towards an Implementation . . . . .	279
5.6	Non-Recursive Algebraic Datatype . . . . .	281
5.6.1	Refinement Assertion . . . . .	281
5.6.2	Constructors . . . . .	282
5.6.3	Destructor . . . . .	283
5.6.4	Regression Test . . . . .	286
5.7	Snippet to Define Custom Combinators . . . . .	287
5.7.1	Definition of the Combinator . . . . .	287
5.7.2	Synthesis of Implementation . . . . .	288
5.7.3	Setup for Sepref . . . . .	288
5.7.4	Example . . . . .	289
5.7.5	Limitations . . . . .	289
<b>6</b>	<b>Benchmarks</b>	<b>290</b>

# 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  $\lambda$ -. *None* for the empty map, or *fun-upd m k (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 by the constants defined below.



**definition**  $[simp]$ :  $PROTECT2\ x\ (y::prop) \equiv x$   
**consts**  $DUMMY :: prop$

**abbreviation**  $PROTECT2\text{-syn}\ (\langle\#\#\rangle)$  **where**  $PROTECT2\text{-syn}\ t \equiv PROTECT2\ t\ DUMMY$

**abbreviation**  $(input)ABS2 :: ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$  (**binder**  $\langle\lambda_2\rangle\ 10$ )  
**where**  $ABS2\ f \equiv (\lambda x. PROTECT2\ (f\ x)\ DUMMY)$

**lemma**  $beta$ :  $(\lambda_2 x. f\ x)\ \$x \equiv f\ x$   $\langle proof \rangle$

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

**definition**  $APP'$  (**infixl**  $\langle\$\rangle\ 900$ ) **where**  $[simp, autoref-tag-defs]$ :  $f\ \$a \equiv f\ a$

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  $PR-CONST$  tag indicates terms that our tool shall handle as atomic constants, and never look into them.

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

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

## 1.1.2 Operation Identification

Indicator predicate for conceptual typing of a constant

**definition**  $intf\text{-type} :: 'a \Rightarrow 'b\ itself \Rightarrow bool$  (**infix**  $\langle::_i\rangle\ 10$ ) **where**  
 $[simp]$ :  $c::_i I \equiv True$

**lemma**  $itypeI$ :  $c::_i I$   $\langle proof \rangle$

**lemma**  $itypeI'$ :  $intf\text{-type}\ c\ TYPE('T)$   $\langle proof \rangle$

**lemma**  $itype\text{-self}$ :  $(c::'a) ::_i TYPE('a)$   $\langle proof \rangle$

**definition**  $CTYPE\text{-ANNOT} :: 'b \Rightarrow 'a\ itself \Rightarrow 'b$  (**infix**  $\langle:::{}_i\rangle\ 10$ ) **where**  
 $[simp]$ :  $c:::{}_i I \equiv c$

Wrapper predicate for an conceptual type inference

**definition**  $ID :: 'a \Rightarrow 'a \Rightarrow 'c\ itself \Rightarrow bool$   
**where**  $[simp]$ :  $ID\ t\ t'\ T \equiv t=t'$

## Conceptual Typing Rules

**lemma**  $ID\text{-unfold-vars}$ :  $ID\ x\ y\ T \Longrightarrow x \equiv y$   $\langle proof \rangle$

**lemma** *ID-PR-CONST-trigger*:  $ID (PR-CONST x) y T \Longrightarrow ID (PR-CONST x) y T \langle proof \rangle$

**lemma** *pat-rule*:  
 $\llbracket p \equiv p'; ID p' t' T \rrbracket \Longrightarrow ID p t' T \langle proof \rangle$

**lemma** *app-rule*:  
 $\llbracket ID f f' TYPE('a \Rightarrow 'b); ID x x' TYPE('a) \rrbracket \Longrightarrow ID (f\$x) (f'\$x') TYPE('b) \langle proof \rangle$

**lemma** *app'-rule*:  
 $\llbracket ID f f' TYPE('a \Rightarrow 'b); ID x x' TYPE('a) \rrbracket \Longrightarrow ID (f'\$x) (f'\$x') TYPE('b) \langle proof \rangle$

**lemma** *abs-rule*:  
 $\llbracket \bigwedge x x'. ID x x' TYPE('a) \Longrightarrow ID (t x) (t' x x') TYPE('b) \rrbracket \Longrightarrow ID (\lambda_2 x. t x) (\lambda_2 x'. t' x' x') TYPE('a \Rightarrow 'b) \langle proof \rangle$

**lemma** *id-rule*:  $c ::_i I \Longrightarrow ID c c I \langle proof \rangle$

**lemma** *annot-rule*:  $ID t t' I \Longrightarrow ID (t ::_i I) t' I \langle proof \rangle$

**lemma** *fallback-rule*:  
 $ID (c :: 'a) c TYPE('c) \langle proof \rangle$

**lemma** *unprotect-rl1*:  $ID (PR-CONST x) t T \Longrightarrow ID (UNPROTECT x) t T \langle proof \rangle$

### 1.1.3 ML-Level code

$\langle ML \rangle$

**named-theorems-rev** *id-rules* Operation identification rules

**named-theorems-rev** *pat-rules* Operation pattern rules

**named-theorems-rev** *def-pat-rules* Definite operation pattern rules (not backtracked over)

$\langle ML \rangle$

### 1.1.4 Default Setup

#### Numerals

**lemma** *pat-numeral* $[def-pat-rules]$ :  $numeral\$x \equiv UNPROTECT (numeral\$x) \langle proof \rangle$

**lemma** *id-nat-const*[*id-rules*]: (*PR-CONST* (*a::nat*)) ::<sub>i</sub> *TYPE*(*nat*) ⟨*proof*⟩

**lemma** *id-int-const*[*id-rules*]: (*PR-CONST* (*a::int*)) ::<sub>i</sub> *TYPE*(*int*) ⟨*proof*⟩

**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** <:::;<sub>i</sub>> 10)

**no-notation** *CONST-INTF* (**infixr** <:::;<sub>i</sub>> 10)

In this theory, we define the basic concept of refinement from a non-deterministic 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* ≡ (λ *a c*. ↑((*c,a*) ∈ *R*))

**lemma** *pure-app-eq*: *pure* *R a c* = ↑((*c,a*) ∈ *R*) ⟨*proof*⟩

**lemma** *pure-eq-conv*[*simp*]: *pure* *R* = *pure* *R'* ↔ *R* = *R'*  
⟨*proof*⟩

**lemma** *pure-rel-eq-false-iff*: *pure* *R x y* = *false* ↔ (*y,x*) ∉ *R*  
⟨*proof*⟩

**definition** *is-pure*  $P \equiv \exists P'. \forall x x'. P x x' = \uparrow(P' x x')$

**lemma** *is-pureI*[*intro?*]:

**assumes**  $\bigwedge x x'. P x x' = \uparrow(P' x x')$

**shows** *is-pure*  $P$

*<proof>*

**lemma** *is-pureE*:

**assumes** *is-pure*  $P$

**obtains**  $P'$  **where**  $\bigwedge x x'. P x x' = \uparrow(P' x x')$

*<proof>*

**lemma** *pure-pure*[*simp*]: *is-pure* (*pure*  $P$ )

*<proof>*

**lemma** *pure-hn-ctxt*[*intro!*]: *is-pure*  $P \implies$  *is-pure* (*hn-ctxt*  $P$ )

*<proof>*

**definition** *the-pure*  $P \equiv \text{THE } P'. \forall x x'. P x x' = \uparrow((x',x) \in P')$

**lemma** *the-pure-pure*[*simp*]: *the-pure* (*pure*  $R$ ) =  $R$

*<proof>*

**lemma** *is-pure-alt-def*: *is-pure*  $R \longleftrightarrow (\exists Ri. \forall x y. R x y = \uparrow((y,x) \in Ri))$

*<proof>*

**lemma** *pure-the-pure*[*simp*]: *is-pure*  $R \implies$  *pure* (*the-pure*  $R$ ) =  $R$

*<proof>*

**lemma** *is-pure-conv*: *is-pure*  $R \longleftrightarrow (\exists R'. R = \text{pure } R')$

*<proof>*

**lemma** *is-pure-the-pure-id-eq*[*simp*]: *is-pure*  $R \implies$  *the-pure*  $R = \text{Id} \longleftrightarrow R = \text{pure } \text{Id}$

*<proof>*

**lemma** *is-pure-iff-pure-assn*: *is-pure*  $P = (\forall x x'. \text{is-pure-assn } (P x x'))$

*<proof>*

**abbreviation** *hn-val*  $R \equiv \text{hn-ctxt } (\text{pure } R)$

**lemma** *hn-val-unfold*: *hn-val*  $R a b = \uparrow((b,a) \in R)$

*<proof>*

**definition** *invalid-assn*  $R x y \equiv \uparrow(\exists h. h \models R x y) * \text{true}$

**abbreviation** *hn-invalid*  $R \equiv \text{hn-ctxt } (\text{invalid-assn } R)$

**lemma** *invalidate-clone*:  $R \ x \ y \Longrightarrow_A \text{invalid-assn } R \ x \ y * R \ x \ y$   
 ⟨*proof*⟩

**lemma** *invalidate-clone'*:  $R \ x \ y \Longrightarrow_A \text{invalid-assn } R \ x \ y * R \ x \ y * \text{true}$   
 ⟨*proof*⟩

**lemma** *invalidate*:  $R \ x \ y \Longrightarrow_A \text{invalid-assn } R \ x \ y$   
 ⟨*proof*⟩

**lemma** *invalid-pure-recover*:  $\text{invalid-assn } (\text{pure } R) \ x \ y = \text{pure } R \ x \ y * \text{true}$   
 ⟨*proof*⟩

**lemma** *hn-invalidI*:  $h \models_{\text{hn-ctxt}} P \ x \ y \Longrightarrow \text{hn-invalid } P \ x \ y = \text{true}$   
 ⟨*proof*⟩

**lemma** *invalid-assn-cong*[*cong*]:  
 assumes  $x \equiv x'$   
 assumes  $y \equiv y'$   
 assumes  $R \ x' \ y' \equiv R' \ x' \ y'$   
 shows  $\text{invalid-assn } R \ x \ y = \text{invalid-assn } R' \ x' \ y'$   
 ⟨*proof*⟩

### 1.2.2 Constraints in Refinement Relations

**lemma** *mod-pure-conv*[*simp*]:  $(h, as) \models_{\text{pure}} R \ a \ b \longleftrightarrow (as = \{\}) \wedge (b, a) \in R$   
 ⟨*proof*⟩

**definition** *rdomp* ::  $('a \Rightarrow 'c \Rightarrow \text{assn}) \Rightarrow 'a \Rightarrow \text{bool}$  **where**  
 $\text{rdomp } R \ a \equiv \exists h \ c. h \models R \ a \ c$

**abbreviation** *rdom*  $R \equiv \text{Collect } (\text{rdomp } R)$

**lemma** *rdomp-ctxt*[*simp*]:  $\text{rdomp } (\text{hn-ctxt } R) = \text{rdomp } R$   
 ⟨*proof*⟩

**lemma** *rdomp-pure*[*simp*]:  $\text{rdomp } (\text{pure } R) \ a \longleftrightarrow a \in \text{Range } R$   
 ⟨*proof*⟩

**lemma** *rdom-pure*[*simp*]:  $\text{rdom } (\text{pure } R) = \text{Range } R$   
 ⟨*proof*⟩

**lemma** *Range-of-constraint-conv*[*simp*]:  $\text{Range } (A \cap \text{UNIV} \times C) = \text{Range } A \cap C$   
 ⟨*proof*⟩

### 1.2.3 Heap-Nres Refinement Calculus

Predicate that expresses refinement. Given a heap  $\Gamma$ , program  $c$  produces a heap  $\Gamma'$  and a concrete result that is related with predicate  $R$  to some

abstract result from  $m$

**definition**  $hn\text{-refine } \Gamma \ c \ \Gamma' \ R \ m \equiv \text{nofail } m \longrightarrow$   
 $\langle \Gamma \rangle \ c \ \langle \lambda r. \Gamma' * (\exists_{Ax}. R \ x \ r * \uparrow(\text{RETURN } x \leq m)) \rangle_t$

$\langle ML \rangle$

**lemma**  $hn\text{-refineI}[\text{intro?}]$ :

**assumes**  $\text{nofail } m$

$\implies \langle \Gamma \rangle \ c \ \langle \lambda r. \Gamma' * (\exists_{Ax}. R \ x \ r * \uparrow(\text{RETURN } x \leq m)) \rangle_t$

**shows**  $hn\text{-refine } \Gamma \ c \ \Gamma' \ R \ m$

$\langle \text{proof} \rangle$

**lemma**  $hn\text{-refineD}$ :

**assumes**  $hn\text{-refine } \Gamma \ c \ \Gamma' \ R \ m$

**assumes**  $\text{nofail } m$

**shows**  $\langle \Gamma \rangle \ c \ \langle \lambda r. \Gamma' * (\exists_{Ax}. R \ x \ r * \uparrow(\text{RETURN } x \leq m)) \rangle_t$

$\langle \text{proof} \rangle$

**lemma**  $hn\text{-refine-preI}$ :

**assumes**  $\bigwedge h. h \models \Gamma \implies hn\text{-refine } \Gamma \ c \ \Gamma' \ R \ a$

**shows**  $hn\text{-refine } \Gamma \ c \ \Gamma' \ R \ a$

$\langle \text{proof} \rangle$

**lemma**  $hn\text{-refine-nofailI}$ :

**assumes**  $\text{nofail } a \implies hn\text{-refine } \Gamma \ c \ \Gamma' \ R \ a$

**shows**  $hn\text{-refine } \Gamma \ c \ \Gamma' \ R \ a$

$\langle \text{proof} \rangle$

**lemma**  $hn\text{-refine-false}[\text{simp}]$ :  $hn\text{-refine } \text{false} \ c \ \Gamma' \ R \ m$

$\langle \text{proof} \rangle$

**lemma**  $hn\text{-refine-fail}[\text{simp}]$ :  $hn\text{-refine } \Gamma \ c \ \Gamma' \ R \ \text{FAIL}$

$\langle \text{proof} \rangle$

**lemma**  $hn\text{-refine-frame}$ :

**assumes**  $hn\text{-refine } P' \ c \ Q' \ R \ m$

**assumes**  $P \implies_t F * P'$

**shows**  $hn\text{-refine } P \ c \ (F * Q') \ R \ m$

$\langle \text{proof} \rangle$

**lemma**  $hn\text{-refine-cons}$ :

**assumes**  $I: P \implies_t P'$

**assumes**  $R: hn\text{-refine } P' \ c \ Q \ R \ m$

**assumes**  $I': Q \implies_t Q'$

**assumes**  $R': \bigwedge x \ y. R \ x \ y \implies_t R' \ x \ y$

**shows**  $hn\text{-refine } P \ c \ Q' \ R' \ m$

$\langle \text{proof} \rangle$

**lemma** *hn-refine-cons-pre*:  
**assumes**  $I: P \Longrightarrow_t P'$   
**assumes**  $R: \text{hn-refine } P' \ c \ Q \ R \ m$   
**shows**  $\text{hn-refine } P \ c \ Q \ R \ m$   
 $\langle \text{proof} \rangle$

**lemma** *hn-refine-cons-post*:  
**assumes**  $R: \text{hn-refine } P \ c \ Q \ R \ m$   
**assumes**  $I: Q \Longrightarrow_t Q'$   
**shows**  $\text{hn-refine } P \ c \ Q' \ R \ m$   
 $\langle \text{proof} \rangle$

**lemma** *hn-refine-cons-res*:  
 $\llbracket \text{hn-refine } \Gamma \ f \ \Gamma' \ R \ g; \bigwedge a \ c. R \ a \ c \Longrightarrow_t R' \ a \ c \rrbracket \Longrightarrow \text{hn-refine } \Gamma \ f \ \Gamma' \ R' \ g$   
 $\langle \text{proof} \rangle$

**lemma** *hn-refine-ref*:  
**assumes**  $LE: m \leq m'$   
**assumes**  $R: \text{hn-refine } P \ c \ Q \ R \ m$   
**shows**  $\text{hn-refine } P \ c \ Q \ R \ m'$   
 $\langle \text{proof} \rangle$

**lemma** *hn-refine-cons-complete*:  
**assumes**  $I: P \Longrightarrow_t P'$   
**assumes**  $R: \text{hn-refine } P' \ c \ Q \ R \ m$   
**assumes**  $I': Q \Longrightarrow_t Q'$   
**assumes**  $R': \bigwedge x \ y. R \ x \ y \Longrightarrow_t R' \ x \ y$   
**assumes**  $LE: m \leq m'$   
**shows**  $\text{hn-refine } P \ c \ Q' \ R' \ m'$   
 $\langle \text{proof} \rangle$

**lemma** *hn-refine-augment-res*:  
**assumes**  $A: \text{hn-refine } \Gamma \ f \ \Gamma' \ R \ g$   
**assumes**  $B: g \leq_n \text{SPEC } \Phi$   
**shows**  $\text{hn-refine } \Gamma \ f \ \Gamma' \ (\lambda a \ c. R \ a \ c * \uparrow(\Phi \ a)) \ g$   
 $\langle \text{proof} \rangle$

## 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* ::  $('a1 \Rightarrow 'c1 \Rightarrow \text{assn}) \Rightarrow ('a2 \Rightarrow 'c2 \Rightarrow \text{assn})$   
 $\Rightarrow 'a1 * 'a2 \Rightarrow 'c1 * 'c2 \Rightarrow \text{assn}$  **where**  
 $\text{prod-assn } P1 \ P2 \ a \ c \equiv \text{case } (a, c) \text{ of } ((a1, a2), (c1, c2)) \Rightarrow$   
 $P1 \ a1 \ c1 * P2 \ a2 \ c2$

**notation** *prod-assn* (**infixr**  $\langle \times_a \rangle$  70)

**lemma** *prod-assn-pure-conv[simp]*:  $\text{prod-assn } (\text{pure } R1) (\text{pure } R2) = \text{pure } (R1 \times_r R2)$

*<proof>*

**lemma** *prod-assn-pair-conv[simp]*:

$\text{prod-assn } A B (a1, b1) (a2, b2) = A a1 a2 * B b1 b2$

*<proof>*

**lemma** *prod-assn-true[simp]*:  $\text{prod-assn } (\lambda \_ \cdot \text{true}) (\lambda \_ \cdot \text{true}) = (\lambda \_ \cdot \text{true})$

*<proof>*

### 1.2.5 Convenience Lemmas

**lemma** *hn-refine-guessI*:

**assumes** *hn-refine*  $P f P' R f'$

**assumes**  $f=f\text{-conc}$

**shows** *hn-refine*  $P f\text{-conc} P' R f'$

— To prove a refinement, first synthesize one, and then prove equality

*<proof>*

**lemma** *imp-correctI*:

**assumes**  $R$ : *hn-refine*  $\Gamma c \Gamma' R a$

**assumes**  $C$ :  $a \leq \text{SPEC } \Phi$

**shows**  $\langle \Gamma \rangle c \langle \lambda r'. \exists_{A r}. \Gamma' * R r r' * \uparrow(\Phi r) \rangle_t$

*<proof>*

**lemma** *hnr-pre-ex-conv*:

**shows** *hn-refine*  $(\exists_{A x}. \Gamma x) c \Gamma' R a \longleftrightarrow (\forall x. \text{hn-refine } (\Gamma x) c \Gamma' R a)$

*<proof>*

**lemma** *hnr-pre-pure-conv*:

**shows** *hn-refine*  $(\Gamma * \uparrow P) c \Gamma' R a \longleftrightarrow (P \longrightarrow \text{hn-refine } \Gamma c \Gamma' R a)$

*<proof>*

**lemma** *hn-refine-split-post*:

**assumes** *hn-refine*  $\Gamma c \Gamma' R a$

**shows** *hn-refine*  $\Gamma c (\Gamma' \vee_A \Gamma'') R a$

*<proof>*

**lemma** *hn-refine-post-other*:

**assumes** *hn-refine*  $\Gamma c \Gamma'' R a$

**shows** *hn-refine*  $\Gamma c (\Gamma' \vee_A \Gamma'') R a$

*<proof>*

### Return

**lemma** *hnr-RETURN-pass*:

*hn-refine*  $(\text{hn-ctxt } R x p) (\text{return } p) (\text{hn-invalid } R x p) R (\text{RETURN } x)$

— Pass on a value from the heap as return value



*<proof>*

**lemma** *hnr-RETURN-pure*:

**assumes**  $(c,a) \in R$

**shows**  $hn\text{-refine } emp (return\ c) \ emp (pure\ R) (RETURN\ a)$

— Return pure value

*<proof>*

## Assertion

**lemma** *hnr-FAIL[simp, intro!]*:  $hn\text{-refine } \Gamma\ c\ \Gamma'\ R\ FAIL$

*<proof>*

**lemma** *hnr-ASSERT*:

**assumes**  $\Phi \implies hn\text{-refine } \Gamma\ c\ \Gamma'\ R\ c'$

**shows**  $hn\text{-refine } \Gamma\ c\ \Gamma'\ R\ (do\ \{ ASSERT\ \Phi; c'\})$

*<proof>*

## Bind

**lemma** *bind-det-aux*:  $\llbracket RETURN\ x \leq m; RETURN\ y \leq f\ x \rrbracket \implies RETURN\ y \leq m \ggg f$

*<proof>*

**lemma** *hnr-bind*:

**assumes** *D1*:  $hn\text{-refine } \Gamma\ m'\ \Gamma1\ Rh\ m$

**assumes** *D2*:

$\bigwedge x\ x'. RETURN\ x \leq m \implies hn\text{-refine } (\Gamma1 * hn\text{-ctxt } Rh\ x\ x') (f'\ x') (\Gamma2\ x\ x')$   
 $R (f\ x)$

**assumes** *IMP*:  $\bigwedge x\ x'. \Gamma2\ x\ x' \implies_t \Gamma' * hn\text{-ctxt } Rx\ x\ x'$

**shows**  $hn\text{-refine } \Gamma (m' \ggg f') \Gamma' R (m \ggg f)$

*<proof>*

## Recursion

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

**lemma** *hn-refine-alt*:  $hn\text{-refine } Fpre\ c\ Fpost\ P\ m \equiv nofail\ m \longrightarrow$

$\langle Fpre \rangle\ c \langle \lambda r. hn\text{-rel } P\ m\ r * Fpost \rangle_t$

*<proof>*

**lemma** *wit-swap-forall*:

**assumes** *W*:  $\langle P \rangle\ c \langle \lambda-. true \rangle$

**assumes** *T*:  $(\forall x. A\ x \longrightarrow \langle P \rangle\ c \langle Q\ x \rangle)$

**shows**  $\langle P \rangle\ c \langle \lambda r. \neg_A (\exists_A x. \uparrow (A\ x) * \neg_A Q\ x\ r) \rangle$

*<proof>*

**apply1** (*rule models-in-range*)

**applyS** (*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))  
 ⟨*proof*⟩

**lemma** *hn-admissible*:

**assumes** *PREC*: *precise Ry*  
**assumes** *E*:  $\forall f \in A. \text{nofail } (f x) \longrightarrow \langle P \rangle c \langle \lambda r. \text{hn-rel } Ry (f x) r * F \rangle$   
**assumes** *NF*: *nofail (INF f ∈ A. f x)*  
**shows**  $\langle P \rangle c \langle \lambda r. \text{hn-rel } Ry (INF f \in A. f x) r * F \rangle$   
 ⟨*proof*⟩

**lemma** *hn-admissible'*:

**assumes** *PREC*: *precise Ry*  
**assumes** *E*:  $\forall f \in A. \text{nofail } (f x) \longrightarrow \langle P \rangle c \langle \lambda r. \text{hn-rel } Ry (f x) r * F \rangle_t$   
**assumes** *NF*: *nofail (INF f ∈ A. f x)*  
**shows**  $\langle P \rangle c \langle \lambda r. \text{hn-rel } Ry (INF f \in A. f x) r * F \rangle_t$   
 ⟨*proof*⟩

**lemma** *hnr-RECT-old*:

**assumes** *S*:  $\bigwedge cf \text{ af } ax \text{ px. } \llbracket$   
 $\bigwedge ax \text{ px. hn-refine } (hn-ctxt \text{ Rx } ax \text{ px} * F) (cf \text{ px}) (F' \text{ ax px}) Ry (af \text{ ax}) \rrbracket$   
 $\implies \text{hn-refine } (hn-ctxt \text{ Rx } ax \text{ px} * F) (cB \text{ cf px}) (F' \text{ ax px}) Ry (aB \text{ af ax})$   
**assumes** *M*:  $(\bigwedge x. \text{mono-Heap } (\lambda f. cB f x))$   
**assumes** *PREC*: *precise Ry*  
**shows** *hn-refine*  
 $(hn-ctxt \text{ Rx } ax \text{ px} * F) (\text{heap.fixp-fun } cB \text{ px}) (F' \text{ ax px}) Ry (RECT \text{ aB } ax)$   
 ⟨*proof*⟩

**lemma** *hnr-RECT*:

**assumes** *S*:  $\bigwedge cf \text{ af } ax \text{ px. } \llbracket$   
 $\bigwedge ax \text{ px. hn-refine } (hn-ctxt \text{ Rx } ax \text{ px} * F) (cf \text{ px}) (F' \text{ ax px}) Ry (af \text{ ax}) \rrbracket$   
 $\implies \text{hn-refine } (hn-ctxt \text{ Rx } ax \text{ px} * F) (cB \text{ cf px}) (F' \text{ ax px}) Ry (aB \text{ af ax})$   
**assumes** *M*:  $(\bigwedge x. \text{mono-Heap } (\lambda f. cB f x))$   
**shows** *hn-refine*  
 $(hn-ctxt \text{ Rx } ax \text{ px} * F) (\text{heap.fixp-fun } cB \text{ px}) (F' \text{ ax px}) Ry (RECT \text{ aB } ax)$   
 ⟨*proof*⟩

**lemma** *hnr-If*:

**assumes** *P*:  $\Gamma \implies_t \Gamma 1 * \text{hn-val bool-rel } a \text{ a}'$   
**assumes** *RT*:  $a \implies \text{hn-refine } (\Gamma 1 * \text{hn-val bool-rel } a \text{ a}') b' \Gamma 2b \text{ R } b$   
**assumes** *RE*:  $\neg a \implies \text{hn-refine } (\Gamma 1 * \text{hn-val bool-rel } a \text{ a}') c' \Gamma 2c \text{ R } c$   
**assumes** *IMP*:  $\Gamma 2b \vee_A \Gamma 2c \implies_t \Gamma'$   
**shows** *hn-refine*  $\Gamma$  (*if a' then b' else c'*)  $\Gamma' \text{ R}$  (*if a then b else c*)  
 ⟨*proof*⟩

**apply1** (*rule hn-refine-preI*)

**applyF** (*cases a; simp add: hn-ctxt-def pure-def*)

**focus**

**apply1** (*rule hn-refine-split-post*)

**applyF** (*rule hn-refine-cons-pre[OF - RT]*)

**applyS** (*simp add: hn-ctxt-def pure-def*)

```

    applyS simp
  solved
solved
apply1 (rule hn-refine-post-other)
applyF (rule hn-refine-cons-pre[OF - RE])
  applyS (simp add: hn-ctxt-def pure-def)
  applyS simp
  solved
solved
applyS (rule IMP)
applyS (rule entt-refl)
⟨proof⟩

```

### 1.2.6 ML-Level Utilities

⟨ML⟩

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]:  $SP\ x \equiv x$

**lemma** *SP-cong*[cong]:  $SP\ x \equiv SP\ x$  ⟨proof⟩

**lemma** *PR-CONST-cong*[cong]:  $PR-CONST\ x \equiv PR-CONST\ x$  ⟨proof⟩

**definition** *RCALL* — Tag that marks recursive call

**where** [simp]:  $RCALL\ D \equiv D$

**definition** *EVAL* — Tag that marks evaluation of plain expression for monadify phase

**where** [simp]:  $EVAL\ x \equiv RETURN\ x$

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*\$(*RETURN*\$x)\$( $\lambda_2x. f x$ ) =  $f x$   
*EVAL*\$x  $\equiv$  *RETURN*\$x  
 ⟨*proof*⟩

**definition** [*simp*]: *PASS*  $\equiv$  *RETURN*

— Pass on value, invalidating old one

**lemma** *remove-pass-simps*:

*Refine-Basic.bind*\$(*PASS*\$x)\$( $\lambda_2x. f x$ )  $\equiv$   $f x$   
*Refine-Basic.bind*\$m\$( $\lambda_2x. *PASS*$x$ )  $\equiv$   $m$   
 ⟨*proof*⟩

**definition** *COPY* :: 'a  $\Rightarrow$  'a

— Marks required copying of parameter

**where** [*simp*]: *COPY*  $x \equiv x$

**lemma** *RET-COPY-PASS-eq*: *RETURN*\$(*COPY*\$p) = *PASS*\$p ⟨*proof*⟩

**named-theorems-rev** *sepref-monadify-arity* *Sepref.Monadify*: *Arity alignment equations*

**named-theorems-rev** *sepref-monadify-comb* *Sepref.Monadify*: *Combinator equations*

⟨*ML*⟩

**lemma** *dflt-arity*[*sepref-monadify-arity*]:

*RETURN*  $\equiv$   $\lambda_2x. SP *RETURN*$x$   
*RECT*  $\equiv$   $\lambda_2B x. SP *RECT*$(\lambda_2D x. B$(\lambda_2x. *RCALL*$D$x)$x)$x$   
*case-list*  $\equiv$   $\lambda_2fn fc l. SP *case-list*$fn$(\lambda_2x xs. *fc*$x$xs)$l$   
*case-prod*  $\equiv$   $\lambda_2fp p. SP *case-prod*$(\lambda_2a b. *fp*$a$b)$p$   
*case-option*  $\equiv$   $\lambda_2fn fs ov. SP *case-option*$fn$(\lambda_2x. *fs*$x)$ov$   
*If*  $\equiv$   $\lambda_2b t e. SP *If*$b$t$e$   
*Let*  $\equiv$   $\lambda_2x f. SP *Let*$x$(\lambda_2x. *f*$x)$   
 ⟨*proof*⟩

**lemma** *dflt-comb*[*sepref-monadify-comb*]:

$\bigwedge B x. *RECT*$B$x  $\equiv$  *Refine-Basic.bind*$(*EVAL*$x)$(\lambda_2x. SP (*RECT*$B$x))$

$\wedge D x. RCALL\$D\$x \equiv Refine-Basic.bind\$(EVAL\$x)\$(\lambda_2x. SP (RCALL\$D\$x))$   
 $\wedge fn\ fc\ l. case-list\$fn\$fc\$l \equiv Refine-Basic.bind\$(EVAL\$l)\$(\lambda_2l. (SP case-list\$fn\$fc\$l))$   
 $\wedge fp\ p. case-prod\$fp\$p \equiv Refine-Basic.bind\$(EVAL\$p)\$(\lambda_2p. (SP case-prod\$fp\$p))$   
 $\wedge fn\ fs\ ov. case-option\$fn\$fs\$ov$   
 $\quad \equiv Refine-Basic.bind\$(EVAL\$ov)\$(\lambda_2ov. (SP case-option\$fn\$fs\$ov))$   
 $\wedge b\ t\ e. If\$b\$t\$e \equiv Refine-Basic.bind\$(EVAL\$b)\$(\lambda_2b. (SP If\$b\$t\$e))$   
 $\wedge x. RETURN\$x \equiv Refine-Basic.bind\$(EVAL\$x)\$(\lambda_2x. SP (RETURN\$x))$   
 $\wedge x\ f. Let\$x\$f \equiv Refine-Basic.bind\$(EVAL\$x)\$(\lambda_2x. (SP Let\$x\$f))$   
 $\langle proof \rangle$

**lemma** *dflt-plain-comb*[*sepref-monadify-comb*]:

$EVAL\$(If\$b\$t\$e) \equiv Refine-Basic.bind\$(EVAL\$b)\$(\lambda_2b. If\$b\$(EVAL\$t)\$(EVAL\$e))$   
 $EVAL\$(case-list\$fn\$(\lambda_2x\ xs. fc\ x\ xs)\$l) \equiv$   
 $\quad Refine-Basic.bind\$(EVAL\$l)\$(\lambda_2l. case-list\$(EVAL\$fn)\$(\lambda_2x\ xs. EVAL\$(fc\ x$   
 $\quad xs)\$l))$   
 $EVAL\$(case-prod\$(\lambda_2a\ b. fp\ a\ b)\$p) \equiv$   
 $\quad Refine-Basic.bind\$(EVAL\$p)\$(\lambda_2p. case-prod\$(\lambda_2a\ b. EVAL\$(fp\ a\ b)\$p))$   
 $EVAL\$(case-option\$fn\$(\lambda_2x. fs\ x)\$ov) \equiv$   
 $\quad Refine-Basic.bind\$(EVAL\$ov)\$(\lambda_2ov. case-option\$(EVAL\$fn)\$(\lambda_2x. EVAL\$(fs$   
 $\quad x)\$ov))$   
 $EVAL\$(Let\ \$\ v\ \$\ (\lambda_2x. f\ x)) \equiv (\gg) \$\ (EVAL\ \$\ v)\ \$\ (\lambda_2x. EVAL\ \$\ (f\ x))$   
 $\langle proof \rangle$

**lemma** *evalcomb-PR-CONST*[*sepref-monadify-comb*]:

$EVAL\$(PR-CONST\ x) \equiv SP (RETURN\$(PR-CONST\ x))$   
 $\langle proof \rangle$

**end**

**theory** *Sepref-Constraints*

**imports** *Main Automatic-Refinement.Refine-Lib Sepref-Basic*

**begin**

**definition** *CONSTRAINT-SLOT* ( $x::prop$ )  $\equiv x$

**lemma** *insert-slot-rl1*:

**assumes**  $PROP\ P \implies PROP (CONSTRAINT-SLOT (Trueprop\ True)) \implies$   
 $PROP\ Q$   
**shows**  $PROP (CONSTRAINT-SLOT (PROP\ P)) \implies PROP\ Q$   
 $\langle proof \rangle$

**lemma** *insert-slot-rl2*:

**assumes**  $PROP\ P \implies PROP (CONSTRAINT-SLOT\ S) \implies PROP\ Q$   
**shows**  $PROP (CONSTRAINT-SLOT (PROP\ S \&\&\& PROP\ P)) \implies PROP\ Q$   
 $\langle proof \rangle$

**lemma** *remove-slot*:  $PROP (CONSTRAINT-SLOT (Trueprop\ True))$

⟨proof⟩

**definition** *CONSTRAINT* **where** [simp]: *CONSTRAINT*  $P\ x \equiv P\ x$

**lemma** *CONSTRAINT-D*:

**assumes** *CONSTRAINT* ( $P::'a \Rightarrow \text{bool}$ )  $x$

**shows**  $P\ x$

⟨proof⟩

**lemma** *CONSTRAINT-I*:

**assumes**  $P\ x$

**shows** *CONSTRAINT* ( $P::'a \Rightarrow \text{bool}$ )  $x$

⟨proof⟩

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* :: ( $'a \Rightarrow \text{bool}$ )  $\Rightarrow 'a \Rightarrow \text{bool}$  **where** [simp]: *CN-FALSE*  $P\ x \equiv \text{False}$

**lemma** *CN-FALSEI*: *CN-FALSE*  $P\ x \Longrightarrow P\ x$  ⟨proof⟩

**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⟩

**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  $F1 \vee_A F2 \Longrightarrow_t ?F$  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* :: ( $'a \Rightarrow 'c \Rightarrow \text{assn}$ )  $\Rightarrow ('a \Rightarrow 'c \Rightarrow \text{assn}) \Rightarrow 'a \Rightarrow 'c \Rightarrow$

*assn*

**where** *mismatch-assn*  $R1\ R2\ x\ y \equiv R1\ x\ y \vee_A R2\ x\ y$

**abbreviation** *hn-mismatch*  $R1\ R2 \equiv hn-ctxt\ (mismatch-assn\ R1\ R2)$

**lemma** *recover-pure-aux*: *CONSTRAINT is-pure*  $R \implies hn-invalid\ R\ x\ y \implies_t$   
*hn-ctxt*  $R\ x\ y$

*<proof>*

**lemma** *frame-thms*:

$P \implies_t P$

$P \implies_t P' \implies F \implies_t F' \implies F * P \implies_t F' * P'$

*hn-ctxt*  $R\ x\ y \implies_t hn-invalid\ R\ x\ y$

*hn-ctxt*  $R\ x\ y \implies_t hn-ctxt\ (\lambda\ -. true)\ x\ y$

*CONSTRAINT is-pure*  $R \implies hn-invalid\ R\ x\ y \implies_t hn-ctxt\ R\ x\ y$

*<proof>*

**applyS** *simp*

**applyS** (*rule entt-star-mono; assumption*)

*<proof>*

**applyS** (*sep-auto simp: hn-ctxt-def*)

**applyS** (*erule recover-pure-aux*)

*<proof>*

**named-theorems-rev** *sepref-frame-match-rules* *<Sepref: Additional frame rules>*

Rules to discharge unmatched stuff

**lemma** *frame-rem1*:  $P \implies_t P$  *<proof>*

**lemma** *frame-rem2*:  $F \implies_t F' \implies F * hn-ctxt\ A\ x\ y \implies_t F' * hn-ctxt\ A\ x\ y$   
*<proof>*

**lemma** *frame-rem3*:  $F \implies_t F' \implies F * hn-ctxt\ A\ x\ y \implies_t F'$   
*<proof>*

**lemma** *frame-rem4*:  $P \implies_t emp$  *<proof>*

**lemmas** *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*:

$[ A \vee_A C \implies_A E; B \vee_A D \implies_A F ] \implies A * B \vee_A C * D \implies_A E * F$   
*<proof>*

**lemma** *entt-disj-star-mono*:

$[ A \vee_A C \implies_t E; B \vee_A D \implies_t F ] \implies A * B \vee_A C * D \implies_t E * F$

$\langle \text{proof} \rangle$

**lemma** *hn-merge1*:

$$\begin{aligned} & F \vee_A F \Longrightarrow_t F \\ & \llbracket \text{hn-ctxt } R1 \ x \ x' \vee_A \text{hn-ctxt } R2 \ x \ x' \Longrightarrow_t \text{hn-ctxt } R \ x \ x'; Fl \vee_A Fr \Longrightarrow_t F \rrbracket \\ & \Longrightarrow Fl * \text{hn-ctxt } R1 \ x \ x' \vee_A Fr * \text{hn-ctxt } R2 \ x \ x' \Longrightarrow_t F * \text{hn-ctxt } R \ x \ x' \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *hn-merge2*:

$$\begin{aligned} & \text{hn-invalid } R \ x \ x' \vee_A \text{hn-ctxt } R \ x \ x' \Longrightarrow_t \text{hn-invalid } R \ x \ x' \\ & \text{hn-ctxt } R \ x \ x' \vee_A \text{hn-invalid } R \ x \ x' \Longrightarrow_t \text{hn-invalid } R \ x \ x' \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *invalid-assn-mono*:  $\text{hn-ctxt } A \ x \ y \Longrightarrow_t \text{hn-ctxt } B \ x \ y$

$$\begin{aligned} & \Longrightarrow \text{hn-invalid } A \ x \ y \Longrightarrow_t \text{hn-invalid } B \ x \ y \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *hn-merge3*:

$$\begin{aligned} & \llbracket \text{NO-MATCH } (\text{hn-invalid } XX) \ R2; \text{hn-ctxt } R1 \ x \ x' \vee_A \text{hn-ctxt } R2 \ x \ x' \Longrightarrow_t \\ & \text{hn-ctxt } Rm \ x \ x' \rrbracket \Longrightarrow \text{hn-invalid } R1 \ x \ x' \vee_A \text{hn-ctxt } R2 \ x \ x' \Longrightarrow_t \text{hn-invalid } Rm \ x \\ & \ x' \\ & \llbracket \text{NO-MATCH } (\text{hn-invalid } XX) \ R1; \text{hn-ctxt } R1 \ x \ x' \vee_A \text{hn-ctxt } R2 \ x \ x' \Longrightarrow_t \\ & \text{hn-ctxt } Rm \ x \ x' \rrbracket \Longrightarrow \text{hn-ctxt } R1 \ x \ x' \vee_A \text{hn-invalid } R2 \ x \ x' \Longrightarrow_t \text{hn-invalid } Rm \ x \\ & \ x' \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemmas** *merge-thms* = *hn-merge1* *hn-merge2*

**named-theorems** *sepref-frame-merge-rules*  $\langle \text{Sepref: Additional merge rules} \rangle$

**lemma** *hn-merge-mismatch*:  $\text{hn-ctxt } R1 \ x \ x' \vee_A \text{hn-ctxt } R2 \ x \ x' \Longrightarrow_t \text{hn-mismatch}$   
 $R1 \ R2 \ x \ x'$

$\langle \text{proof} \rangle$

**lemma** *is-merge*:  $P1 \vee_A P2 \Longrightarrow_t P \Longrightarrow P1 \vee_A P2 \Longrightarrow_t P$   $\langle \text{proof} \rangle$

**lemma** *merge-mono*:  $\llbracket A \Longrightarrow_t A'; B \Longrightarrow_t B'; A \vee_A B' \Longrightarrow_t C \rrbracket \Longrightarrow A \vee_A B \Longrightarrow_t C$

$\langle \text{proof} \rangle$

Apply forward rule on left or right side of merge

**lemma** *gen-merge-cons1*:  $\llbracket A \Longrightarrow_t A'; A \vee_A B \Longrightarrow_t C \rrbracket \Longrightarrow A \vee_A B \Longrightarrow_t C$

$\langle \text{proof} \rangle$

**lemma** *gen-merge-cons2*:  $\llbracket B \Longrightarrow_t B'; A \vee_A B' \Longrightarrow_t C \rrbracket \Longrightarrow A \vee_A B \Longrightarrow_t C$

$\langle \text{proof} \rangle$



**lemmas**  $gen\text{-}merge\text{-}cons = gen\text{-}merge\text{-}cons1\ gen\text{-}merge\text{-}cons2$

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

**definition**  $RECOVER\text{-}PURE\ P\ Q \equiv P \Longrightarrow_t Q$

**lemma** *recover-pure*:

$RECOVER\text{-}PURE\ emp\ emp$   
 $\llbracket RECOVER\text{-}PURE\ P2\ Q2; RECOVER\text{-}PURE\ P1\ Q1 \rrbracket \Longrightarrow RECOVER\text{-}PURE$   
 $(P1 * P2)\ (Q1 * Q2)$   
 $CONSTRAINT\ is\text{-}pure\ R \Longrightarrow RECOVER\text{-}PURE\ (hn\text{-}invalid\ R\ x\ y)\ (hn\text{-}ctxt\ R$   
 $x\ y)$   
 $RECOVER\text{-}PURE\ (hn\text{-}ctxt\ R\ x\ y)\ (hn\text{-}ctxt\ R\ x\ y)$   
 $\langle proof \rangle$

**lemma** *recover-pure-triv*:

$RECOVER\text{-}PURE\ P\ P$   
 $\langle proof \rangle$

Weakening the postcondition by converting *invalid-assn* to  $\lambda\text{-}\cdot$ . *true*

**definition**  $WEAKEN\text{-}HNR\text{-}POST\ \Gamma\ \Gamma'\ \Gamma'' \equiv (\exists h. h \models \Gamma) \longrightarrow (\Gamma'' \Longrightarrow_t \Gamma')$

**lemma** *weaken-hnr-postI*:

**assumes**  $WEAKEN\text{-}HNR\text{-}POST\ \Gamma\ \Gamma''\ \Gamma'$   
**assumes**  $hn\text{-}refine\ \Gamma\ c\ \Gamma'\ R\ a$   
**shows**  $hn\text{-}refine\ \Gamma\ c\ \Gamma''\ R\ a$   
 $\langle proof \rangle$

**lemma** *weaken-hnr-post-triv*:  $WEAKEN\text{-}HNR\text{-}POST\ \Gamma\ P\ P$

$\langle proof \rangle$

**lemma** *weaken-hnr-post*:

$\llbracket WEAKEN\text{-}HNR\text{-}POST\ \Gamma\ P\ P'; WEAKEN\text{-}HNR\text{-}POST\ \Gamma'\ Q\ Q' \rrbracket \Longrightarrow WEAKEN\text{-}HNR\text{-}POST$   
 $(\Gamma * \Gamma')\ (P * Q)\ (P' * Q')$   
 $WEAKEN\text{-}HNR\text{-}POST\ (hn\text{-}ctxt\ R\ x\ y)\ (hn\text{-}ctxt\ R\ x\ y)\ (hn\text{-}ctxt\ R\ x\ y)$   
 $WEAKEN\text{-}HNR\text{-}POST\ (hn\text{-}ctxt\ R\ x\ y)\ (hn\text{-}invalid\ R\ x\ y)\ (hn\text{-}ctxt\ (\lambda\text{-}\cdot.\ true)\ x$   
 $y)$   
 $\langle proof \rangle$

**lemma** *reorder-enttI*:

**assumes**  $A * true = C * true$   
**assumes**  $B * true = D * true$   
**shows**  $(A \Longrightarrow_t B) \equiv (C \Longrightarrow_t D)$   
 $\langle proof \rangle$

**lemma** *merge-sat1*:  $(A \vee_A A' \implies_t Am) \implies (A \vee_A Am \implies_t Am)$   
 ⟨*proof*⟩  
**lemma** *merge-sat2*:  $(A \vee_A A' \implies_t Am) \implies (Am \vee_A A' \implies_t Am)$   
 ⟨*proof*⟩

⟨*ML*⟩

**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** [*sepref-frame-normrel-eqs*] = *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  $'a \Rightarrow 'b$ , 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* ::  $('a \Rightarrow - \Rightarrow \text{assn}) \Rightarrow 'b \text{ itself} \Rightarrow \text{bool}$  **where**  
 [*simp*]: *intf-of-assn* *a b* = *True*

**lemma** *intf-of-assnI*: *intf-of-assn* *R* *TYPE('a)* ⟨*proof*⟩

**named-theorems-rev** *intf-of-assn* ⟨*Links between refinement assertions and interface types*⟩

**lemma** *intf-of-assn-fallback*: *intf-of-assn* (*R* ::  $'a \Rightarrow - \Rightarrow \text{assn}$ ) *TYPE('a)* ⟨*proof*⟩

## 1.5.2 Function Refinement with Precondition

**definition**  $fref :: ('c \Rightarrow bool) \Rightarrow ('a \times 'c) \text{ set} \Rightarrow ('b \times 'd) \text{ set}$   
 $\Rightarrow (('a \Rightarrow 'b) \times ('c \Rightarrow 'd)) \text{ set}$

$\langle [-]_f - \rightarrow - \rangle [0,60,60] 60$

**where**  $[P]_f R \rightarrow S \equiv \{(f,g). \forall x y. P y \wedge (x,y) \in R \longrightarrow (f x, g y) \in S\}$

**abbreviation**  $fref_t \langle - \rightarrow_f - \rangle [60,60] 60$  **where**  $R \rightarrow_f S \equiv ([\lambda-. True]_f R \rightarrow S)$

**lemma**  $rel2p\text{-}fref[rel2p]$ :  $rel2p (fref P R S)$   
 $= (\lambda f g. (\forall x y. P y \longrightarrow rel2p R x y \longrightarrow rel2p S (f x) (g y)))$   
 $\langle proof \rangle$

**lemma**  $fref\text{-}cons$ :

**assumes**  $(f,g) \in [P]_f R \rightarrow S$

**assumes**  $\bigwedge c a. (c,a) \in R' \Longrightarrow Q a \Longrightarrow P a$

**assumes**  $R' \subseteq R$

**assumes**  $S \subseteq S'$

**shows**  $(f,g) \in [Q]_f R' \rightarrow S'$

$\langle proof \rangle$

**lemmas**  $fref\text{-}cons' = fref\text{-}cons[OF - - order\text{-}refl order\text{-}refl]$

**lemma**  $frefI[intro?]$ :

**assumes**  $\bigwedge x y. \llbracket P y; (x,y) \in R \rrbracket \Longrightarrow (f x, g y) \in S$

**shows**  $(f,g) \in fref P R S$

$\langle proof \rangle$

**lemma**  $fref\text{-}ncI$ :  $(f,g) \in R \rightarrow S \Longrightarrow (f,g) \in R \rightarrow_f S$

$\langle proof \rangle$

**lemma**  $frefD$ :

**assumes**  $(f,g) \in fref P R S$

**shows**  $\llbracket P y; (x,y) \in R \rrbracket \Longrightarrow (f x, g y) \in S$

$\langle proof \rangle$

**lemma**  $fref\text{-}ncD$ :  $(f,g) \in R \rightarrow_f S \Longrightarrow (f,g) \in R \rightarrow S$

$\langle proof \rangle$

**lemma**  $fref\text{-}compI$ :

$fref P R1 R2 O fref Q S1 S2 \subseteq$

$fref (\lambda x. Q x \wedge (\forall y. (y,x) \in S1 \longrightarrow P y)) (R1 O S1) (R2 O S2)$

$\langle proof \rangle$

**lemma**  $fref\text{-}compI'$ :

$\llbracket (f,g) \in fref P R1 R2; (g,h) \in fref Q S1 S2 \rrbracket$

$\Longrightarrow (f,h) \in fref (\lambda x. Q x \wedge (\forall y. (y,x) \in S1 \longrightarrow P y)) (R1 O S1) (R2 O S2)$

$\langle proof \rangle$

**lemma** *fref-unit-conv*:

$$(\lambda-. c, \lambda-. a) \in \text{fref } P \text{ unit-rel } S \iff (P () \longrightarrow (c, a) \in S)$$

*<proof>*

**lemma** *fref-uncurry-conv*:

$$(\text{uncurry } c, \text{uncurry } a) \in \text{fref } P (R1 \times_r R2) S$$

$$\iff (\forall x1 y1 x2 y2. P (y1, y2) \longrightarrow (x1, y1) \in R1 \longrightarrow (x2, y2) \in R2 \longrightarrow (c x1 x2, a y1 y2) \in S)$$

*<proof>*

**lemma** *fref-mono*:  $\llbracket \bigwedge x. P' x \implies P x; R' \subseteq R; S \subseteq S' \rrbracket$

$$\implies \text{fref } P R S \subseteq \text{fref } P' R' S'$$

*<proof>*

**lemma** *fref-composeI*:

**assumes** *FR1*:  $(f, g) \in \text{fref } P R1 R2$   
**assumes** *FR2*:  $(g, h) \in \text{fref } Q S1 S2$   
**assumes** *C1*:  $\bigwedge x. P' x \implies Q x$   
**assumes** *C2*:  $\bigwedge x y. \llbracket P' x; (y, x) \in S1 \rrbracket \implies P y$   
**assumes** *R1*:  $R' \subseteq R1 \circ S1$   
**assumes** *R2*:  $R2 \circ S2 \subseteq S'$   
**assumes** *FH*:  $f' = f \circ h' = h$   
**shows**  $(f', h') \in \text{fref } P' R' S'$

*<proof>*

**lemma** *fref-triv*:  $A \subseteq Id \implies (f, f) \in [P]_f A \rightarrow Id$

*<proof>*

### 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** *hfref*

$$\begin{aligned} &:: \\ &('a \Rightarrow \text{bool}) \\ &\Rightarrow (('a \Rightarrow 'ai \Rightarrow \text{assn}) \times ('a \Rightarrow 'ai \Rightarrow \text{assn})) \\ &\Rightarrow ('b \Rightarrow 'bi \Rightarrow \text{assn}) \\ &\Rightarrow (('ai \Rightarrow 'bi \text{ Heap}) \times ('a \Rightarrow 'b \text{ nres})) \text{ set} \\ &(\langle [-]_a \rightarrow \rightarrow [0, 60, 60] 60 \rangle) \end{aligned}$$

**where**

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

**abbreviation** *hfrefT*  $\langle \cdot \rightarrow_a \rightarrow [60, 60] 60 \rangle$  **where**  $RS \rightarrow_a T \equiv ([\lambda-. \text{True}]_a RS \rightarrow T)$

**lemma** *hfrefI[intro?]*:

**assumes**  $\bigwedge c a. P a \implies \text{hn-refine } (fst\ RS\ a\ c)\ (f\ c)\ (snd\ RS\ a\ c)\ T\ (g\ a)$   
**shows**  $(f,g) \in \text{hfref } P\ RS\ T$   
 $\langle \text{proof} \rangle$

**lemma** *hfrefD*:

**assumes**  $(f,g) \in \text{hfref } P\ RS\ T$   
**shows**  $\bigwedge c a. P a \implies \text{hn-refine } (fst\ RS\ a\ c)\ (f\ c)\ (snd\ RS\ a\ c)\ T\ (g\ a)$   
 $\langle \text{proof} \rangle$

**lemma** *hfref-to-ASSERT-conv*:

*NO-MATCH*  $(\lambda-. True)\ P \implies (a,b) \in [P]_a\ R \rightarrow S \longleftrightarrow (a, \lambda x. \text{ASSERT } (P\ x))$   
 $\gg b\ x) \in R \rightarrow_a S$   
 $\langle \text{proof} \rangle$

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*

$:: ('a \Rightarrow 'b \Rightarrow \text{assn}) \Rightarrow \text{bool} \Rightarrow ('a \Rightarrow 'b \Rightarrow \text{assn}) \times ('a \Rightarrow 'b \Rightarrow \text{assn})$   
**where**  
 $\text{hf-pres } R\ \text{True} = (R,R) \mid \text{hf-pres } R\ \text{False} = (R, \text{invalid-assn } R)$

**abbreviation** *hfkeep*

$:: ('a \Rightarrow 'b \Rightarrow \text{assn}) \Rightarrow ('a \Rightarrow 'b \Rightarrow \text{assn}) \times ('a \Rightarrow 'b \Rightarrow \text{assn})$   
 $\langle (-^k) \rangle [1000]\ 999$

**where**  $R^k \equiv \text{hf-pres } R\ \text{True}$

**abbreviation** *hfdrop*

$:: ('a \Rightarrow 'b \Rightarrow \text{assn}) \Rightarrow ('a \Rightarrow 'b \Rightarrow \text{assn}) \times ('a \Rightarrow 'b \Rightarrow \text{assn})$   
 $\langle (-^d) \rangle [1000]\ 999$

**where**  $R^d \equiv \text{hf-pres } R\ \text{False}$

**abbreviation** *hn-kede*  $R\ kd \equiv \text{hn-ctxt } (snd\ (\text{hf-pres } R\ kd))$

**abbreviation** *hn-keep*  $R \equiv \text{hn-kede } R\ \text{True}$

**abbreviation** *hn-dest*  $R \equiv \text{hn-kede } R\ \text{False}$

**lemma** *keep-drop-sels[simp]*:

$\text{fst } (R^k) = R$   
 $\text{snd } (R^k) = R$   
 $\text{fst } (R^d) = R$   
 $\text{snd } (R^d) = \text{invalid-assn } R$   
 $\langle \text{proof} \rangle$

**lemma** *hf-pres-fst[simp]*:  $\text{fst } (\text{hf-pres } R\ k) = R$   $\langle \text{proof} \rangle$

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* ::

$(( 'a \Rightarrow 'b \Rightarrow \text{assn}) \times ('a \Rightarrow 'b \Rightarrow \text{assn}))$

$\Rightarrow (('c \Rightarrow 'd \Rightarrow \text{assn}) \times ('c \Rightarrow 'd \Rightarrow \text{assn}))$   
 $\Rightarrow (((('a \times 'c) \Rightarrow ('b \times 'd) \Rightarrow \text{assn}) \times (('a \times 'c) \Rightarrow ('b \times 'd) \Rightarrow \text{assn}))$   
 (infixl <\*\_a> 65)  
 where  $RR *_a SS \equiv (\text{prod-assn } (fst RR) (fst SS), \text{prod-assn } (snd RR) (snd SS))$

**lemma** *hfprod-fst-snd[simp]*:  
 $\text{fst } (A *_a B) = \text{prod-assn } (fst A) (fst B)$   
 $\text{snd } (A *_a B) = \text{prod-assn } (snd A) (snd B)$   
 <proof>

## Conversion from fref to hfref

**lemma** *fref-to-pure-hfref'*:  
 assumes  $(f, g) \in [P]_f R \rightarrow \langle S \rangle \text{nres-rel}$   
 assumes  $\bigwedge x. x \in \text{Domain } R \cap R^{-1} \text{Collect } P \implies f x = \text{RETURN } (f' x)$   
 shows  $(\text{return } o f', g) \in [P]_a (\text{pure } R)^k \rightarrow \text{pure } S$   
 <proof>

## Conversion from hfref to hnr

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

**lemma** *hf2hnr*:  
 assumes  $(f, g) \in [P]_a R \rightarrow S$   
 shows  $\forall x xi. P x \longrightarrow \text{hn-refine } (\text{emp } * \text{hn-ctxt } (fst R) x xi) (f \$ xi) (\text{emp } * \text{hn-ctxt } (snd R) x xi) S (g \$ x)$   
 <proof>

**definition** [simp]:  $\text{to-hnr-prod} \equiv \text{prod-assn}$

**lemma** *to-hnr-prod-fst-snd*:  
 $\text{fst } (A *_a B) = \text{to-hnr-prod } (fst A) (fst B)$   
 $\text{snd } (A *_a B) = \text{to-hnr-prod } (snd A) (snd B)$   
 <proof>

**lemma** *hnr-uncurry-unfold*:  
 $(\forall x xi. P x \longrightarrow$   
 $\text{hn-refine}$   
 $(\Gamma * \text{hn-ctxt } (\text{to-hnr-prod } A B) x xi)$   
 $(f x)$   
 $(\Gamma' * \text{hn-ctxt } (\text{to-hnr-prod } A' B') x xi)$   
 $R$   
 $(f x))$   
 $\longleftrightarrow (\forall b bi a ai. P (a, b) \longrightarrow$   
 $\text{hn-refine}$

$$\begin{array}{l}
(\Gamma * \text{hn-ctxt } B \ b \ bi * \text{hn-ctxt } A \ a \ ai) \\
(fi \ (ai,bi)) \\
(\Gamma' * \text{hn-ctxt } B' \ b \ bi * \text{hn-ctxt } A' \ a \ ai) \\
R \\
(f \ (a,b)) \\
) \\
\langle \text{proof} \rangle
\end{array}$$

**lemma** *hnr-intro-dummy*:

$$\forall x \ xi. P \ x \longrightarrow \text{hn-refine } (\Gamma \ x \ xi) \ (c \ xi) \ (\Gamma' \ x \ xi) \ R \ (a \ x) \Longrightarrow \forall x \ xi. P \ x \longrightarrow \text{hn-refine } (\text{emp} * \Gamma \ x \ xi) \ (c \ xi) \ (\text{emp} * \Gamma' \ x \ xi) \ R \ (a \ x)$$

$\langle \text{proof} \rangle$

**lemma** *hn-ctxt-ctxt-fix-conv*:  $\text{hn-ctxt } (\text{hn-ctxt } R) = \text{hn-ctxt } R$

$\langle \text{proof} \rangle$

**lemma** *uncurry-APP*:  $\text{uncurry } f(a,b) = f\$a\$b$   $\langle \text{proof} \rangle$

**lemma** *norm-RETURN-o*:

$$\begin{array}{l}
\wedge f. (\text{RETURN } o \ f)\$x = (\text{RETURN}\$(f\$x)) \\
\wedge f. (\text{RETURN } oo \ f)\$x\$y = (\text{RETURN}\$(f\$x\$y)) \\
\wedge f. (\text{RETURN } ooo \ f)\$x\$y\$z = (\text{RETURN}\$(f\$x\$y\$z)) \\
\wedge f. (\lambda x. \text{RETURN } ooo \ f \ x)\$x\$y\$z\$a = (\text{RETURN}\$(f\$x\$y\$z\$a)) \\
\wedge f. (\lambda x \ y. \text{RETURN } ooo \ f \ x \ y)\$x\$y\$z\$a\$b = (\text{RETURN}\$(f\$x\$y\$z\$a\$b))
\end{array}$$

$\langle \text{proof} \rangle$

**lemma** *norm-return-o*:

$$\begin{array}{l}
\wedge f. (\text{return } o \ f)\$x = (\text{return}\$(f\$x)) \\
\wedge f. (\text{return } oo \ f)\$x\$y = (\text{return}\$(f\$x\$y)) \\
\wedge f. (\text{return } ooo \ f)\$x\$y\$z = (\text{return}\$(f\$x\$y\$z)) \\
\wedge f. (\lambda x. \text{return } ooo \ f \ x)\$x\$y\$z\$a = (\text{return}\$(f\$x\$y\$z\$a)) \\
\wedge f. (\lambda x \ y. \text{return } ooo \ f \ x \ y)\$x\$y\$z\$a\$b = (\text{return}\$(f\$x\$y\$z\$a\$b))
\end{array}$$

$\langle \text{proof} \rangle$

**lemma** *hn-val-unit-conv-emp[simp]*:  $\text{hn-val unit-rel } x \ y = \text{emp}$

$\langle \text{proof} \rangle$

## Conversion from hnr to href

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

**abbreviation** *id-assn*  $\equiv \text{pure } Id$

**abbreviation** *(input) unit-assn*  $\equiv \text{id-assn} :: \text{unit} \Rightarrow -$

**lemma** *pure-unit-rel-eq-empty*:  $\text{unit-assn } x \ y = \text{emp}$

$\langle \text{proof} \rangle$

**lemma** *uc-hfprod-sel*:

$\text{fst } (A *_a B) a c = (\text{case } (a,c) \text{ of } ((a1,a2),(c1,c2)) \Rightarrow \text{fst } A a1 c1 * \text{fst } B a2 c2)$   
 $\text{snd } (A *_a B) a c = (\text{case } (a,c) \text{ of } ((a1,a2),(c1,c2)) \Rightarrow \text{snd } A a1 c1 * \text{snd } B a2 c2)$   
 <proof>

## Conversion from relation to fref

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

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

**lemma**  $\text{fref-param1}: R \rightarrow S = \text{fref } (\lambda-. \text{True}) R S$   
 <proof>

**lemma**  $\text{fref-nest}: \text{fref } P1 R1 (\text{fref } P2 R2 S)$   
 $\equiv CURRY (\text{fref } (\lambda(a,b). P1 a \wedge P2 b) (R1 \times_r R2) S)$   
 <proof>

**lemma**  $\text{in-CURRY-conv}: (f,g) \in CURRY R \longleftrightarrow (\text{uncurry } f, \text{uncurry } g) \in R$   
 <proof>

**lemma**  $\text{uncurry0-APP[simp]}: \text{uncurry0 } c \$ x = c$  <proof>

**lemma**  $\text{fref-param0I}: (c,a) \in R \implies (\text{uncurry0 } c, \text{uncurry0 } a) \in \text{fref } (\lambda-. \text{True})$   
 $\text{unit-rel } R$   
 <proof>

## Composition

**definition**  $\text{hr-comp} :: ('b \Rightarrow 'c \Rightarrow \text{assn}) \Rightarrow ('b \times 'a) \text{ set} \Rightarrow 'a \Rightarrow 'c \Rightarrow \text{assn}$   
 — Compose refinement assertion with refinement relation  
**where**  $\text{hr-comp } R1 R2 a c \equiv \exists_A b. R1 b c * \uparrow((b,a) \in R2)$

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

**lemma**  $\text{hr-compI}: (b,a) \in R2 \implies R1 b c \implies_A \text{hr-comp } R1 R2 a c$   
 <proof>

**lemma**  $\text{hr-comp-Id1[simp]}: \text{hr-comp } (\text{pure } \text{Id}) R = \text{pure } R$   
 <proof>

**lemma**  $\text{hr-comp-Id2[simp]}: \text{hr-comp } R \text{ Id} = R$   
 <proof>



**lemma** *hr-comp-emp*[*simp*]:  $hr\text{-comp } (\lambda a c. emp) R a c = \uparrow(\exists b. (b,a)\in R)$   
 ⟨*proof*⟩

**lemma** *hr-comp-prod-conv*[*simp*]:  
 $hr\text{-comp } (prod\text{-assn } Ra Rb) (Ra' \times_r Rb')$   
 $= prod\text{-assn } (hr\text{-comp } Ra Ra') (hr\text{-comp } Rb Rb')$   
 ⟨*proof*⟩

**lemma** *hr-comp-pure*:  $hr\text{-comp } (pure R) S = pure (R O S)$   
 ⟨*proof*⟩

**lemma** *hr-comp-is-pure*[*safe-constraint-rules*]:  $is\text{-pure } A \implies is\text{-pure } (hr\text{-comp } A B)$   
 ⟨*proof*⟩

**lemma** *hr-comp-the-pure*:  $is\text{-pure } A \implies the\text{-pure } (hr\text{-comp } A B) = the\text{-pure } A O B$   
 ⟨*proof*⟩

**lemma** *rdomp-hrcomp-conv*:  $rdomp (hr\text{-comp } A R) x \longleftrightarrow (\exists y. rdomp A y \wedge (y,x)\in R)$   
 ⟨*proof*⟩

**lemma** *hn-rel-compI*:  
 $\llbracket nofail a; (b,a)\in \langle R2 \rangle nres\text{-rel} \rrbracket \implies hn\text{-rel } R1 b c \implies_A hn\text{-rel } (hr\text{-comp } R1 R2) a c$   
 ⟨*proof*⟩

**lemma** *hr-comp-precise*[*constraint-rules*]:  
**assumes** [*safe-constraint-rules*]: *precise R*  
**assumes** *SV*: *single-valued S*  
**shows** *precise (hr-comp R S)*  
 ⟨*proof*⟩

**lemma** *hr-comp-assoc*:  $hr\text{-comp } (hr\text{-comp } R S) T = hr\text{-comp } R (S O T)$   
 ⟨*proof*⟩

**lemma** *hnr-comp*:  
**assumes** *R*:  $\bigwedge b1 c1. P b1 \implies hn\text{-refine } (R1 b1 c1 * \Gamma) (c c1) (R1p b1 c1 * \Gamma')$  *R* (*b b1*)  
**assumes** *S*:  $\bigwedge a1 b1. \llbracket Q a1; (b1,a1)\in R1 \rrbracket \implies (b b1, a a1)\in \langle R' \rangle nres\text{-rel}$   
**assumes** *PQ*:  $\bigwedge a1 b1. \llbracket Q a1; (b1,a1)\in R1 \rrbracket \implies P b1$   
**assumes** *Q*:  $Q a1$   
**shows** *hn-refine*  
 $(hr\text{-comp } R1 R1' a1 c1 * \Gamma)$   
 $(c c1)$   
 $(hr\text{-comp } R1p R1' a1 c1 * \Gamma')$

$(hr\text{-comp } R \ R')$   
 $(a \ a1)$   
 $\langle proof \rangle$

**lemma** *hnr-comp1-aux*:

**assumes**  $R: \bigwedge b1 \ c1. P \ b1 \implies hn\text{-refine } (hn\text{-ctxt } R1 \ b1 \ c1) \ (c \ c1) \ (hn\text{-ctxt } R1p \ b1 \ c1) \ R \ (b\$b1)$   
**assumes**  $S: \bigwedge a1 \ b1. \llbracket Q \ a1; (b1, a1) \in R1 \rrbracket \implies (b\$b1, a\$a1) \in \langle R' \rangle nres\text{-rel}$   
**assumes**  $PQ: \bigwedge a1 \ b1. \llbracket Q \ a1; (b1, a1) \in R1 \rrbracket \implies P \ b1$   
**assumes**  $Q: Q \ a1$   
**shows** *hn-refine*  
 $(hr\text{-comp } R1 \ R1' \ a1 \ c1)$   
 $(c \ c1)$   
 $(hr\text{-comp } R1p \ R1' \ a1 \ c1)$   
 $(hr\text{-comp } R \ R')$   
 $(a \ a1)$   
 $\langle proof \rangle$

**lemma** *hfcomp*:

**assumes**  $A: (f, g) \in [P]_a \ RR' \rightarrow S$   
**assumes**  $B: (g, h) \in [Q]_f \ T \rightarrow \langle U \rangle nres\text{-rel}$   
**shows**  $(f, h) \in [\lambda a. Q \ a \wedge (\forall a'. (a', a) \in T \longrightarrow P \ a')]_a$   
 $hrp\text{-comp } RR' \ T \rightarrow hr\text{-comp } S \ U$   
 $\langle proof \rangle$

**lemma** *hfref-weaken-pre-nofail*:

**assumes**  $(f, g) \in [P]_a \ R \rightarrow S$   
**shows**  $(f, g) \in [\lambda x. nofail \ (g \ x) \longrightarrow P \ x]_a \ R \rightarrow S$   
 $\langle proof \rangle$

**lemma** *hfref-cons*:

**assumes**  $(f, g) \in [P]_a \ R \rightarrow S$   
**assumes**  $\bigwedge x. P' \ x \implies P \ x$   
**assumes**  $\bigwedge x \ y. fst \ R' \ x \ y \implies_t fst \ R \ x \ y$   
**assumes**  $\bigwedge x \ y. snd \ R \ x \ y \implies_t snd \ R' \ x \ y$   
**assumes**  $\bigwedge x \ y. S \ x \ y \implies_t S' \ x \ y$   
**shows**  $(f, g) \in [P']_a \ R' \rightarrow S'$   
 $\langle proof \rangle$

## Composition Automation

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

**lemma** *prod-hrp-comp*:

$hrp\text{-comp } (A \ *_a \ B) \ (C \ \times_r \ D) = hrp\text{-comp } A \ C \ *_a \ hrp\text{-comp } B \ D$   
 $\langle proof \rangle$

**lemma** *hrp-comp-keep*:  $hrp\text{-comp } (A^k) \ B = (hr\text{-comp } A \ B)^k$

$\langle proof \rangle$

**lemma** *hr-comp-invalid*:  $hr\text{-}comp\ (invalid\text{-}assn\ R1)\ R2 = invalid\text{-}assn\ (hr\text{-}comp\ R1\ R2)$

$\langle proof \rangle$

**lemma** *hrp-comp-dest*:  $hrp\text{-}comp\ (A^d)\ B = (hr\text{-}comp\ A\ B)^d$

$\langle proof \rangle$

**definition** *hrp-imp*  $RR\ RR' \equiv$

$\forall a\ b. (fst\ RR'\ a\ b \implies_t fst\ RR\ a\ b) \wedge (snd\ RR\ a\ b \implies_t snd\ RR'\ a\ b)$

**lemma** *hfref-imp*:  $hrp\text{-}imp\ RR\ RR' \implies [P]_a\ RR \rightarrow S \subseteq [P]_a\ RR' \rightarrow S$

$\langle proof \rangle$

**lemma** *hrp-imp-refl*:  $hrp\text{-}imp\ RR\ RR$

$\langle proof \rangle$

**lemma** *hrp-imp-reflI*:  $RR = RR' \implies hrp\text{-}imp\ RR\ RR'$

$\langle proof \rangle$

**lemma** *hrp-comp-cong*:  $hrp\text{-}imp\ A\ A' \implies B=B' \implies hrp\text{-}imp\ (hrp\text{-}comp\ A\ B)$   
 $(hrp\text{-}comp\ A'\ B')$

$\langle proof \rangle$

**lemma** *hrp-prod-cong*:  $hrp\text{-}imp\ A\ A' \implies hrp\text{-}imp\ B\ B' \implies hrp\text{-}imp\ (A *_a B)$   
 $(A' *_a B')$

$\langle proof \rangle$

**lemma** *hrp-imp-trans*:  $hrp\text{-}imp\ A\ B \implies hrp\text{-}imp\ B\ C \implies hrp\text{-}imp\ A\ C$

$\langle proof \rangle$

**lemma** *fcomp-norm-dflt-init*:  $x \in [P]_a\ R \rightarrow T \implies hrp\text{-}imp\ R\ S \implies x \in [P]_a\ S \rightarrow T$

$\langle proof \rangle$

**definition** *comp-PRE*  $R\ P\ Q\ S \equiv \lambda x. S\ x \longrightarrow (P\ x \wedge (\forall y. (y,x) \in R \longrightarrow Q\ x\ y))$

**lemma** *comp-PRE-cong[cong]*:

**assumes**  $R \equiv R'$

**assumes**  $\bigwedge x. P\ x \equiv P'\ x$

**assumes**  $\bigwedge x. S\ x \equiv S'\ x$

**assumes**  $\bigwedge x\ y. \llbracket P\ x; (y,x) \in R; y \in \text{Domain}\ R; S'\ x \rrbracket \implies Q\ x\ y \equiv Q'\ x\ y$

**shows**  $comp\text{-}PRE\ R\ P\ Q\ S \equiv comp\text{-}PRE\ R'\ P'\ Q'\ S'$

$\langle proof \rangle$

**lemma** *fref-compI-PRE*:

$\llbracket (f,g) \in fref\ P\ R1\ R2; (g,h) \in fref\ Q\ S1\ S2 \rrbracket$

$\implies (f,h) \in \text{fref } (\text{comp-PRE } S1 \ Q \ (\lambda\cdot. P) \ (\lambda\cdot. \text{True})) \ (R1 \ O \ S1) \ (R2 \ O \ S2)$   
 $\langle \text{proof} \rangle$

**lemma** *PRE-D1*:  $(Q \ x \wedge P \ x) \longrightarrow \text{comp-PRE } S1 \ Q \ (\lambda x \cdot P \ x) \ S \ x$   
 $\langle \text{proof} \rangle$

**lemma** *PRE-D2*:  $(Q \ x \wedge (\forall y. (y,x) \in S1 \longrightarrow S \ x \longrightarrow P \ x \ y)) \longrightarrow \text{comp-PRE } S1$   
 $Q \ P \ S \ x$   
 $\langle \text{proof} \rangle$

**lemma** *fref-weaken-pre*:  
**assumes**  $\bigwedge x. P \ x \longrightarrow P' \ x$   
**assumes**  $(f,h) \in \text{fref } P' \ R \ S$   
**shows**  $(f,h) \in \text{fref } P \ R \ S$   
 $\langle \text{proof} \rangle$

**lemma** *fref-PRE-D1*:  
**assumes**  $(f,h) \in \text{fref } (\text{comp-PRE } S1 \ Q \ (\lambda x \cdot P \ x) \ X) \ R \ S$   
**shows**  $(f,h) \in \text{fref } (\lambda x. Q \ x \wedge P \ x) \ R \ S$   
 $\langle \text{proof} \rangle$

**lemma** *fref-PRE-D2*:  
**assumes**  $(f,h) \in \text{fref } (\text{comp-PRE } S1 \ Q \ P \ X) \ R \ S$   
**shows**  $(f,h) \in \text{fref } (\lambda x. Q \ x \wedge (\forall y. (y,x) \in S1 \longrightarrow X \ x \longrightarrow P \ x \ y)) \ R \ S$   
 $\langle \text{proof} \rangle$

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

**lemma** *hfref-weaken-pre*:  
**assumes**  $\bigwedge x. P \ x \longrightarrow P' \ x$   
**assumes**  $(f,h) \in \text{hfref } P' \ R \ S$   
**shows**  $(f,h) \in \text{hfref } P \ R \ S$   
 $\langle \text{proof} \rangle$

**lemma** *hfref-weaken-pre'*:  
**assumes**  $\bigwedge x. \llbracket P \ x; \text{rdomp } (\text{fst } R) \ x \rrbracket \implies P' \ x$   
**assumes**  $(f,h) \in \text{hfref } P' \ R \ S$   
**shows**  $(f,h) \in \text{hfref } P \ R \ S$   
 $\langle \text{proof} \rangle$

**lemma** *hfref-weaken-pre-nofail'*:  
**assumes**  $(f,g) \in [P]_a \ R \rightarrow S$   
**assumes**  $\bigwedge x. \llbracket \text{nofail } (g \ x); \ Q \ x \rrbracket \implies P \ x$   
**shows**  $(f,g) \in [Q]_a \ R \rightarrow S$   
 $\langle \text{proof} \rangle$

**lemma** *hfref-compI-PRE-aux*:  
**assumes**  $A: (f,g) \in [P]_a \ RR' \rightarrow S$   
**assumes**  $B: (g,h) \in [Q]_f \ T \rightarrow \langle U \rangle \text{nres-rel}$

**shows**  $(f,h) \in [comp-PRE\ T\ Q\ (\lambda\cdot\ P)\ (\lambda\cdot\ True)]_a$   
 $hrp-comp\ RR'\ T \rightarrow hr-comp\ S\ U$   
 $\langle proof \rangle$

**lemma** *hfref-compI-PRE*:  
**assumes**  $A: (f,g) \in [P]_a\ RR' \rightarrow S$   
**assumes**  $B: (g,h) \in [Q]_f\ T \rightarrow \langle U \rangle nres-rel$   
**shows**  $(f,h) \in [comp-PRE\ T\ Q\ (\lambda x\ y.\ P\ y)\ (\lambda x.\ nofail\ (h\ x))]_a$   
 $hrp-comp\ RR'\ T \rightarrow hr-comp\ S\ U$   
 $\langle proof \rangle$

**lemma** *hfref-PRE-D1*:  
**assumes**  $(f,h) \in hfref\ (comp-PRE\ S1\ Q\ (\lambda x\ \cdot.\ P\ x)\ X)\ R\ S$   
**shows**  $(f,h) \in hfref\ (\lambda x.\ Q\ x \wedge P\ x)\ R\ S$   
 $\langle proof \rangle$

**lemma** *hfref-PRE-D2*:  
**assumes**  $(f,h) \in hfref\ (comp-PRE\ S1\ Q\ P\ X)\ R\ S$   
**shows**  $(f,h) \in hfref\ (\lambda x.\ Q\ x \wedge (\forall y.\ (y,x) \in S1 \longrightarrow X\ x \longrightarrow P\ x\ y))\ R\ S$   
 $\langle proof \rangle$

**lemma** *hfref-PRE-D3*:  
**assumes**  $(f,h) \in hfref\ (comp-PRE\ S1\ Q\ P\ X)\ R\ S$   
**shows**  $(f,h) \in hfref\ (comp-PRE\ S1\ Q\ P\ X)\ R\ S$   
 $\langle proof \rangle$

**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 hnr conversion

**named-theorems** *to-hnr-post*  $\langle to-hnr\ converter: Postprocessing\ unfold\ rules \rangle$

**lemma** *uncurry0-add-app-tag*:  $uncurry0\ (RETURN\ c) = uncurry0\ (RETURN\ \$c)$   
 $\langle proof \rangle$

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

**named-theorems** *to-hfref-post*  $\langle to-hfref\ converter: Postprocessing\ unfold\ rules \rangle$

**lemma** *prod-casesK[to-hfref-post]*:  $case-prod\ (\lambda\cdot\ \cdot.\ k) = (\lambda\cdot\ \cdot.\ k)$   $\langle proof \rangle$

**lemma** *uncurry0-hfref-post[to-hfref-post]*:  $hfref\ (uncurry0\ True)\ R\ S = hfref\ (\lambda\cdot\ True)\ R\ S$   
 $\langle proof \rangle$

Configuration for relation normalization after composition

**named-theorems** *fcomp-norm-unfold*  $\langle$ fcomp-normalizer: *Unfold theorems* $\rangle$   
**named-theorems** *fcomp-norm-simps*  $\langle$ fcomp-normalizer: *Simplification theorems* $\rangle$   
**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  $\implies$  pure (the-pure P) = P*  $\langle$ proof $\rangle$

**lemmas** [*fcomp-norm-simps*] = *True-implies-equals*

**lemmas** [*fcomp-norm-init*] = *fcomp-norm-dftt-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)  $\in$  [P]<sub>f</sub> R  $\rightarrow$  S  $\implies$  (RETURN o f, RETURN o g)  $\in$  [P]<sub>f</sub> R  $\rightarrow$  (S) nres-rel*  $\langle$ proof $\rangle$

**lemma** *ensure-fref-nres-unfold:*

$\bigwedge f. \text{RETURN } o (\text{uncurry0 } f) = \text{uncurry0 } (\text{RETURN } f)$

$\bigwedge f. \text{RETURN } o (\text{uncurry } f) = \text{uncurry } (\text{RETURN } oo f)$

$\bigwedge f. (\text{RETURN } ooo \text{uncurry}) f = \text{uncurry } (\text{RETURN } ooo f)$

$\langle$ proof $\rangle$

Composed precondition normalizer

**named-theorems** *fcomp-prenorm-simps*  $\langle$ fcomp precondition-normalizer: *Simplification theorems* $\rangle$

Support for preconditions of the form  $\neg \text{Domain } R$ , where  $R$  is the relation of the next more abstract level.

**declare** *DomainI* [*fcomp-prenorm-simps*]

**lemma** *auto-weaken-pre-init-hf:*

**assumes**  $\bigwedge x. \text{PROTECT } P x \longrightarrow P' x$

**assumes**  $(f,h) \in \text{hhref } P' R S$

**shows**  $(f,h) \in \text{href} P R S$   
 $\langle \text{proof} \rangle$

**lemma** *auto-weaken-pre-init-f*:  
**assumes**  $\bigwedge x. \text{PROTECT } P x \longrightarrow P' x$   
**assumes**  $(f,h) \in \text{fref } P' R S$   
**shows**  $(f,h) \in \text{fref } P R S$   
 $\langle \text{proof} \rangle$

**lemmas** *auto-weaken-pre-init = auto-weaken-pre-init-hf auto-weaken-pre-init-f*

**lemma** *auto-weaken-pre-uncurry-step*:  
**assumes**  $\text{PROTECT } f a \equiv f'$   
**shows**  $\text{PROTECT } (\lambda(x,y). f x y) (a,b) \equiv f' b$   
 $\langle \text{proof} \rangle$

**lemma** *auto-weaken-pre-uncurry-finish*:  
 $\text{PROTECT } f x \equiv f x \langle \text{proof} \rangle$

**lemma** *auto-weaken-pre-uncurry-start*:  
**assumes**  $P \equiv P'$   
**assumes**  $P' \longrightarrow Q$   
**shows**  $P \longrightarrow Q$   
 $\langle \text{proof} \rangle$

**lemma** *auto-weaken-pre-comp-PRE-I*:  
**assumes**  $S x \implies P x$   
**assumes**  $\bigwedge y. \llbracket (y,x) \in R; P x; S x \rrbracket \implies Q x y$   
**shows**  $\text{comp-PRE } R P Q S x$   
 $\langle \text{proof} \rangle$

**lemma** *auto-weaken-pre-to-imp-nf*:  
 $(A \longrightarrow B \longrightarrow C) = (A \wedge B \longrightarrow C)$   
 $((A \wedge B) \wedge C) = (A \wedge B \wedge C)$   
 $\langle \text{proof} \rangle$

**lemma** *auto-weaken-pre-add-dummy-imp*:  
 $P \implies \text{True} \longrightarrow P \langle \text{proof} \rangle$

Synthesis for href statements

**definition** *hfsynth-ID-R* ::  $('a \Rightarrow - \Rightarrow \text{assn}) \Rightarrow 'a \Rightarrow \text{bool}$  **where**  
 $[\text{simp}]: \text{hfsynth-ID-R } - \equiv \text{True}$

**lemma** *hfsynth-ID-R-D*:  
**fixes**  $I :: 'a \text{ itself}$   
**assumes** *hfsynth-ID-R*  $R a$   
**assumes** *intf-of-assn*  $R I$   
**shows**  $a ::_i I$   
 $\langle \text{proof} \rangle$

```

lemma hfsynth-hnr-from-hfI:
  assumes  $\forall x xi. P x \wedge \text{hfsynth-ID-R } (fst R) x \longrightarrow \text{hn-refine } (emp * \text{hn-ctxt } (fst R) x xi) (f\$xi) (emp * \text{hn-ctxt } (snd R) x xi) S (g\$x)$ 
  shows  $(f,g) \in [P]_a R \rightarrow S$ 
   $\langle proof \rangle$ 

```

```

lemma hfsynth-ID-R-uncurry-unfold:
   $\text{hfsynth-ID-R } (to-hnr-prod R S) (a,b) \equiv \text{hfsynth-ID-R } R a \wedge \text{hfsynth-ID-R } S b$ 
   $\text{hfsynth-ID-R } (fst (hf-pres R k)) \equiv \text{hfsynth-ID-R } R$ 
   $\langle proof \rangle$ 

```

$\langle ML \rangle$

**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 :: string  $\Rightarrow$  unit list  $\Rightarrow$  unit where T - -  $\equiv$  ()
  private lemma unit-eq:  $(a::unit) \equiv b$   $\langle proof \rangle$ 
  named-theorems --itype-rewrite

```

$\langle ML \rangle$

**end**

### 1.6.2 Rewriting Inferred Interface Types

```

definition map-type-eq :: 'a itself  $\Rightarrow$  'b itself  $\Rightarrow$  bool
  (infixr  $\langle \rightarrow_{nt} \rangle$  60)
  where [simp]: map-type-eq - -  $\equiv$  True
lemma map-type-eqI: map-type-eq L R  $\langle proof \rangle$ 

```

```

named-theorems-rew map-type-eqs

```



### 1.6.3 ML-Code

**context begin**

**private lemma** *start-eval*:  $x \equiv SP\ x$  *<proof>* **lemma** *add-eval*:  $f\ x \equiv (\gg) \$ (EVAL\ \$x) \$ (\lambda_2\ x.\ f\ x)$  *<proof>* **lemma** *init-mk-arity*:  $f \equiv id\ (SP\ f)$  *<proof>* **lemma** *add-mk-arity*:  $id\ f \equiv (\lambda_2\ x.\ id\ (f\ \$x))$  *<proof>* **lemma** *finish-mk-arity*:  $id\ f \equiv f$  *<proof>*

*<ML>*

**end**

*<ML>*

### 1.6.4 Obsolete Manual Setup Rules

**lemma**

*mk-mcomb1*:  $\bigwedge c.\ c\ \$x1 \equiv (\gg) \$ (EVAL\ \$x1) \$ (\lambda_2\ x1.\ SP\ (c\ \$x1))$   
**and** *mk-mcomb2*:  $\bigwedge c.\ c\ \$x1\ \$x2 \equiv (\gg) \$ (EVAL\ \$x1) \$ (\lambda_2\ x1.\ (\gg) \$ (EVAL\ \$x2) \$ (\lambda_2\ x2.\ SP\ (c\ \$x1\ \$x2)))$   
**and** *mk-mcomb3*:  $\bigwedge c.\ c\ \$x1\ \$x2\ \$x3 \equiv (\gg) \$ (EVAL\ \$x1) \$ (\lambda_2\ x1.\ (\gg) \$ (EVAL\ \$x2) \$ (\lambda_2\ x2.\ (\gg) \$ (EVAL\ \$x3) \$ (\lambda_2\ x3.\ SP\ (c\ \$x1\ \$x2\ \$x3))))$   
*<proof>*

**end**

## 1.7 Translation

**theory** *Sepref-Translate*

**imports**

*Sepref-Monadify*  
*Sepref-Constraints*  
*Sepref-Frame*  
*Lib/Pf-Mono-Prover*  
*Sepref-Rules*  
*Sepref-Combinator-Setup*  
*Lib/User-Smashing*

**begin**

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*  $x\ m \equiv \text{RETURN } x \leq m$

Tag to keep track of preconditions in assertions

**definition** *vassn-tag*  $\Gamma \equiv \exists h. h \models \Gamma$

**lemma** *vassn-tagI*:  $h \models \Gamma \implies \text{vassn-tag } \Gamma$   
*<proof>*

**lemma** *vassn-dest[dest!]*:

$\text{vassn-tag } (\Gamma_1 * \Gamma_2) \implies \text{vassn-tag } \Gamma_1 \wedge \text{vassn-tag } \Gamma_2$

$\text{vassn-tag } (\text{hn-ctxt } R\ a\ b) \implies a \in \text{rdom } R$

*<proof>*

**lemma** *entails-preI*:

**assumes**  $\text{vassn-tag } A \implies A \implies_A B$

**shows**  $A \implies_A B$

*<proof>*

**lemma** *invalid-assn-const*:

$\text{invalid-assn } (\lambda\ - . P)\ x\ y = \uparrow(\text{vassn-tag } P) * \text{true}$

*<proof>*

**lemma** *vassn-tag-simps[simp]*:

$\text{vassn-tag } \text{emp}$

$\text{vassn-tag } \text{true}$

*<proof>*

**definition** *GEN-ALGO*  $f\ \Phi \equiv \Phi\ f$

— Tag to synthesize  $f$  with property  $\Phi$ .

**lemma** *is-GEN-ALGO*:  $\text{GEN-ALGO } f\ \Phi \implies \text{GEN-ALGO } f\ \Phi$  *<proof>*

Tag for side-condition solver to discharge by assumption

**definition**  $RPREM :: \text{bool} \Rightarrow \text{bool}$  **where**  $[simp]: RPREM P = P$   
**lemma**  $RPREMI: P \Longrightarrow RPREM P$   $\langle \text{proof} \rangle$

**lemma** *trans-frame-rule*:  
**assumes**  $RECOVER-PURE \Gamma \Gamma'$   
**assumes**  $vassn\text{-}tag \Gamma' \Longrightarrow hn\text{-}refine \Gamma' c \Gamma'' R a$   
**shows**  $hn\text{-}refine (F*\Gamma) c (F*\Gamma'') R a$   
 $\langle \text{proof} \rangle$   
**applyF**  $(rule\ hn\text{-}refine\text{-}cons\text{-}pre)$   
**focus**  $\langle \text{proof} \rangle$  **solved**  
  
**apply1**  $(rule\ hn\text{-}refine\text{-}preI)$   
**apply1**  $(rule\ assms)$   
**applyS**  $(auto\ simp\ add: vassn\text{-}tag\text{-}def)$   
**solved**  
 $\langle \text{proof} \rangle$

**lemma** *recover-pure-cons*: — Used for debugging  
**assumes**  $RECOVER-PURE \Gamma \Gamma'$   
**assumes**  $hn\text{-}refine \Gamma' c \Gamma'' R a$   
**shows**  $hn\text{-}refine (\Gamma) c (\Gamma'') R a$   
 $\langle \text{proof} \rangle$

**definition**  $CPR-TAG :: \text{assn} \Rightarrow \text{assn} \Rightarrow \text{bool}$  **where**  $[simp]: CPR-TAG y x \equiv True$

**lemma**  $CPR-TAG\text{-}starI$ :  
**assumes**  $CPR-TAG P1 Q1$   
**assumes**  $CPR-TAG P2 Q2$   
**shows**  $CPR-TAG (P1*P2) (Q1*Q2)$   
 $\langle \text{proof} \rangle$

**lemma**  $CPR\text{-}tag\text{-}ctxtI: CPR-TAG (hn\text{-}ctxt R x xi) (hn\text{-}ctxt R' x xi)$   $\langle \text{proof} \rangle$

**lemma**  $CPR\text{-}tag\text{-}fallbackI: CPR-TAG P Q$   $\langle \text{proof} \rangle$

**lemmas**  $CPR-TAG\text{-}rules = CPR-TAG\text{-}starI CPR\text{-}tag\text{-}ctxtI CPR\text{-}tag\text{-}fallbackI$

**lemma** *cons-pre-rule*: — Consequence rule to be applied if no direct operation rule matches  
**assumes**  $CPR-TAG P P'$   
**assumes**  $P \Longrightarrow_t P'$   
**assumes**  $hn\text{-}refine P' c Q R m$   
**shows**  $hn\text{-}refine P c Q R m$   
 $\langle \text{proof} \rangle$

**named-theorems-rev**  $sepref\text{-}gen\text{-}algo\text{-}rules$   $\langle Sepref: \text{Generic algorithm rules} \rangle$

$\langle ML \rangle$

## Basic Setup

**lemma**  $hn\text{-}pass[sepref\text{-}fr\text{-}rules]$ :

**shows**  $hn\text{-refine } (hn\text{-ctxt } P \ x \ x') \ (return \ x') \ (hn\text{-invalid } P \ x \ x') \ P \ (PASS\$x)$   
 $\langle proof \rangle$

**lemma**  $hn\text{-bind}[sepref\text{-comb-rules}]$ :

**assumes**  $D1$ :  $hn\text{-refine } \Gamma \ m' \ \Gamma 1 \ Rh \ m$

**assumes**  $D2$ :

$\bigwedge x \ x'. \ bind\text{-ref-tag } x \ m \implies$

$hn\text{-refine } (\Gamma 1 \ * \ hn\text{-ctxt } Rh \ x \ x') \ (f' \ x') \ (\Gamma 2 \ x \ x') \ R \ (f \ x)$

**assumes**  $IMP$ :  $\bigwedge x \ x'. \ \Gamma 2 \ x \ x' \implies_t \ \Gamma' \ * \ hn\text{-ctxt } Rx \ x \ x'$

**shows**  $hn\text{-refine } \Gamma \ (m' \gg f') \ \Gamma' \ R \ (Refine\text{-Basic.bind}\$m\$(\lambda_2 x. f \ x))$

$\langle proof \rangle$

**lemma**  $hn\text{-RECT}'[sepref\text{-comb-rules}]$ :

**assumes**  $INDEP \ Ry \ INDEP \ Rx \ INDEP \ Rx'$

**assumes**  $FR$ :  $P \implies_t \ hn\text{-ctxt } Rx \ ax \ px \ * \ F$

**assumes**  $S$ :  $\bigwedge cf \ af \ ax \ px. \ \llbracket$

$\bigwedge ax \ px. \ hn\text{-refine } (hn\text{-ctxt } Rx \ ax \ px \ * \ F) \ (cf \ px) \ (hn\text{-ctxt } Rx' \ ax \ px \ * \ F) \ Ry$   
 $(RCALL\$af\$ax)\rrbracket$

$\implies \ hn\text{-refine } (hn\text{-ctxt } Rx \ ax \ px \ * \ F) \ (cB \ cf \ px) \ (F' \ ax \ px) \ Ry$   
 $(aB \ af \ ax)$

**assumes**  $FR'$ :  $\bigwedge ax \ px. \ F' \ ax \ px \implies_t \ hn\text{-ctxt } Rx' \ ax \ px \ * \ F$

**assumes**  $M$ :  $(\bigwedge x. \ mono\text{-Heap } (\lambda f. \ cB \ f \ x))$

**shows**  $hn\text{-refine}$

$(P) \ (heap.\text{fixp-fun } cB \ px) \ (hn\text{-ctxt } Rx' \ ax \ px \ * \ F) \ Ry$   
 $(RECT\$(\lambda_2 D \ x. \ aB \ D \ x)\$ax)$

$\langle proof \rangle$

**lemma**  $hn\text{-RCALL}[sepref\text{-comb-rules}]$ :

**assumes**  $RPREM$   $(hn\text{-refine } P' \ c \ Q' \ R \ (RCALL \ \$ \ a \ \$ \ b))$

**and**  $P \implies_t \ F \ * \ P'$

**shows**  $hn\text{-refine } P \ c \ (F \ * \ Q') \ R \ (RCALL \ \$ \ a \ \$ \ b)$

$\langle proof \rangle$

**definition**  $monadic\text{-WHILEIT } I \ b \ f \ s \equiv \ do \ \{$

$RECT \ (\lambda D \ s. \ do \ \{$

$ASSERT \ (I \ s);$

$bv \leftarrow b \ s;$

$\text{if } bv \text{ then do } \{$

$s \leftarrow f \ s;$

$D \ s$

$\} \text{ else do } \{RETURN \ s\}$

$\}) \ s$

$\}$

**definition** *heap-WHILET*  $b f s \equiv do \{$   
 $heap.fixp-fun (\lambda D s. do \{$   
 $bv \leftarrow b s;$   
 $if bv then do \{$   
 $$s \leftarrow f s;$   
 $$D s$   
 $\} else do \{return s\}$   
 $\}) s$   
 $\}$$$

**lemma** *heap-WHILET-unfold*[code]: *heap-WHILET*  $b f s =$   
 $do \{$   
 $bv \leftarrow b s;$   
 $if bv then do \{$   
 $$s \leftarrow f s;$   
 $$heap-WHILET b f s$   
 $\} else$   
 $$return s$   
 $\}$   
 $\langle proof \rangle$$$$

**lemma** *WHILEIT-to-monadic*: *WHILEIT*  $I b f s = monadic-WHILEIT I (\lambda s.$   
 $RETURN (b s)) f s$   
 $\langle proof \rangle$

**lemma** *WHILEIT-pat*[def-pat-rules]:  
 $WHILEIT \$I \equiv UNPROTECT (WHILEIT I)$   
 $WHILET \equiv PR-CONST (WHILEIT (\lambda-. True))$   
 $\langle proof \rangle$

**lemma** *id-WHILEIT*[id-rules]:  
 $PR-CONST (WHILEIT I) ::_i TYPE (('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a nres) \Rightarrow 'a \Rightarrow 'a$   
 $nres)$   
 $\langle proof \rangle$

**lemma** *WHILE-arities*[sepref-monadify-arity]:

$PR-CONST (WHILEIT I) \equiv \lambda_2 b f s. SP (PR-CONST (WHILEIT I)) \$ (\lambda_2 s.$   
 $b \$ s) \$ (\lambda_2 s. f \$ s) \$ s$   
 $\langle proof \rangle$

**lemma** *WHILEIT-comb*[sepref-monadify-comb]:

$PR-CONST (WHILEIT I) \$ (\lambda_2 x. b x) \$ f \$ s \equiv$   
 $Refine-Basic.bind \$ (EVAL \$ s) \$ (\lambda_2 s.$   
 $SP (PR-CONST (monadic-WHILEIT I)) \$ (\lambda_2 x. (EVAL \$ (b x))) \$ f \$ s$   
 $)$   
 $\langle proof \rangle$

**lemma** *hn-monadic-WHILE-aux*:

**assumes** *FR*:  $P \Longrightarrow_t \Gamma * \text{hn-ctxt } Rs \ s' \ s$

**assumes** *b-ref*:  $\bigwedge s \ s'. I \ s' \Longrightarrow \text{hn-refine}$   
 $(\Gamma * \text{hn-ctxt } Rs \ s' \ s)$   
 $(b \ s)$   
 $(\Gamma b \ s' \ s)$   
*(pure bool-rel)*  
 $(b' \ s')$

**assumes** *b-fr*:  $\bigwedge s' \ s. \Gamma b \ s' \ s \Longrightarrow_t \Gamma * \text{hn-ctxt } Rs \ s' \ s$

**assumes** *f-ref*:  $\bigwedge s' \ s. \llbracket I \ s' \rrbracket \Longrightarrow \text{hn-refine}$   
 $(\Gamma * \text{hn-ctxt } Rs \ s' \ s)$   
 $(f \ s)$   
 $(\Gamma f \ s' \ s)$   
 $Rs$   
 $(f' \ s')$

**assumes** *f-fr*:  $\bigwedge s' \ s. \Gamma f \ s' \ s \Longrightarrow_t \Gamma * \text{hn-ctxt } (\lambda - . \text{true}) \ s' \ s$

**shows** *hn-refine*  $(P)$  (*heap-WHILET*  $b \ f \ s$ )  $(\Gamma * \text{hn-invalid } Rs \ s' \ s) \ Rs$  (*monadic-WHILEIT*  $I \ b' \ f' \ s'$ )  
 $\langle \text{proof} \rangle$

**apply1** (*rule hn-refine-cons-pre*[*OF FR*])  
 $\langle \text{proof} \rangle$

**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*[*OF - b-ref b-fr entt-refl*])  
**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*])  
 $\langle \text{proof} \rangle$   
**solved**

**focus** (*rule hn-refine-frame*)  
**applyS** *rprems*  
**applyS** (*rule enttI*; *solve-entails*)  
**solved**

$\langle \text{proof} \rangle$   
**solved**  
**applyF** (*sep-auto*, *rule hn-refine-frame*)

**applyS** (rule *hnr-RETURN-pass*)  
 $\langle proof \rangle$   
**solved**  
  
 $\langle proof \rangle$   
**solved**  
  
 $\langle proof \rangle$   
**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**  
 $\langle proof \rangle$   
**solved**  
 $\langle proof \rangle$

**lemma** *hn-monadic-WHILE-lin*[*seprel-comb-rules*]:  
**assumes** *INDEP*  $R_s$   
**assumes** *FR*:  $P \Longrightarrow_t \Gamma * hn-ctxt\ R_s\ s'\ s$   
**assumes** *b-ref*:  $\bigwedge s\ s'. I\ s' \Longrightarrow hn-refine$   
 $(\Gamma * hn-ctxt\ R_s\ s'\ s)$   
 $(b\ s)$   
 $(\Gamma b\ s'\ s)$   
 $(pure\ bool-rel)$   
 $(b'\ s')$   
**assumes** *b-fr*:  $\bigwedge s'\ s. TERM\ (monadic-WHILEIT, "cond") \Longrightarrow \Gamma b\ s'\ s \Longrightarrow_t \Gamma * hn-ctxt\ R_s\ s'\ s$   
  
**assumes** *f-ref*:  $\bigwedge s'\ s. I\ s' \Longrightarrow hn-refine$   
 $(\Gamma * hn-ctxt\ R_s\ s'\ s)$   
 $(f\ s)$   
 $(\Gamma f\ s'\ s)$   
 $R_s$   
 $(f'\ s')$   
**assumes** *f-fr*:  $\bigwedge s'\ s. TERM\ (monadic-WHILEIT, "body") \Longrightarrow \Gamma f\ s'\ s \Longrightarrow_t \Gamma * hn-ctxt\ (\lambda -. true)\ s'\ s$   
**shows** *hn-refine*  
 $P$   
 $(heap-WHILET\ b\ f\ s)$   
 $(\Gamma * hn-invalid\ R_s\ s'\ s)$   
 $R_s$   
 $(PR-CONST\ (monadic-WHILEIT\ I)\ $(\lambda_2 s'. b'\ s')\ $(\lambda_2 s'. f'\ s')\ $(s'))$   
 $\langle proof \rangle$

**lemma** *monadic-WHILEIT-refine*[*refine*]:

**assumes** [*refine*]:  $(s',s) \in R$   
**assumes** [*refine*]:  $\bigwedge s' s. \llbracket (s',s) \in R; I s \rrbracket \implies I' s'$   
**assumes** [*refine*]:  $\bigwedge s' s. \llbracket (s',s) \in R; I s; I' s' \rrbracket \implies b' s' \leq \Downarrow \text{bool-rel } (b s)$   
**assumes** [*refine*]:  $\bigwedge s' s. \llbracket (s',s) \in R; I s; I' s'; \text{nofail } (b s); \text{inres } (b s) \text{ True} \rrbracket \implies f' s' \leq \Downarrow R (f s)$   
**shows** *monadic-WHILEIT*  $I' b' f' s' \leq \Downarrow R$  (*monadic-WHILEIT*  $I b f s$ )  
*<proof>*

**lemma** *monadic-WHILEIT-refine-WHILEIT*[*refine*]:

**assumes** [*refine*]:  $(s',s) \in R$   
**assumes** [*refine*]:  $\bigwedge s' s. \llbracket (s',s) \in R; I s \rrbracket \implies I' s'$   
**assumes** [*THEN order-trans,refine-vcg*]:  $\bigwedge s' s. \llbracket (s',s) \in R; I s; I' s' \rrbracket \implies b' s' \leq \text{SPEC } (\lambda r. r = b s)$   
**assumes** [*refine*]:  $\bigwedge s' s. \llbracket (s',s) \in R; I s; I' s'; b s \rrbracket \implies f' s' \leq \Downarrow R (f s)$   
**shows** *monadic-WHILEIT*  $I' b' f' s' \leq \Downarrow R$  (*WHILEIT*  $I b f s$ )  
*<proof>*

**lemma** *monadic-WHILEIT-refine-WHILET*[*refine*]:

**assumes** [*refine*]:  $(s',s) \in R$   
**assumes** [*THEN order-trans,refine-vcg*]:  $\bigwedge s' s. \llbracket (s',s) \in R \rrbracket \implies b' s' \leq \text{SPEC } (\lambda r. r = b s)$   
**assumes** [*refine*]:  $\bigwedge s' s. \llbracket (s',s) \in R; b s \rrbracket \implies f' s' \leq \Downarrow R (f s)$   
**shows** *monadic-WHILEIT*  $(\lambda-. \text{True}) b' f' s' \leq \Downarrow R$  (*WHILET*  $b f s$ )  
*<proof>*

**lemma** *monadic-WHILEIT-pat*[*def-pat-rules*]:

*monadic-WHILEIT*  $I \equiv \text{UNPROTECT } (\text{monadic-WHILEIT } I)$   
*<proof>*

**lemma** *id-monadic-WHILEIT*[*id-rules*]:

*PR-CONST* (*monadic-WHILEIT*  $I$ )  $::_i \text{TYPE}(( 'a \Rightarrow \text{bool nres}) \Rightarrow ( 'a \Rightarrow 'a \text{ nres}))$   
 $\Rightarrow 'a \Rightarrow 'a \text{ nres}$   
*<proof>*

**lemma** *monadic-WHILEIT-arities*[*sepref-monadify-arity*]:

*PR-CONST* (*monadic-WHILEIT*  $I$ )  $\equiv \lambda_2 b f s. \text{SP } (\text{PR-CONST } (\text{monadic-WHILEIT } I)) (\lambda_2 s. b \$ s) (\lambda_2 s. f \$ s) \$ s$   
*<proof>*

**lemma** *monadic-WHILEIT-comb*[*sepref-monadify-comb*]:

*PR-CONST* (*monadic-WHILEIT*  $I$ )  $\$ b \$ f \$ s \equiv \text{Refine-Basic.bind} \$ (\text{EVAL} \$ s) \$ (\lambda_2 s. \text{SP } (\text{PR-CONST } (\text{monadic-WHILEIT } I)) \$ b \$ f \$ s)$   
*<proof>*



**definition** [simp]:  $op\text{-}ASSERT\text{-}bind\ I\ m \equiv Refine\text{-}Basic.bind\ (ASSERT\ I)\ (\lambda\cdot. m)$

**lemma**  $pat\text{-}ASSERT\text{-}bind$ [def-pat-rules]:

$Refine\text{-}Basic.bind\$(ASSERT\$I)\$(\lambda_2\cdot. m) \equiv UNPROTECT\ (op\text{-}ASSERT\text{-}bind\ I)\$m$   
 ⟨proof⟩

**term**  $PR\text{-}CONST\ (op\text{-}ASSERT\text{-}bind\ I)$

**lemma**  $id\text{-}op\text{-}ASSERT\text{-}bind$ [id-rules]:

$PR\text{-}CONST\ (op\text{-}ASSERT\text{-}bind\ I) ::_i\ TYPE('a\ nres \Rightarrow 'a\ nres)$   
 ⟨proof⟩

**lemma**  $arity\text{-}ASSERT\text{-}bind$ [sepref-monadify-arity]:

$PR\text{-}CONST\ (op\text{-}ASSERT\text{-}bind\ I) \equiv \lambda_2 m. SP\ (PR\text{-}CONST\ (op\text{-}ASSERT\text{-}bind\ I))\$m$   
 ⟨proof⟩

**lemma**  $hn\text{-}ASSERT\text{-}bind$ [sepref-comb-rules]:

**assumes**  $I \Longrightarrow hn\text{-}refine\ \Gamma\ c\ \Gamma'\ R\ m$   
**shows**  $hn\text{-}refine\ \Gamma\ c\ \Gamma'\ R\ (PR\text{-}CONST\ (op\text{-}ASSERT\text{-}bind\ I))\$m$   
 ⟨proof⟩

**definition** [simp]:  $op\text{-}ASSUME\text{-}bind\ I\ m \equiv Refine\text{-}Basic.bind\ (ASSUME\ I)\ (\lambda\cdot. m)$

**lemma**  $pat\text{-}ASSUME\text{-}bind$ [def-pat-rules]:

$Refine\text{-}Basic.bind\$(ASSUME\$I)\$(\lambda_2\cdot. m) \equiv UNPROTECT\ (op\text{-}ASSUME\text{-}bind\ I)\$m$   
 ⟨proof⟩

**lemma**  $id\text{-}op\text{-}ASSUME\text{-}bind$ [id-rules]:

$PR\text{-}CONST\ (op\text{-}ASSUME\text{-}bind\ I) ::_i\ TYPE('a\ nres \Rightarrow 'a\ nres)$   
 ⟨proof⟩

**lemma**  $arity\text{-}ASSUME\text{-}bind$ [sepref-monadify-arity]:

$PR\text{-}CONST\ (op\text{-}ASSUME\text{-}bind\ I) \equiv \lambda_2 m. SP\ (PR\text{-}CONST\ (op\text{-}ASSUME\text{-}bind\ I))\$m$   
 ⟨proof⟩

**lemma**  $hn\text{-}ASSUME\text{-}bind$ [sepref-comb-rules]:

**assumes**  $vassn\text{-}tag\ \Gamma \Longrightarrow I$   
**assumes**  $I \Longrightarrow hn\text{-}refine\ \Gamma\ c\ \Gamma'\ R\ m$   
**shows**  $hn\text{-}refine\ \Gamma\ c\ \Gamma'\ R\ (PR\text{-}CONST\ (op\text{-}ASSUME\text{-}bind\ I))\$m$   
 ⟨proof⟩

### 1.7.1 Import of Parametricity Theorems

**lemma**  $pure\text{-}hn\text{-}refineI$ :

**assumes**  $Q \longrightarrow (c, a) \in R$   
**shows**  $hn\text{-}refine\ (\uparrow Q)\ (return\ c)\ (\uparrow Q)\ (pure\ R)\ (RETURN\ a)$   
 ⟨proof⟩

**lemma** *pure-hn-refineI-no-asm*:

**assumes**  $(c,a) \in R$

**shows**  $hn\text{-refine } emp \text{ (return } c) \text{ emp (pure } R) \text{ (RETURN } a)$

$\langle proof \rangle$

**lemma** *import-param-0*:

$(P \Longrightarrow Q) \equiv Trueprop \text{ (PROTECT } P \longrightarrow Q)$

$\langle proof \rangle$

**lemma** *import-param-1*:

$(P \Longrightarrow Q) \equiv Trueprop \text{ (} P \longrightarrow Q)$

$(P \longrightarrow Q \longrightarrow R) \longleftrightarrow (P \wedge Q \longrightarrow R)$

$PROTECT \text{ (} P \wedge Q) \equiv PROTECT \text{ } P \wedge PROTECT \text{ } Q$

$(P \wedge Q) \wedge R \equiv P \wedge Q \wedge R$

$(a,c) \in Rel \wedge PROTECT \text{ } P \longleftrightarrow PROTECT \text{ } P \wedge (a,c) \in Rel$

$\langle proof \rangle$

**lemma** *import-param-2*:

$Trueprop \text{ (} PROTECT \text{ } P \wedge Q \longrightarrow R) \equiv (P \Longrightarrow Q \longrightarrow R)$

$\langle proof \rangle$

**lemma** *import-param-3*:

$\uparrow(P \wedge Q) = \uparrow P * \uparrow Q$

$\uparrow((c,a) \in R) = hn\text{-val } R \text{ } a \text{ } c$

$\langle proof \rangle$

**named-theorems-rev** *sepref-import-rewrite*  $\langle Rewrite \text{ rules on importing parametricity theorems} \rangle$

**lemma** *to-import-frefD*:

**assumes**  $(f,g) \in fref \text{ } P \text{ } R \text{ } S$

**shows**  $\llbracket PROTECT \text{ (} P \text{ } y); (x,y) \in R \rrbracket \Longrightarrow (f \text{ } x, g \text{ } y) \in S$

$\langle proof \rangle$

**lemma** *add-PR-CONST*:  $(c,a) \in R \Longrightarrow (c, PR\text{-CONST } a) \in R$   $\langle proof \rangle$

$\langle ML \rangle$

## 1.7.2 Purity

**definition** *import-rel1*  $R \equiv \lambda A \text{ } c \text{ } ci. \uparrow(is\text{-pure } A \wedge (ci,c) \in \langle the\text{-pure } A \rangle R)$

**definition** *import-rel2*  $R \equiv \lambda A \text{ } B \text{ } c \text{ } ci. \uparrow(is\text{-pure } A \wedge is\text{-pure } B \wedge (ci,c) \in \langle the\text{-pure } A, the\text{-pure } B \rangle R)$

**lemma** *import-rel1-pure-conv*:  $import\text{-rel1 } R \text{ (pure } A) = pure \text{ (}\langle A \rangle R)$

$\langle proof \rangle$

**lemma** *import-rel2-pure-conv*:  $import\text{-rel2 } R \text{ (pure } A) \text{ (pure } B) = pure \text{ (}\langle A, B \rangle R)$

*<proof>*

**lemma** *precise-pure*[*constraint-rules*]: *single-valued R*  $\implies$  *precise (pure R)*  
*<proof>*

**lemma** *precise-pure-iff-sv*: *precise (pure R)*  $\longleftrightarrow$  *single-valued R*  
*<proof>*

**lemma** *pure-precise-iff-sv*:  $\llbracket$ *is-pure R* $\rrbracket$   
 $\implies$  *precise R*  $\longleftrightarrow$  *single-valued (the-pure R)*  
*<proof>*

**lemmas** [*safe-constraint-rules*] = *single-valued-Id br-sv*

**end**

## 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**  $\llbracket$ *cd-patterns hn-refine - ?f - -* $\rrbracket$

**lemma** *heap-fixp-codegen*:  
  **assumes** *DEF*:  $f \equiv \text{heap.fixp-fun } cB$   
  **assumes** *M*:  $(\bigwedge x. \text{mono-Heap } (\lambda f. cB f x))$   
  **shows**  $f x = cB f x$   
  *<proof>*

*<ML>*

### 1.8.2 Synthesis setup for sepref-definition goals

**consts** *UNSPEC*::*'a*

**abbreviation** *hfunspec*  
  ::  $('a \Rightarrow 'b \Rightarrow \text{assn}) \Rightarrow ('a \Rightarrow 'b \Rightarrow \text{assn}) \times ('a \Rightarrow 'b \Rightarrow \text{assn})$   
   $\langle (-^?) \rangle$  [1000] 999  
  **where**  $R^? \equiv \text{hf-pres } R \text{ UNSPEC}$

**definition** *SYNTH* :: ('a ⇒ 'r nres) ⇒ (('ai ⇒ 'ri Heap) × ('a ⇒ 'r nres)) set  
 ⇒ bool

where *SYNTH* f R ≡ True

**definition** [simp]: CP-UNCURRY - - ≡ True

**definition** [simp]: INTRO-KD - - ≡ True

**definition** [simp]: SPEC-RES-ASSN - - ≡ True

**lemma** [synth-rules]: CP-UNCURRY f g ⟨proof⟩

**lemma** [synth-rules]: CP-UNCURRY (uncurry0 f) (uncurry0 g) ⟨proof⟩

**lemma** [synth-rules]: CP-UNCURRY f g ⇒ CP-UNCURRY (uncurry f) (uncurry g) ⟨proof⟩

**lemma** [synth-rules]: [[INTRO-KD R1 R1'; INTRO-KD R2 R2']] ⇒ INTRO-KD (R1 \*\_a R2) (R1' \*\_a R2') ⟨proof⟩

**lemma** [synth-rules]: INTRO-KD (R<sup>?</sup>) (hf-pres R k) ⟨proof⟩

**lemma** [synth-rules]: INTRO-KD (R<sup>k</sup>) (R<sup>k</sup>) ⟨proof⟩

**lemma** [synth-rules]: INTRO-KD (R<sup>d</sup>) (R<sup>d</sup>) ⟨proof⟩

**lemma** [synth-rules]: SPEC-RES-ASSN R R ⟨proof⟩

**lemma** [synth-rules]: SPEC-RES-ASSN UNSPEC R ⟨proof⟩

**lemma** synth-hnrI:

[[CP-UNCURRY fi f; INTRO-KD R R'; SPEC-RES-ASSN S S']] ⇒ SYNTH-TERM  
 (SYNTH f ([P]<sub>a</sub> R → S)) ((fi, SDUMMY) ∈ SDUMMY, (fi, f) ∈ ([P]<sub>a</sub> R' → S'))  
 ⟨proof⟩

**term** starts-with

⟨ML⟩

end

## 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) *<proof>*  
**declare** *intf-of-rel*[*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)  $\implies$  *INTF-OF-REL* (*<R>**option-rel*) *TYPE*('a *option*)  
*INTF-OF-REL* *R* *TYPE*('a)  $\implies$  *INTF-OF-REL* (*<R>**list-rel*) *TYPE*('a *list*)  
*INTF-OF-REL* *R* *TYPE*('a)  $\implies$  *INTF-OF-REL* (*<R>**nres-rel*) *TYPE*('a *nres*)  
 $\llbracket$ *INTF-OF-REL* *R* *TYPE*('a); *INTF-OF-REL* *S* *TYPE*('b) $\rrbracket \implies$  *INTF-OF-REL* (*R*×*r**S*) *TYPE*('a×'b)  
 $\llbracket$ *INTF-OF-REL* *R* *TYPE*('a); *INTF-OF-REL* *S* *TYPE*('b) $\rrbracket \implies$  *INTF-OF-REL* (*<R,S>**sum-rel*) *TYPE*('a+'b)  
 $\llbracket$ *INTF-OF-REL* *R* *TYPE*('a); *INTF-OF-REL* *S* *TYPE*('b) $\rrbracket \implies$  *INTF-OF-REL* (*R*→*S*) *TYPE*('a⇒'b)  
*<proof>*

**lemma** *synth-intf-of-relI*: *INTF-OF-REL* *R* *I*  $\implies$  *SYNTH-TERM* *R* *I* *<proof>*

## 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* ≡ λ*x*. *ASSERT* (*P* *x*)  $\gg$  *f* *x*

**lemma** *param-op-mop-iff*:  
**assumes** (*Q,P*)∈*R*→*bool-rel*  
**shows**  
(*f, g*) ∈ [*P*]<sub>*f*</sub> *R* → *<S>**nres-rel*  
 $\longleftrightarrow$   
(*mop* *Q* *f, mop* *P* *g*) ∈ *R* →<sub>*f*</sub> *<S>**nres-rel*  
*<proof>*

**lemma** *param-mopI*:  
**assumes** (*f,g*) ∈ [*P*]<sub>*f*</sub> *R* → *<S>**nres-rel*  
**assumes** (*Q,P*) ∈ *R* → *bool-rel*  
**shows** (*mop* *Q* *f, mop* *P* *g*) ∈ *R* →<sub>*f*</sub> *<S>**nres-rel*  
*<proof>*

**lemma** *mop-spec-rl*: *P* *x*  $\implies$  *mop* *P* *f* *x* ≤ *f* *x* *<proof>*

**lemma** *mop-spec-rl-from-def*:  
**assumes** *f* ≡ *mop* *P* *g*  
**assumes** *P* *x*

**assumes**  $g\ x \leq z$   
**shows**  $f\ x \leq z$   
 $\langle$ *proof* $\rangle$

**lemma** *mop-leof-rl-from-def*:  
**assumes**  $f \equiv \text{mop } P\ g$   
**assumes**  $P\ x \implies g\ x \leq_n z$   
**shows**  $f\ x \leq_n z$   
 $\langle$ *proof* $\rangle$

**lemma** *assert-true-bind-conv*:  $\text{ASSERT } \text{True} \gg m = m\ \langle$ *proof* $\rangle$

**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*:  $\llbracket \text{PROP } P; \text{CONSTRAINT } \text{is-pure } A \rrbracket \implies \text{PROP } P\ \langle$ *proof* $\rangle$

**lemma** *sepref-relpropI*:  $P\ R = \text{CONSTRAINT } P\ R\ \langle$ *proof* $\rangle$

#### Purity

**lemmas** [*constraint-simps*] = *the-pure-pure*

**definition** [*constraint-abbrevs*]:  $\text{IS-PURE } P\ R \equiv \text{is-pure } R \wedge P\ (\text{the-pure } R)$

**lemma** *IS-PURE-pureI*:

$P\ R \implies \text{IS-PURE } P\ (\text{pure } R)$   
 $\langle$ *proof* $\rangle$

**lemma** [*fcomp-norm-simps*]:  $\text{CONSTRAINT } (\text{IS-PURE } \Phi)\ P \implies \text{pure } (\text{the-pure } P) = P$

$\langle$ *proof* $\rangle$

**lemma** [*fcomp-norm-simps*]:  $\text{CONSTRAINT } (\text{IS-PURE } P)\ A \implies P\ (\text{the-pure } A)$

$\langle$ *proof* $\rangle$

**lemma** *handle-purity1*:

$\text{CONSTRAINT } (\text{IS-PURE } \Phi)\ A \implies \text{CONSTRAINT } \Phi\ (\text{the-pure } A)$

$\langle$ *proof* $\rangle$

**lemma** *handle-purity2*:

$\text{CONSTRAINT } (\text{IS-PURE } \Phi)\ A \implies \text{CONSTRAINT } \text{is-pure } A$

$\langle$ *proof* $\rangle$

### 1.9.4 Composition

#### Preconditions

**definition** [*simp*]:  $t\text{comp-pre } Q\ T\ P \equiv \lambda a. Q\ a \wedge (\forall a'. (a', a) \in T \longrightarrow P\ a')$

**definition** *and-pre*  $P1\ P2 \equiv \lambda x. P1\ x \wedge P2\ x$

**definition** *imp-pre*  $P1\ P2 \equiv \lambda x. P1\ x \longrightarrow P2\ x$

**lemma** *and-pre-beta*:  $PP \longrightarrow P\ x \wedge Q\ x \Longrightarrow PP \longrightarrow \text{and-pre}\ P\ Q\ x$  *<proof>*

**lemma** *imp-pre-beta*:  $PP \longrightarrow P\ x \longrightarrow Q\ x \Longrightarrow PP \longrightarrow \text{imp-pre}\ P\ Q\ x$  *<proof>*

**definition** *IMP-PRE*  $P1\ P2 \equiv \forall x. P1\ x \longrightarrow P2\ x$

**lemma** *IMP-PRED*:  $\text{IMP-PRE}\ P1\ P2 \Longrightarrow P1\ x \Longrightarrow P2\ x$  *<proof>*

**lemma** *IMP-PRE-refl*:  $\text{IMP-PRE}\ P\ P$  *<proof>*

**definition** *IMP-PRE-CUSTOM*  $\equiv \text{IMP-PRE}$

**lemma** *IMP-PRE-CUSTOMD*:  $\text{IMP-PRE-CUSTOM}\ P1\ P2 \Longrightarrow \text{IMP-PRE}\ P1\ P2$  *<proof>*

**lemma** *IMP-PRE-CUSTOMI*:  $[\wedge x. P1\ x \Longrightarrow P2\ x] \Longrightarrow \text{IMP-PRE-CUSTOM}\ P1\ P2$  *<proof>*

**lemma** *imp-and-triv-pre*:  $\text{IMP-PRE}\ P\ (\text{and-pre}\ (\lambda-. \text{True})\ P)$  *<proof>*

## Premises

**definition** *ALL-LIST*  $A \equiv (\forall x \in \text{set}\ A. x)$

**definition** *IMP-LIST*  $A\ B \equiv \text{ALL-LIST}\ A \longrightarrow B$

**lemma** *to-IMP-LISTI*:  
 $P \Longrightarrow \text{IMP-LIST}\ []\ P$   
*<proof>*

**lemma** *to-IMP-LIST*:  $(P \Longrightarrow \text{IMP-LIST}\ Ps\ Q) \equiv \text{Trueprop}\ (\text{IMP-LIST}\ (P\#\Ps)\ Q)$   
*<proof>*

**lemma** *from-IMP-LIST*:  
 $\text{Trueprop}\ (\text{IMP-LIST}\ As\ B) \equiv (\text{ALL-LIST}\ As \Longrightarrow B)$   
 $(\text{ALL-LIST}\ [] \Longrightarrow B) \equiv \text{Trueprop}\ B$   
 $(\text{ALL-LIST}\ (A\#\As) \Longrightarrow B) \equiv (A \Longrightarrow \text{ALL-LIST}\ As \Longrightarrow B)$   
*<proof>*

**lemma** *IMP-LIST-trivial*:  $\text{IMP-LIST}\ A\ B \Longrightarrow \text{IMP-LIST}\ A\ B$  *<proof>*

## Composition Rules

**lemma** *hfcomp-tcomp-pre*:  
**assumes**  $B: (g,h) \in [Q]_f\ T \rightarrow \langle U \rangle \text{nres-rel}$   
**assumes**  $A: (f,g) \in [P]_a\ RR' \rightarrow S$   
**shows**  $(f,h) \in [tcomp-pre\ Q\ T\ P]_a\ \text{hrp-comp}\ RR'\ T \rightarrow \text{hr-comp}\ S\ U$

$\langle proof \rangle$

**lemma** *transform-pre-param:*

**assumes**  $A: IMP-LIST\ Cns\ ((f, h) \in [tcomp-pre\ Q\ T\ P]_a\ hrp-comp\ RR'\ T \rightarrow hr-comp\ S\ U)$

**assumes**  $P: IMP-LIST\ Cns\ ((P, P') \in T \rightarrow bool-rel)$

**assumes**  $C: IMP-PRE\ PP'\ (and-pre\ P'\ Q)$

**shows**  $IMP-LIST\ Cns\ ((f, h) \in [PP']_a\ hrp-comp\ RR'\ T \rightarrow hr-comp\ S\ U)$

$\langle proof \rangle$

**lemma** *hfref-mop-conv:*  $((g, mop\ P\ f) \in [Q]_a\ R \rightarrow S) \longleftrightarrow (g, f) \in [\lambda x. P\ x \wedge Q\ x]_a\ R \rightarrow S$

$\langle proof \rangle$

**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**  $(impl, mf) \in [PP']_a\ R \rightarrow S$

$\langle proof \rangle$

**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**  $(impl, f) \in [PP']_a\ R \rightarrow S$

$\langle proof \rangle$

## Precondition Simplification

**lemma** *IMP-PRE-eqI:*

**assumes**  $\bigwedge x. P\ x \longrightarrow Q\ x$

**assumes**  $CNV\ P\ P'$

**shows**  $IMP-PRE\ P'\ Q$

$\langle proof \rangle$

**lemma** *simp-and1:*

**assumes**  $Q \implies CNV\ P\ P'$

**assumes**  $PP \longrightarrow P' \wedge Q$

**shows**  $PP \longrightarrow P \wedge Q$

$\langle proof \rangle$

**lemma** *simp-and2:*

**assumes**  $P \implies CNV\ Q\ Q'$

**assumes**  $PP \longrightarrow P \wedge Q'$

**shows**  $PP \longrightarrow P \wedge Q$

$\langle proof \rangle$

**lemma** *triv-and1:*  $Q \longrightarrow True \wedge Q$   $\langle proof \rangle$



**lemma** *simp-imp*:  
**assumes**  $P \implies CNV\ Q\ Q'$   
**assumes**  $PP \longrightarrow Q'$   
**shows**  $PP \longrightarrow (P \longrightarrow Q)$   
 $\langle proof \rangle$

**lemma** *CNV-split*:  
**assumes**  $CNV\ A\ A'$   
**assumes**  $CNV\ B\ B'$   
**shows**  $CNV\ (A \wedge B)\ (A' \wedge B')$   
 $\langle proof \rangle$

**lemma** *CNV-prove*:  
**assumes**  $P$   
**shows**  $CNV\ P\ True$   
 $\langle proof \rangle$

**lemma** *simp-pre-final-simp*:  
**assumes**  $CNV\ P\ P'$   
**shows**  $P' \longrightarrow P$   
 $\langle proof \rangle$

**lemma** *auto-weaken-pre-uncurry-step'*:  
**assumes**  $PROTECT\ f\ a \equiv f'$   
**shows**  $PROTECT\ (uncurry\ f)\ (a,b) \equiv f'\ b$   
 $\langle proof \rangle$

### 1.9.5 Protected Constants

**lemma** *add-PR-CONST-to-def*:  $x \equiv y \implies PR-CONST\ x \equiv y$   $\langle proof \rangle$

### 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*  $\langle Sepref: Simp - rules\ to\ transform\ re-  
relator\ properties \rangle$

**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

$\langle ML \rangle$

## 1.9.8 Obsolete Manual Specification Helpers

**lemma** *vcg-of-RETURN-np*:

**assumes**  $f \equiv \text{RETURN } r$   
**shows**  $\text{SPEC } (\lambda x. x=r) \leq m \implies f \leq m$   
**and**  $\text{SPEC } (\lambda x. x=r) \leq_n m \implies f \leq_n m$   
 $\langle \text{proof} \rangle$

**lemma** *vcg-of-RETURN*:

**assumes**  $f \equiv \text{do } \{ \text{ASSERT } \Phi; \text{RETURN } r \}$   
**shows**  $\llbracket \Phi; \text{SPEC } (\lambda x. x=r) \leq m \rrbracket \implies f \leq m$   
**and**  $\llbracket \Phi \implies \text{SPEC } (\lambda x. x=r) \leq_n m \rrbracket \implies f \leq_n m$   
 $\langle \text{proof} \rangle$

**lemma** *vcg-of-SPEC*:

**assumes**  $f \equiv \text{do } \{ \text{ASSERT } \text{pre}; \text{SPEC } \text{post} \}$   
**shows**  $\llbracket \text{pre}; \text{SPEC } \text{post} \leq m \rrbracket \implies f \leq m$   
**and**  $\llbracket \text{pre} \implies \text{SPEC } \text{post} \leq_n m \rrbracket \implies f \leq_n m$   
 $\langle \text{proof} \rangle$

**lemma** *vcg-of-SPEC-np*:

**assumes**  $f \equiv \text{SPEC } \text{post}$   
**shows**  $\text{SPEC } \text{post} \leq m \implies f \leq m$   
**and**  $\text{SPEC } \text{post} \leq_n m \implies f \leq_n m$   
 $\langle \text{proof} \rangle$

**lemma** *mk-mop-rl1*:

**assumes**  $\bigwedge x. mf\ x \equiv \text{ASSERT } (P\ x) \gg \text{RETURN } (f\ x)$   
**shows**  $(\text{RETURN } o\ f, mf) \in \text{Id} \rightarrow \langle \text{Id} \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**lemma** *mk-mop-rl2*:

**assumes**  $\bigwedge x\ y. mf\ x\ y \equiv \text{ASSERT } (P\ x\ y) \gg \text{RETURN } (f\ x\ y)$   
**shows**  $(\text{RETURN } oo\ f, mf) \in \text{Id} \rightarrow \text{Id} \rightarrow \langle \text{Id} \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**lemma** *mk-mop-rl3*:

**assumes**  $\bigwedge x\ y\ z. mf\ x\ y\ z \equiv \text{ASSERT } (P\ x\ y\ z) \gg \text{RETURN } (f\ x\ y\ z)$   
**shows**  $(\text{RETURN } ooo\ f, mf) \in \text{Id} \rightarrow \text{Id} \rightarrow \text{Id} \rightarrow \langle \text{Id} \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**lemma** *mk-mop-rl0-np*:

**assumes**  $mf \equiv \text{RETURN } f$   
**shows**  $(\text{RETURN } f, mf) \in \langle \text{Id} \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**lemma** *mk-mop-rl1-np*:

**assumes**  $\bigwedge x. mf\ x \equiv RETURN\ (f\ x)$   
**shows**  $(RETURN\ o\ f, mf) \in Id \rightarrow \langle Id \rangle nres\text{-}rel$   
*<proof>*

**lemma** *mk-mop-rl2-np*:

**assumes**  $\bigwedge x\ y. mf\ x\ y \equiv RETURN\ (f\ x\ y)$   
**shows**  $(RETURN\ oo\ f, mf) \in Id \rightarrow Id \rightarrow \langle Id \rangle nres\text{-}rel$   
*<proof>*

**lemma** *mk-mop-rl3-np*:

**assumes**  $\bigwedge x\ y\ z. mf\ x\ y\ z \equiv RETURN\ (f\ x\ y\ z)$   
**shows**  $(RETURN\ ooo\ f, mf) \in Id \rightarrow Id \rightarrow Id \rightarrow \langle Id \rangle nres\text{-}rel$   
*<proof>*

**lemma** *mk-op-rl0-np*:

**assumes**  $mf \equiv RETURN\ f$   
**shows**  $(uncurry0\ mf, uncurry0\ (RETURN\ f)) \in unit\text{-}rel \rightarrow_f \langle Id \rangle nres\text{-}rel$   
*<proof>*

**lemma** *mk-op-rl1*:

**assumes**  $\bigwedge x. mf\ x \equiv ASSERT\ (P\ x) \gg RETURN\ (f\ x)$   
**shows**  $(mf, RETURN\ o\ f) \in [P]_f\ Id \rightarrow \langle Id \rangle nres\text{-}rel$   
*<proof>*

**lemma** *mk-op-rl1-np*:

**assumes**  $\bigwedge x. mf\ x \equiv RETURN\ (f\ x)$   
**shows**  $(mf, (RETURN\ o\ f)) \in Id \rightarrow_f \langle Id \rangle nres\text{-}rel$   
*<proof>*

**lemma** *mk-op-rl2*:

**assumes**  $\bigwedge x\ y. mf\ x\ y \equiv ASSERT\ (P\ x\ y) \gg RETURN\ (f\ x\ y)$   
**shows**  $(uncurry\ mf, uncurry\ (RETURN\ oo\ f)) \in [uncurry\ P]_f\ Id \times_r Id \rightarrow$   
 $\langle Id \rangle nres\text{-}rel$   
*<proof>*

**lemma** *mk-op-rl2-np*:

**assumes**  $\bigwedge x\ y. mf\ x\ y \equiv RETURN\ (f\ x\ y)$   
**shows**  $(uncurry\ mf, uncurry\ (RETURN\ oo\ f)) \in Id \times_r Id \rightarrow_f \langle Id \rangle nres\text{-}rel$   
*<proof>*

**lemma** *mk-op-rl3*:

**assumes**  $\bigwedge x\ y\ z. mf\ x\ y\ z \equiv ASSERT\ (P\ x\ y\ z) \gg RETURN\ (f\ x\ y\ z)$   
**shows**  $(uncurry2\ mf, uncurry2\ (RETURN\ ooo\ f)) \in [uncurry2\ P]_f\ (Id \times_r Id) \times_r Id$   
 $\rightarrow \langle Id \rangle nres\text{-}rel$   
*<proof>*

**lemma** *mk-op-rl3-np*:  
**assumes**  $\bigwedge x y z. mf\ x\ y\ z \equiv RETURN\ (f\ x\ y\ z)$   
**shows**  $(uncurry2\ mf, uncurry2\ (RETURN\ ooo\ f)) \in (Id \times_r Id) \times_r Id \rightarrow_f \langle Id \rangle nres\text{-}rel$   
 $\langle proof \rangle$

**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**  $hn\text{-}refine\ \Gamma\ c\ \Gamma'\ R\ a$   
**assumes**  $\Gamma' \Longrightarrow_t \Gamma c'$   
**assumes**  $\bigwedge a\ c. hn\text{-}ctxt\ R\ a\ c \Longrightarrow_t hn\text{-}ctxt\ Rc\ a\ c$   
**shows**  $hn\text{-}refine\ \Gamma\ c\ \Gamma'\ Rc\ a$   
 $\langle proof \rangle$

**lemma** *ID-init*:  $\llbracket ID\ a\ a'\ TYPE('T); hn\text{-}refine\ \Gamma\ c\ \Gamma'\ R\ a \rrbracket$   
 $\Longrightarrow hn\text{-}refine\ \Gamma\ c\ \Gamma'\ R\ a \langle proof \rangle$

**lemma** *TRANS-init*:  $\llbracket hn\text{-}refine\ \Gamma\ c\ \Gamma'\ R\ a; CNV\ c\ c' \rrbracket$   
 $\Longrightarrow hn\text{-}refine\ \Gamma\ c'\ \Gamma'\ R\ a$   
 $\langle proof \rangle$

**lemma** *infer-post-triv*:  $P \Longrightarrow_t P \langle proof \rangle$

$\langle ML \rangle$

### Default Optimizer Setup

**lemma** *return-bind-eq-let*:  $do\ \{ x \leftarrow return\ v; f\ x \} = do\ \{ let\ x=v; f\ x \} \langle proof \rangle$   
**lemmas**  $[sepref\text{-}opt\text{-}simps] = return\text{-}bind\text{-}eq\text{-}let\ bind\text{-}return\ bind\text{-}bind\ id\text{-}def$

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**  $case-prod-return-opt[sepref-opt-simps]$ :  
 $case-prod (\lambda a b. return (f a b)) p = return (case-prod f p)$   
 $\langle proof \rangle$

**lemma**  $case-option-return-opt[sepref-opt-simps]$ :  
 $case-option (return fn) (\lambda s. return (fs s)) v = return (case-option fn fs v)$   
 $\langle proof \rangle$

**lemma**  $case-list-return[sepref-opt-simps]$ :  
 $case-list (return fn) (\lambda x xs. return (fc x xs)) l = return (case-list fn fc l)$   
 $\langle proof \rangle$

**lemma**  $if-return[sepref-opt-simps]$ :  
 $If b (return t) (return e) = return (If b t e) \langle proof \rangle$

In some cases, pushing in the returns is more convenient

**lemma**  $case-prod-opt2[sepref-opt-simps2]$ :  
 $(\lambda x. return (case x of (a,b) \Rightarrow f a b))$   
 $= (\lambda(a,b). return (f a b))$   
 $\langle proof \rangle$

## 1.10.2 Debugging Methods

$\langle ML \rangle$

## 1.10.3 Utilities

### Manual href-proofs

$\langle ML \rangle$

### Copying of Parameters

**lemma**  $fold-COPY: x = COPY x \langle proof \rangle$

**sepref-register**  $COPY$

Copy is treated as normal operator, and one can just declare rules for it!

**lemma**  $hnr-pure-COPY[sepref-fr-rules]$ :  
 $CONSTRAINT is-pure R \implies (return, RETURN o COPY) \in R^k \rightarrow_a R$   
 $\langle proof \rangle$

## 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*:

$(a \wedge b) \longleftrightarrow (if\ a\ then\ b\ else\ False)$

$(a \vee b) \longleftrightarrow (if\ a\ then\ True\ else\ b)$

$(a \longrightarrow b) \longleftrightarrow (if\ a\ then\ b\ else\ True)$

$\langle proof \rangle$

## Eliminating higher-order

**lemma** *ho-prod-move*[*sepref-preproc*]:  $case\text{-}prod\ (\lambda a\ b\ x.\ f\ x\ a\ b) = (\lambda p\ x.\ case\text{-}prod\ (f\ x)\ p)$

$\langle proof \rangle$

**declare** *o-apply*[*sepref-preproc*]

## Precision Proofs

We provide a method that tries to extract equalities from an assumption of the form  $- \models P1 * \dots * Pn \wedge_A P1' * \dots * Pn'$ , if it find a precision rule for  $Pi$  and  $Pi'$ . 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*  $h\ \Gamma\ \Gamma' \equiv h \models \Gamma * true \wedge_A \Gamma' * true$

**lemma** *prec-specI*:  $h \models \Gamma \wedge_A \Gamma' \implies prec\text{-}spec\ h\ \Gamma\ \Gamma'$

$\langle proof \rangle$

**lemma** *prec-split1-aux*:  $A * B * true \implies_A A * true$

$\langle proof \rangle$

**lemma** *prec-split2-aux*:  $A * B * true \implies_A B * true$

$\langle proof \rangle$

**lemma** *prec-spec-splitE*:

**assumes** *prec-spec*  $h\ (A * B)\ (C * D)$

**obtains** *prec-spec*  $h\ A\ C\ prec\text{-}spec\ h\ B\ D$

$\langle proof \rangle$

**lemma** *prec-specD*:

**assumes** *precise*  $R$

**assumes** *prec-spec*  $h\ (R\ a\ p)\ (R\ a'\ p)$

**shows**  $a = a'$

$\langle proof \rangle$

$\langle ML \rangle$

### Combinator Rules

**lemma** *split-merge*:  $\llbracket A \vee_A B \Longrightarrow_t X; X \vee_A C \Longrightarrow_t D \rrbracket \Longrightarrow (A \vee_A B \vee_A C \Longrightarrow_t D)$   
*<proof>*

*<ML>*

**end**

# 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  $\Rightarrow$  'ai  $\Rightarrow$  assn)  $\Rightarrow$  'a  $\Rightarrow$  'a where [simp]: ASSN-ANNOT
A x  $\equiv$  x
context fixes A :: 'a  $\Rightarrow$  'ai  $\Rightarrow$  assn begin
  sepref-register PR-CONST (ASSN-ANNOT A)
  lemma [def-pat-rules]: ASSN-ANNOT$A  $\equiv$  UNPROTECT (ASSN-ANNOT A)
  <proof>
  lemma [sepref-fr-rules]: (return o ( $\lambda$ x. x), RETURN o PR-CONST (ASSN-ANNOT
A))  $\in$  Ad $\rightarrow$ aA
  <proof>
end
```

```
lemma annotate-assn: x  $\equiv$  ASSN-ANNOT A x <proof>
```

#### 2.1.2 Shortcuts

```
abbreviation (input) nat-assn  $\equiv$  (id-assn::nat  $\Rightarrow$  -)
abbreviation (input) int-assn  $\equiv$  (id-assn::int  $\Rightarrow$  -)
abbreviation (input) bool-assn  $\equiv$  (id-assn::bool  $\Rightarrow$  -)
```



### 2.1.3 Identity Relations

**definition**  $IS-ID\ R \equiv R=Id$

**definition**  $IS-BELOW-ID\ R \equiv R \subseteq Id$

**lemma**  $[safe-constraint-rules]$ :

$IS-ID\ Id$

$IS-ID\ R1 \implies IS-ID\ R2 \implies IS-ID\ (R1 \rightarrow R2)$

$IS-ID\ R \implies IS-ID\ (\langle R \rangle option-rel)$

$IS-ID\ R \implies IS-ID\ (\langle R \rangle list-rel)$

$IS-ID\ R1 \implies IS-ID\ R2 \implies IS-ID\ (R1 \times_r R2)$

$IS-ID\ R1 \implies IS-ID\ R2 \implies IS-ID\ (\langle R1, R2 \rangle sum-rel)$

$\langle proof \rangle$

**lemma**  $[safe-constraint-rules]$ :

$IS-BELOW-ID\ Id$

$IS-BELOW-ID\ R \implies IS-BELOW-ID\ (\langle R \rangle option-rel)$

$IS-BELOW-ID\ R1 \implies IS-BELOW-ID\ R2 \implies IS-BELOW-ID\ (R1 \times_r R2)$

$IS-BELOW-ID\ R1 \implies IS-BELOW-ID\ R2 \implies IS-BELOW-ID\ (\langle R1, R2 \rangle sum-rel)$

$\langle proof \rangle$

**lemma**  $IS-BELOW-ID-fun-rel-aux: R1 \supseteq Id \implies IS-BELOW-ID\ R2 \implies IS-BELOW-ID\ (R1 \rightarrow R2)$

$\langle proof \rangle$

**corollary**  $IS-BELOW-ID-fun-rel[safe-constraint-rules]$ :

$IS-ID\ R1 \implies IS-BELOW-ID\ R2 \implies IS-BELOW-ID\ (R1 \rightarrow R2)$

$\langle proof \rangle$

**lemma**  $IS-BELOW-ID-list-rel[safe-constraint-rules]$ :

$IS-BELOW-ID\ R \implies IS-BELOW-ID\ (\langle R \rangle list-rel)$

$\langle proof \rangle$

**lemma**  $IS-ID-imp-BELOW-ID[constraint-rules]$ :

$IS-ID\ R \implies IS-BELOW-ID\ R$

$\langle proof \rangle$

### 2.1.4 Inverse Relation

**lemma**  $inv-fun-rel-eq[simp]: (A \rightarrow B)^{-1} = A^{-1} \rightarrow B^{-1}$

$\langle proof \rangle$

**lemma**  $inv-option-rel-eq[simp]: (\langle K \rangle option-rel)^{-1} = \langle K^{-1} \rangle option-rel$

$\langle proof \rangle$

**lemma**  $inv-prod-rel-eq[simp]: (P \times_r Q)^{-1} = P^{-1} \times_r Q^{-1}$

$\langle proof \rangle$

**lemma**  $inv-sum-rel-eq[simp]: (\langle P, Q \rangle sum-rel)^{-1} = \langle P^{-1}, Q^{-1} \rangle sum-rel$

*<proof>*

**lemma** *inv-list-rel-eq[simp]*:  $(\langle R \rangle \text{list-rel})^{-1} = \langle R^{-1} \rangle \text{list-rel}$   
*<proof>*

**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*  $R \equiv \text{single-valued } (R^{-1})$

**definition** *IS-LEFT-TOTAL*  $R \equiv \text{Domain } R = \text{UNIV}$

**definition** *IS-RIGHT-TOTAL*  $R \equiv \text{Range } R = \text{UNIV}$

**abbreviation** (*input*) *IS-RIGHT-UNIQUE*  $\equiv \text{single-valued}$

**lemmas** *IS-RIGHT-UNIQUED* = *single-valuedD*

**lemma** *IS-LEFT-UNIQUED*:  $\llbracket \text{IS-LEFT-UNIQUE } r; (y, x) \in r; (z, x) \in r \rrbracket \implies y = z$   
*<proof>*

**lemma** *prop2p*:

*IS-LEFT-UNIQUE*  $R = \text{left-unique } (\text{rel2p } R)$   
*IS-RIGHT-UNIQUE*  $R = \text{right-unique } (\text{rel2p } R)$   
*right-unique*  $(\text{rel2p } (R^{-1})) = \text{left-unique } (\text{rel2p } R)$   
*IS-LEFT-TOTAL*  $R = \text{left-total } (\text{rel2p } R)$   
*IS-RIGHT-TOTAL*  $R = \text{right-total } (\text{rel2p } R)$   
*<proof>*

**lemma** *p2prop*:

*left-unique*  $P = \text{IS-LEFT-UNIQUE } (p2\text{rel } P)$   
*right-unique*  $P = \text{IS-RIGHT-UNIQUE } (p2\text{rel } P)$   
*left-total*  $P = \text{IS-LEFT-TOTAL } (p2\text{rel } P)$   
*right-total*  $P = \text{IS-RIGHT-TOTAL } (p2\text{rel } P)$   
*bi-unique*  $P \longleftrightarrow \text{left-unique } P \wedge \text{right-unique } P$   
*<proof>*

**lemmas** [*safe-constraint-rules*] =  
*single-valued-Id*  
*prod-rel-sv*  
*list-rel-sv*  
*option-rel-sv*  
*sum-rel-sv*

**lemma** [*safe-constraint-rules*]:

*IS-LEFT-UNIQUE Id*  
*IS-LEFT-UNIQUE R1*  $\implies$  *IS-LEFT-UNIQUE R2*  $\implies$  *IS-LEFT-UNIQUE (R1  $\times_r$  R2)*  
*IS-LEFT-UNIQUE R1*  $\implies$  *IS-LEFT-UNIQUE R2*  $\implies$  *IS-LEFT-UNIQUE ( $\langle R1, R2 \rangle$  sum-rel)*  
*IS-LEFT-UNIQUE R*  $\implies$  *IS-LEFT-UNIQUE ( $\langle R \rangle$  option-rel)*  
*IS-LEFT-UNIQUE R*  $\implies$  *IS-LEFT-UNIQUE ( $\langle R \rangle$  list-rel)*  
 <proof>

**lemma** *IS-LEFT-TOTAL-alt*: *IS-LEFT-TOTAL R*  $\longleftrightarrow$   $(\forall x. \exists y. (x,y) \in R)$   
 <proof>

**lemma** *IS-RIGHT-TOTAL-alt*: *IS-RIGHT-TOTAL R*  $\longleftrightarrow$   $(\forall x. \exists y. (y,x) \in R)$   
 <proof>

**lemma** [*safe-constraint-rules*]:

*IS-LEFT-TOTAL Id*  
*IS-LEFT-TOTAL R1*  $\implies$  *IS-LEFT-TOTAL R2*  $\implies$  *IS-LEFT-TOTAL (R1  $\times_r$  R2)*  
*IS-LEFT-TOTAL R1*  $\implies$  *IS-LEFT-TOTAL R2*  $\implies$  *IS-LEFT-TOTAL ( $\langle R1, R2 \rangle$  sum-rel)*  
*IS-LEFT-TOTAL R*  $\implies$  *IS-LEFT-TOTAL ( $\langle R \rangle$  option-rel)*  
 <proof>

**lemma** [*safe-constraint-rules*]: *IS-LEFT-TOTAL R*  $\implies$  *IS-LEFT-TOTAL ( $\langle R \rangle$  list-rel)*  
 <proof>

**lemma** [*safe-constraint-rules*]:

*IS-RIGHT-TOTAL Id*  
*IS-RIGHT-TOTAL R1*  $\implies$  *IS-RIGHT-TOTAL R2*  $\implies$  *IS-RIGHT-TOTAL (R1  $\times_r$  R2)*  
*IS-RIGHT-TOTAL R1*  $\implies$  *IS-RIGHT-TOTAL R2*  $\implies$  *IS-RIGHT-TOTAL ( $\langle R1, R2 \rangle$  sum-rel)*  
*IS-RIGHT-TOTAL R*  $\implies$  *IS-RIGHT-TOTAL ( $\langle R \rangle$  option-rel)*  
 <proof>

**lemma** [*safe-constraint-rules*]: *IS-RIGHT-TOTAL R*  $\implies$  *IS-RIGHT-TOTAL ( $\langle R \rangle$  list-rel)*  
 <proof>

**lemma** [*constraint-simps*]:

*IS-LEFT-TOTAL (R<sup>-1</sup>)*  $\longleftrightarrow$  *IS-RIGHT-TOTAL R*  
*IS-RIGHT-TOTAL (R<sup>-1</sup>)*  $\longleftrightarrow$  *IS-LEFT-TOTAL R*  
*IS-LEFT-UNIQUE (R<sup>-1</sup>)*  $\longleftrightarrow$  *IS-RIGHT-UNIQUE R*  
*IS-RIGHT-UNIQUE (R<sup>-1</sup>)*  $\longleftrightarrow$  *IS-LEFT-UNIQUE R*  
 <proof>

**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)*  
 <proof>

**lemma** [*constraint-rules*]:

*IS-BELOW-ID R*  $\implies$  *IS-RIGHT-UNIQUE R*

$IS-BELOW-ID R \implies IS-LEFT-UNIQUE R$   
 $IS-ID R \implies IS-RIGHT-TOTAL R$   
 $IS-ID R \implies IS-LEFT-TOTAL R$   
 ⟨proof⟩

**thm** *constraint-rules*

### Additional Parametricity Lemmas

**lemma** *param-distinct*[*param*]:  $\llbracket IS-LEFT-UNIQUE A; IS-RIGHT-UNIQUE A \rrbracket \implies$   
 $(distinct, distinct) \in \langle A \rangle list-rel \rightarrow bool-rel$   
 ⟨proof⟩

**lemma** *param-Image*[*param*]:  
**assumes**  $IS-LEFT-UNIQUE A$   $IS-RIGHT-UNIQUE A$   
**shows**  $((\cdot), (\cdot)) \in \langle A \times_r B \rangle set-rel \rightarrow \langle A \rangle set-rel \rightarrow \langle B \rangle set-rel$   
 ⟨proof⟩

**lemma** *pres-eq-iff-svb*:  $((=), (=)) \in K \rightarrow K \rightarrow bool-rel \iff (single-valued K \wedge single-valued (K^{-1}))$   
 ⟨proof⟩

**definition** *IS-PRES-EQ*  $R \equiv ((=), (=)) \in R \rightarrow R \rightarrow bool-rel$

**lemma** [*constraint-rules*]:  $\llbracket single-valued R; single-valued (R^{-1}) \rrbracket \implies IS-PRES-EQ R$   
 ⟨proof⟩

### 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*]:  $b-assn (pure R) P = pure (b-rel R P)$   
 ⟨proof⟩

**lemmas** [*sepref-import-rewrite*, *sepref-frame-normrel-eqs*, *fcomp-norm-unfold*]  
 $= b-assn-pure-conv[symmetric]$

**lemma** *b-rel-nesting*[*simp*]:  
 $b-rel (b-rel R P1) P2 = b-rel R (\lambda x. P1 x \wedge P2 x)$   
 ⟨proof⟩

**lemma** *b-rel-triv*[*simp*]:  
 $b-rel R (\lambda-. True) = R$   
 ⟨proof⟩

**lemma** *b-assn-nesting*[*simp*]:  
 $b-assn (b-assn A P1) P2 = b-assn A (\lambda x. P1 x \wedge P2 x)$   
 ⟨proof⟩

**lemma** *b-assn-triv*[*simp*]:  
 $b-assn A (\lambda-. True) = A$   
 ⟨proof⟩

**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*]:  $(x,y) \in b\text{-rel } R \ P \longleftrightarrow (x,y) \in R \wedge P \ y$   
 $\langle$ *proof* $\rangle$

**lemma** *b-assn-simp*[*simp*]:  $b\text{-assn } A \ P \ x \ y = A \ x \ y \ * \ \uparrow(P \ x)$   
 $\langle$ *proof* $\rangle$

**lemma** *b-rel-Range*[*simp*]:  $\text{Range } (b\text{-rel } R \ P) = \text{Range } R \cap \text{Collect } P$   $\langle$ *proof* $\rangle$

**lemma** *b-assn-rdom*[*simp*]:  $\text{rdomp } (b\text{-assn } R \ P) \ x \longleftrightarrow \text{rdomp } R \ x \wedge P \ x$   
 $\langle$ *proof* $\rangle$

**lemma** *b-rel-below-id*[*constraint-rules*]:  
 $IS\text{-BELOW-ID } R \Longrightarrow IS\text{-BELOW-ID } (b\text{-rel } R \ P)$   
 $\langle$ *proof* $\rangle$

**lemma** *b-rel-left-unique*[*constraint-rules*]:  
 $IS\text{-LEFT-UNIQUE } R \Longrightarrow IS\text{-LEFT-UNIQUE } (b\text{-rel } R \ P)$   
 $\langle$ *proof* $\rangle$

**lemma** *b-rel-right-unique*[*constraint-rules*]:  
 $IS\text{-RIGHT-UNIQUE } R \Longrightarrow IS\text{-RIGHT-UNIQUE } (b\text{-rel } R \ P)$   
 $\langle$ *proof* $\rangle$

**lemma** *b-assn-is-pure*[*safe-constraint-rules*]:  
 $is\text{-pure } A \Longrightarrow is\text{-pure } (b\text{-assn } A \ P)$   
 $\langle$ *proof* $\rangle$

**lemma** *b-assn-subtyping-match*[*sepref-frame-match-rules*]:  
**assumes**  $hn\text{-ctxt } (b\text{-assn } A \ P) \ x \ y \Longrightarrow_t hn\text{-ctxt } A' \ x \ y$   
**assumes**  $\llbracket vassn\text{-tag } (hn\text{-ctxt } A \ x \ y); vassn\text{-tag } (hn\text{-ctxt } A' \ x \ y); P \ x \rrbracket \Longrightarrow P' \ x$   
**shows**  $hn\text{-ctxt } (b\text{-assn } A \ P) \ x \ y \Longrightarrow_t hn\text{-ctxt } (b\text{-assn } A' \ P') \ x \ y$   
 $\langle$ *proof* $\rangle$

**lemma** *b-assn-subtyping-match-eqA*[*sepref-frame-match-rules*]:  
**assumes**  $\llbracket vassn\text{-tag } (hn\text{-ctxt } A \ x \ y); P \ x \rrbracket \Longrightarrow P' \ x$   
**shows**  $hn\text{-ctxt } (b\text{-assn } A \ P) \ x \ y \Longrightarrow_t hn\text{-ctxt } (b\text{-assn } A \ P') \ x \ y$   
 $\langle$ *proof* $\rangle$

**lemma** *b-assn-subtyping-match-tR*[*sepref-frame-match-rules*]:  
**assumes**  $\llbracket P \ x \rrbracket \Longrightarrow hn\text{-ctxt } A \ x \ y \Longrightarrow_t hn\text{-ctxt } A' \ x \ y$   
**shows**  $hn\text{-ctxt } (b\text{-assn } A \ P) \ x \ y \Longrightarrow_t hn\text{-ctxt } A' \ x \ y$   
 $\langle$ *proof* $\rangle$

**lemma** *b-assn-subtyping-match-tL*[*sepref-frame-match-rules*]:  
**assumes**  $hn\text{-ctxt } A \ x \ y \Longrightarrow_t hn\text{-ctxt } A' \ x \ y$   
**assumes**  $\llbracket vassn\text{-tag } (hn\text{-ctxt } A \ x \ y) \rrbracket \Longrightarrow P' \ x$   
**shows**  $hn\text{-ctxt } A \ x \ y \Longrightarrow_t hn\text{-ctxt } (b\text{-assn } A' \ P') \ x \ y$   
 $\langle$ *proof* $\rangle$

**lemma** *b-assn-subtyping-match-eqA-tR*[*sepref-frame-match-rules*]:

$hn\text{-}ctxt (b\text{-}assn A P) x y \Longrightarrow_t hn\text{-}ctxt A x y$   
(*proof*)

**lemma** *b-assn-subtyping-match-eqA-tL*[*sepref-frame-match-rules*]:

**assumes**  $\llbracket v\text{-}assn\text{-}tag (hn\text{-}ctxt A x y) \rrbracket \Longrightarrow P' x$   
**shows**  $hn\text{-}ctxt A x y \Longrightarrow_t hn\text{-}ctxt (b\text{-}assn A P') x y$   
(*proof*)

**lemma** *b-rel-subtyping-merge*[*sepref-frame-merge-rules*]:

**assumes**  $hn\text{-}ctxt A x y \vee_A hn\text{-}ctxt A' x y \Longrightarrow_t hn\text{-}ctxt Am x y$   
**shows**  $hn\text{-}ctxt (b\text{-}assn A P) x y \vee_A hn\text{-}ctxt (b\text{-}assn A' P') x y \Longrightarrow_t hn\text{-}ctxt$   
 $(b\text{-}assn Am (\lambda x. P x \vee P' x)) x y$   
(*proof*)

**lemma** *b-rel-subtyping-merge-eqA*[*sepref-frame-merge-rules*]:

**shows**  $hn\text{-}ctxt (b\text{-}assn A P) x y \vee_A hn\text{-}ctxt (b\text{-}assn A P') x y \Longrightarrow_t hn\text{-}ctxt (b\text{-}assn$   
 $A (\lambda x. P x \vee P' x)) x y$   
(*proof*)

**lemma** *b-rel-subtyping-merge-tL*[*sepref-frame-merge-rules*]:

**assumes**  $hn\text{-}ctxt A x y \vee_A hn\text{-}ctxt A' x y \Longrightarrow_t hn\text{-}ctxt Am x y$   
**shows**  $hn\text{-}ctxt A x y \vee_A hn\text{-}ctxt (b\text{-}assn A' P') x y \Longrightarrow_t hn\text{-}ctxt Am x y$   
(*proof*)

**lemma** *b-rel-subtyping-merge-tR*[*sepref-frame-merge-rules*]:

**assumes**  $hn\text{-}ctxt A x y \vee_A hn\text{-}ctxt A' x y \Longrightarrow_t hn\text{-}ctxt Am x y$   
**shows**  $hn\text{-}ctxt (b\text{-}assn A P) x y \vee_A hn\text{-}ctxt A' x y \Longrightarrow_t hn\text{-}ctxt Am x y$   
(*proof*)

**lemma** *b-rel-subtyping-merge-eqA-tL*[*sepref-frame-merge-rules*]:

**shows**  $hn\text{-}ctxt A x y \vee_A hn\text{-}ctxt (b\text{-}assn A P') x y \Longrightarrow_t hn\text{-}ctxt A x y$   
(*proof*)

**lemma** *b-rel-subtyping-merge-eqA-tR*[*sepref-frame-merge-rules*]:

**shows**  $hn\text{-}ctxt (b\text{-}assn A P) x y \vee_A hn\text{-}ctxt A x y \Longrightarrow_t hn\text{-}ctxt A x y$   
(*proof*)

**lemma** *b-assn-invalid-merge1*:  $hn\text{-}invalid (b\text{-}assn A P) x y \vee_A hn\text{-}invalid (b\text{-}assn$   
 $A P') x y$

$\Longrightarrow_t hn\text{-}invalid (b\text{-}assn A (\lambda x. P x \vee P' x)) x y$   
(*proof*)

**lemma** *b-assn-invalid-merge2*:  $hn\text{-}invalid (b\text{-}assn A P) x y \vee_A hn\text{-}invalid A x y$

$\Longrightarrow_t hn\text{-}invalid A x y$   
(*proof*)

**lemma** *b-assn-invalid-merge3*:  $hn\text{-}invalid A x y \vee_A hn\text{-}invalid (b\text{-}assn A P) x y$

$\Longrightarrow_t hn\text{-}invalid A x y$   
(*proof*)

**lemma** *b-assn-invalid-merge4*:  $hn\text{-invalid } (b\text{-assn } A \ P) \ x \ y \ \vee_A \ hn\text{-ctxt } (b\text{-assn } A \ P') \ x \ y$   
 $\implies_t \ hn\text{-invalid } (b\text{-assn } A \ (\lambda x. \ P \ x \ \vee \ P' \ x)) \ x \ y$   
 $\langle \text{proof} \rangle$

**lemma** *b-assn-invalid-merge5*:  $hn\text{-ctxt } (b\text{-assn } A \ P') \ x \ y \ \vee_A \ hn\text{-invalid } (b\text{-assn } A \ P) \ x \ y$   
 $\implies_t \ hn\text{-invalid } (b\text{-assn } A \ (\lambda x. \ P \ x \ \vee \ P' \ x)) \ x \ y$   
 $\langle \text{proof} \rangle$

**lemma** *b-assn-invalid-merge6*:  $hn\text{-invalid } (b\text{-assn } A \ P) \ x \ y \ \vee_A \ hn\text{-ctxt } A \ x \ y$   
 $\implies_t \ hn\text{-invalid } A \ x \ y$   
 $\langle \text{proof} \rangle$

**lemma** *b-assn-invalid-merge7*:  $hn\text{-ctxt } A \ x \ y \ \vee_A \ hn\text{-invalid } (b\text{-assn } A \ P) \ x \ y$   
 $\implies_t \ hn\text{-invalid } A \ x \ y$   
 $\langle \text{proof} \rangle$

**lemma** *b-assn-invalid-merge8*:  $hn\text{-ctxt } (b\text{-assn } A \ P) \ x \ y \ \vee_A \ hn\text{-invalid } A \ x \ y$   
 $\implies_t \ hn\text{-invalid } A \ x \ y$   
 $\langle \text{proof} \rangle$

**lemma** *b-assn-invalid-merge9*:  $hn\text{-invalid } A \ x \ y \ \vee_A \ hn\text{-ctxt } (b\text{-assn } A \ P) \ x \ y$   
 $\implies_t \ hn\text{-invalid } A \ x \ y$   
 $\langle \text{proof} \rangle$

**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* ::  $nat \Rightarrow (nat \times nat) \text{ set}$   
— Natural numbers with upper bound.  
**where** *nbn-rel*  $n \equiv b\text{-rel } nat\text{-rel } (\lambda x::nat. \ x < n)$

**abbreviation** *nbn-assn* ::  $nat \Rightarrow nat \Rightarrow nat \Rightarrow \text{assn}$   
— Natural numbers with upper bound.  
**where** *nbn-assn*  $n \equiv b\text{-assn } nat\text{-assn } (\lambda x::nat. \ x < n)$

## 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  $\Phi$  for  $\Phi$ ]  
 sepref-relpropI[of IS-ID]  
 sepref-relpropI[of IS-BELOW-ID]

**lemma** [sepref-relprops-simps]:  
 CONSTRAINT (IS-PURE IS-ID) A  $\implies$  CONSTRAINT (IS-PURE IS-BELOW-ID) A  
 CONSTRAINT (IS-PURE IS-ID) A  $\implies$  CONSTRAINT (IS-PURE IS-LEFT-TOTAL) A  
 CONSTRAINT (IS-PURE IS-ID) A  $\implies$  CONSTRAINT (IS-PURE IS-RIGHT-TOTAL) A  
 CONSTRAINT (IS-PURE IS-BELOW-ID) A  $\implies$  CONSTRAINT (IS-PURE IS-LEFT-UNIQUE) A  
 CONSTRAINT (IS-PURE IS-BELOW-ID) A  $\implies$  CONSTRAINT (IS-PURE IS-RIGHT-UNIQUE) A  
 <proof>

**declare** True-implies-equals[sepref-relprops-simps]

**lemma** [sepref-relprops-transform]: single-valued ( $R^{-1}$ ) = IS-LEFT-UNIQUE R  
 <proof>

## 2.1.8 HOL Combinators

**lemma** hn-if[sepref-comb-rules]:  
 assumes P:  $\Gamma \implies_t \Gamma 1 * \text{hn-val bool-rel } a \ a'$   
 assumes RT:  $a \implies \text{hn-refine } (\Gamma 1 * \text{hn-val bool-rel } a \ a') \ b' \ \Gamma 2b \ R \ b$   
 assumes RE:  $\neg a \implies \text{hn-refine } (\Gamma 1 * \text{hn-val bool-rel } a \ a') \ c' \ \Gamma 2c \ R \ c$   
 assumes IMP:  $\text{TERM If} \implies \Gamma 2b \vee_A \Gamma 2c \implies_t \Gamma'$   
 shows  $\text{hn-refine } \Gamma \ (\text{if } a' \ \text{then } b' \ \text{else } c') \ \Gamma' \ R \ (\text{If } \$a \$b \$c)$   
 <proof>

**lemmas** [sepref-opt-simps] = if-True if-False

**lemma** hn-let[sepref-comb-rules]:  
 assumes P:  $\Gamma \implies_t \Gamma 1 * \text{hn-ctxt } R \ v \ v'$   
 assumes R:  $\bigwedge x \ x'. \ x = v \implies \text{hn-refine } (\Gamma 1 * \text{hn-ctxt } R \ x \ x') \ (f' \ x')$   
 ( $\Gamma' \ x \ x') \ R 2 \ (f \ x)$   
 assumes F:  $\bigwedge x \ x'. \ \Gamma' \ x \ x' \implies_t \Gamma 2 * \text{hn-ctxt } R' \ x \ x'$   
 shows



*hn-refine*  $\Gamma$  (*Let*  $v' f'$ ) ( $\Gamma 2 * \text{hn-ctxt } R' v v'$ )  $R2$  (*Let*  $v (\lambda_2 x. f x)$ )  
 $\langle \text{proof} \rangle$

### 2.1.9 Basic HOL types

**lemma** *hnr-default*[*sepref-import-param*]: (*default, default*)  $\in Id$   $\langle \text{proof} \rangle$

**lemma** *unit-hnr*[*sepref-import-param*]: ( $()$ ,  $()$ )  $\in \text{unit-rel}$   $\langle \text{proof} \rangle$

**lemmas** [*sepref-import-param*] =  
*param-bool*  
*param-nat1*  
*param-int*

**lemmas** [*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** *param-min-nat*[*param, sepref-import-param*]: (*min, min*)  $\in \text{nat-rel} \rightarrow \text{nat-rel}$   
 $\rightarrow \text{nat-rel}$   $\langle \text{proof} \rangle$

**lemma** *param-max-nat*[*param, sepref-import-param*]: (*max, max*)  $\in \text{nat-rel} \rightarrow \text{nat-rel}$   
 $\rightarrow \text{nat-rel}$   $\langle \text{proof} \rangle$

**lemma** *param-min-int*[*param, sepref-import-param*]: (*min, min*)  $\in \text{int-rel} \rightarrow \text{int-rel}$   
 $\rightarrow \text{int-rel}$   $\langle \text{proof} \rangle$

**lemma** *param-max-int*[*param, sepref-import-param*]: (*max, max*)  $\in \text{int-rel} \rightarrow \text{int-rel}$   
 $\rightarrow \text{int-rel}$   $\langle \text{proof} \rangle$

**lemma** *uminus-hnr*[*sepref-import-param*]: (*uminus, uminus*)  $\in \text{int-rel} \rightarrow \text{int-rel}$   $\langle \text{proof} \rangle$

**lemma** *nat-param*[*param, sepref-import-param*]: (*nat, nat*)  $\in \text{int-rel} \rightarrow \text{nat-rel}$   $\langle \text{proof} \rangle$

**lemma** *int-param*[*param, sepref-import-param*]: (*int, int*)  $\in \text{nat-rel} \rightarrow \text{int-rel}$   $\langle \text{proof} \rangle$

### 2.1.10 Product

**lemmas** [*sepref-import-rewrite, sepref-frame-normrel-eqs, fcomp-norm-unfold*] =  
*prod-assn-pure-conv*[*symmetric*]

**lemma** *prod-assn-precise*[*constraint-rules*]:  
*precise P1  $\implies$  precise P2  $\implies$  precise (prod-assn P1 P2)*  
 $\langle \text{proof} \rangle$

**lemma**

*precise P1*  $\implies$  *precise P2*  $\implies$  *precise (prod-assn P1 P2)* — Original proof  
 ⟨proof⟩

**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)*  
 ⟨proof⟩

**lemma** *pure-prod*[*constraint-rules*]:  
**assumes** *P1: is-pure P1 and P2: is-pure P2*  
**shows** *is-pure (prod-assn P1 P2)*  
 ⟨proof⟩

**lemma** *prod-frame-match*[*sepref-frame-match-rules*]:  
**assumes** *hn-ctxt A (fst x) (fst y)  $\implies_t$  hn-ctxt A' (fst x) (fst y)*  
**assumes** *hn-ctxt B (snd x) (snd y)  $\implies_t$  hn-ctxt B' (snd x) (snd y)*  
**shows** *hn-ctxt (prod-assn A B) x y  $\implies_t$  hn-ctxt (prod-assn A' B') x y*  
 ⟨proof⟩

**lemma** *prod-frame-merge*[*sepref-frame-merge-rules*]:  
**assumes** *hn-ctxt A (fst x) (fst y)  $\vee_A$  hn-ctxt A' (fst x) (fst y)  $\implies_t$  hn-ctxt Am (fst x) (fst y)*  
**assumes** *hn-ctxt B (snd x) (snd y)  $\vee_A$  hn-ctxt B' (snd x) (snd y)  $\implies_t$  hn-ctxt Bm (snd x) (snd y)*  
**shows** *hn-ctxt (prod-assn A B) x y  $\vee_A$  hn-ctxt (prod-assn A' B') x y  $\implies_t$  hn-ctxt (prod-assn Am Bm) x y*  
 ⟨proof⟩

**lemma** *entt-invalid-prod*: *hn-invalid (prod-assn A B) p p'  $\implies_t$  hn-ctxt (prod-assn (invalid-assn A) (invalid-assn B)) p p'*  
 ⟨proof⟩

**lemmas** *invalid-prod-merge*[*sepref-frame-merge-rules*] = *gen-merge-cons*[*OF entt-invalid-prod*]

**lemma** *prod-assn-ctxt*: *prod-assn A1 A2 x y = z  $\implies$  hn-ctxt (prod-assn A1 A2) x y = z*  
 ⟨proof⟩

**lemma** *hn-case-prod'*[*sepref-prep-comb-rule,sepref-comb-rules*]:  
**assumes** *FR:  $\Gamma \implies_t$  hn-ctxt (prod-assn P1 P2) p' p \*  $\Gamma 1$*   
**assumes** *Pair:  $\bigwedge a1 a2 a1' a2'. \llbracket p' = (a1', a2') \rrbracket$*   
 $\implies$  *hn-refine (hn-ctxt P1 a1' a1 \* hn-ctxt P2 a2' a2 \*  $\Gamma 1$  \* 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 \*  $\Gamma 1'$ ) R (f' a1' a2')*  
**shows** *hn-refine  $\Gamma$  (case-prod f p) (hn-ctxt (prod-assn P1' P2') p' p \*  $\Gamma 1'$ )*  
*R (case-prod\$(\lambda\_2 a b. f' a b)\$p')* (**is** ?G  $\Gamma$ )  
**apply1** (*rule hn-refine-cons-pre*[*OF FR*])

**apply1** *extract-hnr-invalids*  
**apply1** (*cases p; cases p'; simp add: prod-assn-pair-conv[THEN prod-assn-ctxt]*)  
 ⟨*proof*⟩  
**applyS** (*simp add: hn-ctxt-def*)  
**applyS** *simp*  
**applyS** (*simp add: hn-ctxt-def entt-fr-refl entt-fr-drop*)  
 ⟨*proof*⟩

**lemma** *hn-case-prod-old:*

**assumes**  $P: \Gamma \Longrightarrow_t \Gamma 1 * \text{hn-ctxt } (\text{prod-assn } P1 \ P2) \ p' \ p$   
**assumes**  $R: \bigwedge a1 \ a2 \ a1' \ a2'. \llbracket p' = (a1', a2') \rrbracket$   
 $\implies \text{hn-refine } (\Gamma 1 * \text{hn-ctxt } P1 \ a1' \ a1 * \text{hn-ctxt } P2 \ a2' \ a2 * \text{hn-invalid}$   
 $(\text{prod-assn } P1 \ P2) \ p' \ p) \ (f \ a1 \ a2)$   
 $(\Gamma h \ a1 \ a1' \ a2 \ a2') \ R \ (f' \ a1' \ a2')$   
**assumes**  $M: \bigwedge a1 \ a1' \ a2 \ a2'. \Gamma h \ a1 \ a1' \ a2 \ a2'$   
 $\implies_t \Gamma' * \text{hn-ctxt } P1' \ a1' \ a1 * \text{hn-ctxt } P2' \ a2' \ a2 * \text{hn-ctxt } Pxx \ p' \ p$   
**shows**  $\text{hn-refine } \Gamma \ (\text{case-prod } f \ p) \ (\Gamma' * \text{hn-ctxt } (\text{prod-assn } P1' \ P2') \ p' \ p)$   
 $R \ (\text{case-prod } (\lambda_2 a \ b. \ f' \ a \ b) \ p')$   
**apply1** (*cases p; cases p'; simp*)  
**apply1** (*rule hn-refine-cons-pre[OF P]*)  
 ⟨*proof*⟩  
**apply1** (*rule enttI*)  
**applyS** (*sep-auto simp add: hn-ctxt-def invalid-assn-def mod-star-conv*)

**applyS** *simp*  
**apply1** (*rule entt-trans[OF M]*)  
**applyS** (*sep-auto intro!: enttI simp: hn-ctxt-def*)

**applyS** *simp*  
 ⟨*proof*⟩

**lemma** *hn-Pair[sepref-fr-rules]: hn-refine*  
 $(\text{hn-ctxt } P1 \ x1 \ x1' * \text{hn-ctxt } P2 \ x2 \ x2')$   
 $(\text{return } (x1', x2'))$   
 $(\text{hn-invalid } P1 \ x1 \ x1' * \text{hn-invalid } P2 \ x2 \ x2')$   
 $(\text{prod-assn } P1 \ P2)$   
 $(\text{RETURN } (\text{Pair } \$x1 \$x2))$   
 ⟨*proof*⟩

**lemma** *fst-hnr[sepref-fr-rules]: (return o fst, RETURN o fst) ∈ (prod-assn A B)<sup>d</sup>*  
 $\rightarrow_a A$   
 ⟨*proof*⟩

**lemma** *snd-hnr[sepref-fr-rules]: (return o snd, RETURN o snd) ∈ (prod-assn A B)<sup>d</sup>*  
 $\rightarrow_a B$   
 ⟨*proof*⟩

**lemmas** [*constraint-simps*] = *prod-assn-pure-conv*  
**lemmas** [*sepref-import-param*] = *param-prod-swap*

**lemma** *rdomp-prodD[dest!]*:  $rdomp (prod-assn A B) (a,b) \implies rdomp A a \wedge rdomp B b$   
 ⟨proof⟩

### 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' = ↑(v'=None)  
*option-assn* P v None = ↑(v=None)  
 ⟨proof⟩

**lemma** *option-assn-alt-def*: *option-assn* R a b =  
 (case (a,b) of (Some x, Some y) ⇒ R x y  
 | (None, None) ⇒ emp  
 | - ⇒ false)  
 ⟨proof⟩

**lemma** *option-assn-pure-conv[constraint-simps]*: *option-assn* (pure R) = pure (⟨R⟩*option-rel*)  
 ⟨proof⟩

**lemmas** [*sepref-import-rewrite*, *sepref-frame-normrel-eqs*, *fcomp-norm-unfold*] =  
*option-assn-pure-conv[symmetric]*

**lemma** *hr-comp-option-conv[simp, fcomp-norm-unfold]*:  
*hr-comp* (option-assn R) (⟨R'⟩*option-rel*)  
 = *option-assn* (*hr-comp* R R')  
 ⟨proof⟩

**lemma** *option-assn-precise[safe-constraint-rules]*:  
**assumes** *precise* P  
**shows** *precise* (option-assn P)  
 ⟨proof⟩

**lemma** *pure-option[safe-constraint-rules]*:  
**assumes** P: *is-pure* P  
**shows** *is-pure* (option-assn P)  
 ⟨proof⟩

**lemma** *hn-ctxt-option*: *option-assn* A x y = z ⟹ *hn-ctxt* (option-assn A) x y = z  
 ⟨proof⟩

**lemma** *hn-case-option*[*sepref-prep-comb-rule, sepref-comb-rules*]:  
**fixes**  $p\ p'\ P$   
**defines** [*simp*]:  $INVE \equiv hn\text{-invalid}\ (option\text{-assn}\ P)\ p\ p'$   
**assumes**  $FR: \Gamma \Longrightarrow_t hn\text{-ctxt}\ (option\text{-assn}\ P)\ p\ p' * F$   
**assumes**  $Rn: p=None \Longrightarrow hn\text{-refine}\ (hn\text{-ctxt}\ (option\text{-assn}\ P)\ p\ p' * F)\ f1'$   
 $(hn\text{-ctxt}\ XX1\ p\ p' * \Gamma1')\ R\ f1$   
**assumes**  $Rs: \bigwedge x\ x'. \llbracket p=Some\ x; p'=Some\ x' \rrbracket \Longrightarrow$   
 $hn\text{-refine}\ (hn\text{-ctxt}\ P\ x\ x' * INVE * F)\ (f2'\ x')\ (hn\text{-ctxt}\ P'\ x\ x' * hn\text{-ctxt}\ XX2$   
 $p\ p' * \Gamma2')\ R\ (f2\ x)$   
**assumes**  $MERGE1: \Gamma1' \vee_A \Gamma2' \Longrightarrow_t \Gamma'$   
**shows**  $hn\text{-refine}\ \Gamma\ (case\text{-option}\ f1'\ f2'\ p')\ (hn\text{-ctxt}\ (option\text{-assn}\ P')\ p\ p' * \Gamma')\ R$   
 $(case\text{-option}\ f1\ (\lambda_2x. f2\ x)\ p)$   
 $\langle proof \rangle$   
**apply1** *extract-hnr-invalids*  
 $\langle proof \rangle$   
**applyS** (*simp add: hn-ctxt-def*)  
  
 $\langle proof \rangle$   
**applyS** (*simp add: hn-ctxt-def*)  
 $\langle proof \rangle$   
**apply1** (*rule entt-fr-drop*)  
**applyS** (*simp add: hn-ctxt-def*)  
**apply1** (*rule entt-trans*[*OF - MERGE1*])  
**applyS** (*simp add: hn-ctxt-def*)  
 $\langle proof \rangle$

**lemma** *hn-None*[*sepref-fr-rules*]:  
 $hn\text{-refine}\ emp\ (return\ None)\ emp\ (option\text{-assn}\ P)\ (RETURN\ \$None)$   
 $\langle proof \rangle$

**lemma** *hn-Some*[*sepref-fr-rules*]: *hn-refine*  
 $(hn\text{-ctxt}\ P\ v\ v')$   
 $(return\ (Some\ v'))$   
 $(hn\text{-invalid}\ P\ v\ v')$   
 $(option\text{-assn}\ P)$   
 $(RETURN\ \$ (Some\ \$v))$   
 $\langle proof \rangle$

**definition** *imp-option-eq*  $eq\ a\ b \equiv case\ (a,b)\ of$   
 $(None, None) \Rightarrow return\ True$   
 $| (Some\ a, Some\ b) \Rightarrow eq\ a\ b$   
 $| - \Rightarrow return\ False$

**lemma** *option-assn-eq*[*sepref-comb-rules*]:  
**fixes**  $a\ b :: 'a\ option$   
**assumes**  $F1: \Gamma \Longrightarrow_t hn\text{-ctxt}\ (option\text{-assn}\ P)\ a\ a' * hn\text{-ctxt}\ (option\text{-assn}\ P)\ b\ b'$   
 $*\ \Gamma1$   
**assumes**  $EQ: \bigwedge va\ va'\ vb\ vb'. hn\text{-refine}$

$(hn-ctxt\ P\ va\ va' * hn-ctxt\ P\ vb\ vb' * \Gamma 1)$   
 $(eq'\ va'\ vb')$   
 $(\Gamma'\ va\ va'\ vb\ vb')$   
*bool-assn*  
 $(RETURN\ \$((=)\ \$va\$vb))$   
**assumes**  $F2$ :  
 $\bigwedge va\ va'\ vb\ vb'.$   
 $\Gamma'\ va\ va'\ vb\ vb' \implies_t hn-ctxt\ P\ va\ va' * hn-ctxt\ P\ vb\ vb' * \Gamma 1$   
**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*  
 $(RETURN\ \$((=)\ \$a\$b))$   
 $\langle proof \rangle$

**lemma** [*pat-rules*]:  
 $(=)\ \$a\$None \equiv is-None\$a$   
 $(=)\ \$None\$a \equiv is-None\$a$   
 $\langle proof \rangle$

**lemma** *hn-is-None*[*sepref-fr-rules*]: *hn-refine*  
 $(hn-ctxt\ (option-assn\ P)\ a\ a')$   
 $(return\ (is-None\ a'))$   
 $(hn-ctxt\ (option-assn\ P)\ a\ a')$   
*bool-assn*  
 $(RETURN\ \$(is-None\$a))$   
 $\langle proof \rangle$

**lemma** (**in**  $-$ ) *sepref-the-complete*[*sepref-fr-rules*]:  
**assumes**  $x \neq None$   
**shows** *hn-refine*  
 $(hn-ctxt\ (option-assn\ R)\ x\ xi)$   
 $(return\ (the\ xi))$   
 $(hn-invalid\ (option-assn\ R)\ x\ xi)$   
 $(R)$   
 $(RETURN\ \$(the\$x))$   
 $\langle proof \rangle$

**lemma** (**in**  $-$ ) *sepref-the-id*:  
**assumes** *CONSTRAINT* (*IS-PURE IS-ID*)  $R$   
**shows** *hn-refine*  
 $(hn-ctxt\ (option-assn\ R)\ x\ xi)$   
 $(return\ (the\ xi))$   
 $(hn-ctxt\ (option-assn\ R)\ x\ xi)$   
 $(R)$   
 $(RETURN\ \$(the\$x))$   
 $\langle proof \rangle$

## 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=[]))  
⟨proof⟩

**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'  
⟨proof⟩

**lemma** *list-assn-aux-ineq-len*: *length* l ≠ *length* li ⇒ *list-assn* A l li = false  
⟨proof⟩

**lemma** *list-assn-aux-append2*[simp]:

**assumes** *length* l2 = *length* l2'  
**shows** *list-assn* P (l1@l2) (l1'@l2')  
= *list-assn* P l1 l1' \* *list-assn* P l2 l2'  
⟨proof⟩

**lemma** *list-assn-pure-conv*[constraint-simps]: *list-assn* (pure R) = pure (⟨R⟩*list-rel*)  
⟨proof⟩

**lemmas** [*seprel-import-rewrite*, *seprel-frame-normrel-egs*, *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) [] [] = 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  
⟨proof⟩

**lemma** *list-assn-precise*[constraint-rules]: *precise* P ⇒ *precise* (*list-assn* P)  
⟨proof⟩

**lemma** *list-assn-pure*[constraint-rules]:

**assumes** P: *is-pure* P  
**shows** *is-pure* (*list-assn* P)  
⟨proof⟩

**lemma** *list-assn-mono*:

$\llbracket \bigwedge x x'. P x x' \implies_A P' x x' \rrbracket \implies \text{list-assn } P \text{ } l \text{ } l' \implies_A \text{list-assn } P' \text{ } l \text{ } l'$   
 ⟨proof⟩

**lemma** *list-assn-monot*:

$\llbracket \bigwedge x x'. P x x' \implies_t P' x x' \rrbracket \implies \text{list-assn } P \text{ } l \text{ } l' \implies_t \text{list-assn } P' \text{ } l \text{ } l'$   
 ⟨proof⟩

**lemma** *list-match-cong*[*sepref-frame-match-rules*]:

$\llbracket \bigwedge x x'. \llbracket x \in \text{set } l; x' \in \text{set } l' \rrbracket \implies \text{hn-ctxt } A \text{ } x \text{ } x' \implies_t \text{hn-ctxt } A' \text{ } x \text{ } x' \rrbracket \implies \text{hn-ctxt}$   
 $(\text{list-assn } A) \text{ } l \text{ } l' \implies_t \text{hn-ctxt } (\text{list-assn } A') \text{ } l \text{ } l'$   
 ⟨proof⟩

**lemma** *list-merge-cong*[*sepref-frame-merge-rules*]:

**assumes**  $\bigwedge x x'. \llbracket x \in \text{set } l; x' \in \text{set } l' \rrbracket \implies \text{hn-ctxt } A \text{ } x \text{ } x' \vee_A \text{hn-ctxt } A' \text{ } x \text{ } x' \implies_t$   
 $\text{hn-ctxt } Am \text{ } x \text{ } x'$   
**shows**  $\text{hn-ctxt } (\text{list-assn } A) \text{ } l \text{ } l' \vee_A \text{hn-ctxt } (\text{list-assn } A') \text{ } l \text{ } l' \implies_t \text{hn-ctxt } (\text{list-assn}$   
 $Am) \text{ } l \text{ } l'$   
 ⟨proof⟩

**lemma** *invalid-list-split*:

$\text{invalid-assn } (\text{list-assn } A) \text{ } (x \# xs) \text{ } (y \# ys) \implies_t \text{invalid-assn } A \text{ } x \text{ } y * \text{invalid-assn}$   
 $(\text{list-assn } A) \text{ } xs \text{ } ys$   
 ⟨proof⟩

**lemma** *entt-invalid-list*:  $\text{hn-invalid } (\text{list-assn } A) \text{ } l \text{ } l' \implies_t \text{hn-ctxt } (\text{list-assn } (\text{invalid-assn}$   
 $A)) \text{ } l \text{ } l'$

⟨proof⟩

**applyS** *simp*

⟨proof⟩

**apply1** (*simp add: hn-ctxt-def cong del: invalid-assn-cong*)

**apply1** (*rule entt-trans[OF invalid-list-split]*)

⟨proof⟩

**applyS** *simp*

⟨proof⟩

**applyS** *assumption*

**applyS** *simp*

⟨proof⟩

**applyS** (*simp add: hn-ctxt-def invalid-assn-def*)

**applyS** (*simp add: hn-ctxt-def invalid-assn-def*)

⟨proof⟩

**lemmas** *invalid-list-merge*[*sepref-frame-merge-rules*] = *gen-merge-cons*[*OF entt-invalid-list*]

**lemma** *list-assn-comp*[*fcomp-norm-unfold*]:  $\text{hr-comp } (\text{list-assn } A) \text{ } (\langle B \rangle \text{list-rel}) =$



*list-assn* (*hr-comp* *A B*)  
 ⟨*proof*⟩

**lemma** *hn-ctxt-eq*:  $A \ x \ y = z \implies \text{hn-ctxt } A \ x \ y = z$  ⟨*proof*⟩

**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*]:  $INVE \equiv \text{hn-invalid } (list-assn \ P) \ p \ p'$

**assumes** *FR*:  $\Gamma \implies_t \text{hn-ctxt } (list-assn \ P) \ p \ p' * F$

**assumes** *Rn*:  $p = [] \implies \text{hn-refine } (\text{hn-ctxt } (list-assn \ P) \ p \ p' * F) \ f1' \ (\text{hn-ctxt } XX1 \ p \ p' * \Gamma 1') \ R \ f1$

**assumes** *Rs*:  $\bigwedge x \ l \ x' \ l'. \llbracket p = x \# l; p' = x' \# l' \rrbracket \implies$

$\text{hn-refine } (\text{hn-ctxt } P \ x \ x' * \text{hn-ctxt } (list-assn \ P) \ l \ l' * INVE * F) \ (f2' \ x' \ l') \ (\text{hn-ctxt } P1' \ x \ x' * \text{hn-ctxt } (list-assn \ P2') \ l \ l' * \text{hn-ctxt } XX2 \ p \ p' * \Gamma 2') \ R \ (f2 \ x \ l)$

**assumes** *MERGE1*[*unfolded hn-ctxt-def*]:  $\bigwedge x \ x'. \text{hn-ctxt } P1' \ x \ x' \vee_A \text{hn-ctxt } P2' \ x \ x' \implies_t \text{hn-ctxt } P' \ x \ x'$

**assumes** *MERGE2*:  $\Gamma 1' \vee_A \Gamma 2' \implies_t \Gamma'$

**shows**  $\text{hn-refine } \Gamma \ (\text{case-list } f1' \ f2' \ p') \ (\text{hn-ctxt } (list-assn \ P') \ p \ p' * \Gamma') \ R \ (\text{case-list } f1 \ (\lambda_2 x \ l. \ f2 \ x \ l)) \ p$

⟨*proof*⟩

**apply1** *extract-hnr-invalids*

⟨*proof*⟩

**applyS** (*simp add: hn-ctxt-def*)

⟨*proof*⟩

**applyS** (*simp add: hn-ctxt-def*)

⟨*proof*⟩

**apply1** (*rule entt-fr-drop*)

⟨*proof*⟩

**apply1** (*simp add: hn-ctxt-def*)

**apply1** (*rule entt-trans[OF - MERGE1]*)

**applyS** (*simp*)

**apply1** (*simp add: hn-ctxt-def*)

⟨*proof*⟩

**apply1** (*rule entt-trans[OF - MERGE1]*)

**applyS** (*simp*)

**apply1** (*rule entt-trans[OF - MERGE2]*)

**applyS** (*simp*)

⟨*proof*⟩

**lemma** *hn-Nil*[*sepref-fr-rules*]:

$\text{hn-refine } emp \ (\text{return } []) \ emp \ (list-assn \ P) \ (RETURN \$ [])$

⟨*proof*⟩

**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*\$((\$\#) \$\\$xs\$))  
*<proof>*

**lemma** *list-assn-aux-len*:  
*list-assn*  $P$   $l$   $l'$  = *list-assn*  $P$   $l$   $l'$  \*  $\uparrow(\text{length } l = \text{length } l')$   
*<proof>*

**lemma** *list-assn-aux-eqlen-simp*:  
*vassn-tag* (*list-assn*  $P$   $l$   $l'$ )  $\implies$   $\text{length } l' = \text{length } l$   
 $h \models (\text{list-assn } P \ l \ l') \implies \text{length } l' = \text{length } l$   
*<proof>*

**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\$))  
*<proof>*

**lemma** *list-assn-aux-cons-conv1*:  
*list-assn*  $R$  ( $a\#l$ )  $m$  =  $(\exists_A b \ m'. R \ a \ b * \text{list-assn } R \ l \ m' * \uparrow(m=b\#m'))$   
*<proof>*

**lemma** *list-assn-aux-cons-conv2*:  
*list-assn*  $R$   $l$  ( $b\#m$ ) =  $(\exists_A a \ l'. R \ a \ b * \text{list-assn } R \ l' \ m * \uparrow(l=a\#l'))$   
*<proof>*

**lemmas** *list-assn-aux-cons-conv* = *list-assn-aux-cons-conv1* *list-assn-aux-cons-conv2*

**lemma** *list-assn-aux-append-conv1*:  
*list-assn*  $R$  ( $l1\@l2$ )  $m$  =  $(\exists_A m1 \ m2. \text{list-assn } R \ l1 \ m1 * \text{list-assn } R \ l2 \ m2 * \uparrow(m=m1\@m2))$   
*<proof>*

**lemma** *list-assn-aux-append-conv2*:  
*list-assn*  $R$   $l$  ( $m1\@m2$ ) =  $(\exists_A l1 \ l2. \text{list-assn } R \ l1 \ m1 * \text{list-assn } R \ l2 \ m2 * \uparrow(l=l1\@l2))$   
*<proof>*

**lemmas** *list-assn-aux-append-conv* = *list-assn-aux-append-conv1* *list-assn-aux-append-conv2*

**declare** *param-upt*[sepref-import-param]

### 2.1.13 Sum-Type

**fun** *sum-assn* :: ( $'ai \Rightarrow 'a \Rightarrow \text{assn}$ )  $\Rightarrow$  ( $'bi \Rightarrow 'b \Rightarrow \text{assn}$ )  $\Rightarrow$  ( $'ai+'bi$ )  $\Rightarrow$  ( $'a+'b$ )  
 $\Rightarrow \text{assn}$  **where**  
*sum-assn*  $A$   $B$  (*Inl*  $ai$ ) (*Inl*  $a$ ) =  $A \ ai \ a$

|  $sum\text{-}assn\ A\ B\ (Inr\ bi)\ (Inr\ b) = B\ bi\ b$   
|  $sum\text{-}assn\ A\ B\ -\ - = false$

**notation**  $sum\text{-}assn$  (**infixr**  $\langle +_a \rangle$  67)

**lemma**  $sum\text{-}assn\text{-}pure[safe\text{-}constraint\text{-}rules]$ :  $\llbracket is\text{-}pure\ A; is\text{-}pure\ B \rrbracket \implies is\text{-}pure\ (sum\text{-}assn\ A\ B)$   
 $\langle proof \rangle$

**lemma**  $sum\text{-}assn\text{-}id[simp]$ :  $sum\text{-}assn\ id\text{-}assn\ id\text{-}assn = id\text{-}assn$   
 $\langle proof \rangle$

**lemma**  $sum\text{-}assn\text{-}pure\text{-}conv[simp]$ :  $sum\text{-}assn\ (pure\ A)\ (pure\ B) = pure\ (\langle A,B \rangle sum\text{-}rel)$   
 $\langle proof \rangle$

**lemma**  $sum\text{-}match\text{-}cong[sepref\text{-}frame\text{-}match\text{-}rules]$ :

$\llbracket$   
 $\wedge x\ y. \llbracket e = Inl\ x; e' = Inl\ y \rrbracket \implies hn\text{-}ctxt\ A\ x\ y \implies_t hn\text{-}ctxt\ A'\ x\ y;$   
 $\wedge x\ y. \llbracket e = Inr\ x; e' = Inr\ y \rrbracket \implies hn\text{-}ctxt\ B\ x\ y \implies_t hn\text{-}ctxt\ B'\ x\ y$   
 $\rrbracket \implies hn\text{-}ctxt\ (sum\text{-}assn\ A\ B)\ e\ e' \implies_t hn\text{-}ctxt\ (sum\text{-}assn\ A'\ B')\ e\ e'$   
 $\langle proof \rangle$

**lemma**  $enum\text{-}merge\text{-}cong[sepref\text{-}frame\text{-}merge\text{-}rules]$ :

**assumes**  $\wedge x\ y. \llbracket e = Inl\ x; e' = Inl\ y \rrbracket \implies hn\text{-}ctxt\ A\ x\ y \vee_A hn\text{-}ctxt\ A'\ x\ y \implies_t$   
 $hn\text{-}ctxt\ Am\ x\ y$   
**assumes**  $\wedge x\ y. \llbracket e = Inr\ x; e' = Inr\ y \rrbracket \implies hn\text{-}ctxt\ B\ x\ y \vee_A hn\text{-}ctxt\ B'\ x\ y \implies_t$   
 $hn\text{-}ctxt\ Bm\ x\ y$   
**shows**  $hn\text{-}ctxt\ (sum\text{-}assn\ A\ B)\ e\ e' \vee_A hn\text{-}ctxt\ (sum\text{-}assn\ A'\ B')\ e\ e' \implies_t hn\text{-}ctxt$   
 $(sum\text{-}assn\ Am\ Bm)\ e\ e'$   
 $\langle proof \rangle$

**lemma**  $entt\text{-}invalid\text{-}sum$ :  $hn\text{-}invalid\ (sum\text{-}assn\ A\ B)\ e\ e' \implies_t hn\text{-}ctxt\ (sum\text{-}assn$   
 $(invalid\text{-}assn\ A)\ (invalid\text{-}assn\ B))\ e\ e'$   
 $\langle proof \rangle$

**lemmas**  $invalid\text{-}sum\text{-}merge[sepref\text{-}frame\text{-}merge\text{-}rules] = gen\text{-}merge\text{-}cons[OF\ entt\text{-}invalid\text{-}sum]$

**sepref-register**  $Inr\ Inl$

**lemma**  $[sepref\text{-}fr\text{-}rules]$ :  $(return\ o\ Inl, RETURN\ o\ Inl) \in A^d \rightarrow_a sum\text{-}assn\ A\ B$   
 $\langle proof \rangle$

**lemma**  $[sepref\text{-}fr\text{-}rules]$ :  $(return\ o\ Inr, RETURN\ o\ Inr) \in B^d \rightarrow_a sum\text{-}assn\ A\ B$   
 $\langle proof \rangle$

**sepref-register**  $case\text{-}sum$

In the monadify phase, this eta-expands to make visible all required arguments

**lemma** [*sepref-monadify-arity*]:  $case\text{-}sum \equiv \lambda_2 f1\ f2\ x.\ SP\ case\text{-}sum\ (\lambda_2 x.\ f1\ \$x)\ (\lambda_2 x.\ f2\ \$x)\ \$x$   
 ⟨*proof*⟩

This determines an evaluation order for the first-order operands

**lemma** [*sepref-monadify-comb*]:  $case\text{-}sum\ f1\ f2\ \$x \equiv (\gg) \$ (EVAL\ \$x)\ (\lambda_2 x.\ SP\ case\text{-}sum\ f1\ f2\ \$x)$  ⟨*proof*⟩

This enables translation of the case-distinction in a non-monadic context.

**lemma** [*sepref-monadify-comb*]:  $EVAL\ (case\text{-}sum\ (\lambda_2 x.\ f1\ x)\ (\lambda_2 x.\ f2\ x)\ \$x) \equiv (\gg) \$ (EVAL\ \$x)\ (\lambda_2 x.\ SP\ case\text{-}sum\ (\lambda_2 x.\ EVAL\ \$\ f1\ x)\ (\lambda_2 x.\ EVAL\ \$\ f2\ x)\ \$x)$   
 ⟨*proof*⟩

Auxiliary lemma, to lift simp-rule over *hn-ctxt*

**lemma** *sum-assn-ctxt*:  $sum\text{-}assn\ A\ B\ x\ y = z \implies hn\text{-}ctxt\ (sum\text{-}assn\ A\ B)\ x\ y = z$   
 ⟨*proof*⟩

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*]:  $INVe \equiv hn\text{-}invalid\ (sum\text{-}assn\ A\ B)\ e\ e'$

**assumes**  $FR: \Gamma \implies_t hn\text{-}ctxt\ (sum\text{-}assn\ A\ B)\ e\ e' * F$

**assumes**  $E1: \bigwedge x1\ x1a.\ \llbracket e = Inl\ x1; e' = Inl\ x1a \rrbracket \implies hn\text{-}refine\ (hn\text{-}ctxt\ A\ x1\ x1a * INVe * F)\ (f1'\ x1a)\ (hn\text{-}ctxt\ A'\ x1\ x1a * hn\text{-}ctxt\ XX1\ e\ e' * \Gamma 1')\ R\ (f1\ x1)$

**assumes**  $E2: \bigwedge x2\ x2a.\ \llbracket e = Inr\ x2; e' = Inr\ x2a \rrbracket \implies hn\text{-}refine\ (hn\text{-}ctxt\ B\ x2\ x2a * INVe * F)\ (f2'\ x2a)\ (hn\text{-}ctxt\ B'\ x2\ x2a * hn\text{-}ctxt\ XX2\ e\ e' * \Gamma 2')\ R\ (f2\ x2)$

**assumes**  $MERGE[unfolding\ hn\text{-}ctxt\text{-}def]: \Gamma 1' \vee_A \Gamma 2' \implies_t \Gamma'$

**shows**  $hn\text{-}refine\ \Gamma\ (case\text{-}sum\ f1'\ f2'\ e')\ (hn\text{-}ctxt\ (sum\text{-}assn\ A'\ B')\ e\ e' * \Gamma')\ R\ (case\text{-}sum\ (\lambda_2 x.\ f1\ x)\ (\lambda_2 x.\ f2\ x)\ \$e)$

⟨*proof*⟩

**apply1** *extract-hnr-invalids*

⟨*proof*⟩

**applyS** (*simp add: hn-ctxt-def*) — Match precondition for case, get *enum-assn* from assumption generated by *extract-hnr-invalids*

⟨*proof*⟩

**apply1** (*rule entt-fr-drop*)

**applyS** (*simp add: hn-ctxt-def entt-disjI1' entt-disjI2'*)

**apply1** (*rule entt-trans[OF - MERGE]*)

```

applyS (simp add: entt-disjI1' entt-disjI2')
⟨proof⟩
applyS (simp add: hn-ctxt-def)
⟨proof⟩
apply1 (rule entt-fr-drop)
applyS (simp add: hn-ctxt-def entt-disjI1' entt-disjI2')
apply1 (rule entt-trans[OF - MERGE])
applyS (simp add: entt-disjI1' entt-disjI2')
⟨proof⟩

```

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

```

lemmas [sepref-comb-rules] = sum-cases-hnr[sepref-prep-comb-rule]

```

```

sepref-register isl projl projr

```

```

lemma isl-hnr[sepref-fr-rules]: (return o isl, RETURN o isl) ∈ (sum-assn A B)k
→a bool-assn
⟨proof⟩

```

```

lemma projl-hnr[sepref-fr-rules]: (return o projl, RETURN o projl) ∈ [isl]a (sum-assn
A B)d → A
⟨proof⟩

```

```

lemma projr-hnr[sepref-fr-rules]: (return o projr, RETURN o projr) ∈ [Not o isl]a
(sum-assn A B)d → B
⟨proof⟩

```

### 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
⟨proof⟩

```

```

lemma empty-literal-pat [def-pat-rules]:
String.empty-literal ≡ UNPROTECT String.empty-literal
⟨proof⟩

```

```

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*)  $\in$  *Id*  
 <proof>

**end**

**lemma** *Literal-pat* [*def-pat-rules*]:

*String.Literal* \$ *b0* \$ *b1* \$ *b2* \$ *b3* \$ *b4* \$ *b5* \$ *b6* \$ *s*  $\equiv$   
*UNPROTECT* (*String.Literal* \$ *b0* \$ *b1* \$ *b2* \$ *b3* \$ *b4* \$ *b5* \$ *b6* \$ *s*)  
 <proof>

**end**

## 2.2 Setup for Foreach Combinator

**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  $\lambda x. \text{RETURN } (c x)$

**definition** *FOREACH-inv* *xs*  $\Phi$  *s*  $\equiv$

*case s of* (*it*,  $\sigma$ )  $\Rightarrow \exists xs'. xs = xs' @ it \wedge \Phi$  (*set it*)  $\sigma$

**definition** *monadic-FOREACH* *R*  $\Phi$  *S* *c* *f*  $\sigma 0$   $\equiv$  *do* {

*ASSERT* (*finite S*);

*xs0*  $\leftarrow$  *it-to-sorted-list* *R S*;

( $-, \sigma$ )  $\leftarrow$  *RECT* ( $\lambda W (xs, \sigma). do$  {

*ASSERT* (*FOREACH-inv* *xs0*  $\Phi$  (*xs, \sigma*));

*if* *xs*  $\neq []$  *then do* {

*b*  $\leftarrow$  *c*  $\sigma$ ;

*if* *b* *then*

*FOREACH-body* *f* (*xs, \sigma*)  $\gg=$  *W*

*else*

*RETURN* (*xs, \sigma*)

} *else RETURN* (*xs, \sigma*)

} (*xs0, \sigma 0*);

*RETURN*  $\sigma$

}

**lemma** *FOREACH-oci-to-monadic*:

*FOREACHoci* *R*  $\Phi$  *S* *c* *f*  $\sigma 0 = monadic-FOREACH *R*  $\Phi$  *S* ( $\lambda \sigma. \text{RETURN } (c$   
 $\sigma)) f \sigma 0$   
 <proof>$

Next, we define a characterization w.r.t. *nfoldli*

**definition** *monadic-nfoldli*  $l\ c\ f\ s \equiv \text{RECT } (\lambda D\ (l,s)). \text{ case } l \text{ of}$

$$\begin{aligned} & \square \Rightarrow \text{RETURN } s \\ & | x\#\!ls \Rightarrow \text{do } \{ \\ & \quad b \leftarrow c\ s; \\ & \quad \text{if } b \text{ then } \text{do } \{ s' \leftarrow f\ x\ s; D\ (ls,s') \} \text{ else } \text{RETURN } s \\ & \quad \} \\ & \} (l,s) \end{aligned}$$

**lemma** *monadic-nfoldli-eq*:

$$\begin{aligned} \text{monadic-nfoldli } l\ c\ f\ s = & ( \\ & \text{case } l \text{ of} \\ & \square \Rightarrow \text{RETURN } s \\ & | x\#\!ls \Rightarrow \text{do } \{ \\ & \quad b \leftarrow c\ s; \\ & \quad \text{if } b \text{ then } f\ x\ s \ggg \text{monadic-nfoldli } ls\ c\ f \text{ else } \text{RETURN } s \\ & \quad \} \\ & ) \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *monadic-nfoldli-simp[simp]*:

$$\begin{aligned} \text{monadic-nfoldli } \square\ c\ f\ s &= \text{RETURN } s \\ \text{monadic-nfoldli } (x\#\!ls)\ c\ f\ s &= \text{do } \{ \\ & \quad b \leftarrow c\ s; \\ & \quad \text{if } b \text{ then } f\ x\ s \ggg \text{monadic-nfoldli } ls\ c\ f \text{ else } \text{RETURN } s \\ & \quad \} \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *nfoldli-to-monadic*:

$$\text{nfoldli } l\ c\ f = \text{monadic-nfoldli } l\ (\lambda x. \text{RETURN } (c\ x))\ f$$

*<proof>*

**definition** *nfoldli-alt*  $l\ c\ f\ s \equiv \text{RECT } (\lambda D\ (l,s)). \text{ case } l \text{ of}$

$$\begin{aligned} & \square \Rightarrow \text{RETURN } s \\ & | x\#\!ls \Rightarrow \text{do } \{ \\ & \quad \text{let } b = c\ s; \\ & \quad \text{if } b \text{ then } \text{do } \{ s' \leftarrow f\ x\ s; D\ (ls,s') \} \text{ else } \text{RETURN } s \\ & \quad \} \\ & \} (l,s) \end{aligned}$$

**lemma** *nfoldli-alt-eq*:

$$\begin{aligned} \text{nfoldli-alt } l\ c\ f\ s = & ( \\ & \text{case } l \text{ of} \\ & \square \Rightarrow \text{RETURN } s \\ & | x\#\!ls \Rightarrow \text{do } \{ \text{let } b = c\ s; \text{ if } b \text{ then } f\ x\ s \ggg \text{nfoldli-alt } ls\ c\ f \text{ else } \text{RETURN } s \} \\ & ) \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *nfoldli-alt-simp[simp]*:

$nfoldli\text{-}alt \ [] \ c \ f \ s = RETURN \ s$   
 $nfoldli\text{-}alt \ (x\#ls) \ c \ f \ s = do \ \{$   
 $\quad let \ b = c \ s;$   
 $\quad if \ b \ then \ f \ x \ s \gg\gg \ nfoldli\text{-}alt \ ls \ c \ f \ else \ RETURN \ s$   
 $\}$   
 $\langle proof \rangle$

**lemma** *nfoldli-alt*:

$(nfoldli::'a \ list \Rightarrow ('b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow 'b \ nres) \Rightarrow 'b \Rightarrow 'b \ nres)$   
 $= nfoldli\text{-}alt$   
 $\langle proof \rangle$

**lemma** *monadic-nfoldli-rec*:

$monadic\text{-}nfoldli \ x' \ c \ f \ \sigma$   
 $\leq \Downarrow Id \ (REC_T$   
 $\quad (\lambda W \ (xs, \sigma).$   
 $\quad \quad ASSERT \ (FOREACH\text{-}inv \ xs \ 0 \ I \ (xs, \sigma)) \gg\gg$   
 $\quad \quad (\lambda\text{-}. \ if \ xs = [] \ then \ RETURN \ (xs, \sigma)$   
 $\quad \quad \quad else \ c \ \sigma \gg\gg$   
 $\quad \quad \quad (\lambda b. \ if \ b \ then \ FOREACH\text{-}body \ f \ (xs, \sigma) \gg\gg W$   
 $\quad \quad \quad \quad else \ RETURN \ (xs, \sigma))))$   
 $\quad (x', \sigma) \gg\gg$   
 $\quad (\lambda(-, y). \ RETURN \ y))$   
 $\langle proof \rangle$

**lemma** *monadic-nfoldli-arities*[*sepref-monadify-arity*]:

$monadic\text{-}nfoldli \equiv \lambda_2 s \ c \ f \ \sigma. \ SP \ (monadic\text{-}nfoldli) \ \$s \ (\lambda_2 x. \ c \ \$x) \ (\lambda_2 x \ \sigma. \ f \ \$x \ \$\sigma) \ \$\sigma$   
 $\langle proof \rangle$

**lemma** *monadic-nfoldli-comb*[*sepref-monadify-comb*]:

$\bigwedge s \ c \ f \ \sigma. \ (monadic\text{-}nfoldli) \ \$s \ \$c \ \$f \ \$\sigma \equiv$   
 $\quad Refine\text{-}Basic.\ bind \ \$ \ (EVAL \ \$s) \ (\lambda_2 s. \ Refine\text{-}Basic.\ bind \ \$ \ (EVAL \ \$\sigma) \ (\lambda_2 \sigma. \$   
 $\quad \quad SP \ (monadic\text{-}nfoldli) \ \$s \ \$c \ \$f \ \$\sigma$   
 $\quad \quad ))$   
 $\langle proof \rangle$

**lemma** *list-rel-congD*:

**assumes**  $A: (li, l) \in \langle S \rangle list\text{-}rel$   
**shows**  $(li, l) \in \langle S \cap (set \ li \times set \ l) \rangle list\text{-}rel$   
 $\langle proof \rangle$

**lemma** *monadic-nfoldli-refine*[*refine*]:

**assumes**  $L: (li, l) \in \langle S \rangle list\text{-}rel$   
**and** [*simp*]:  $(si, s) \in R$   
**and**  $CR[refine]: \bigwedge si \ s. \ (si, s) \in R \implies ci \ si \leq \Downarrow bool\text{-}rel \ (c \ s)$   
**and** [*refine*]:  $\bigwedge xi \ x \ si \ s. \ [\ (xi, x) \in S; \ x \in set \ l; \ (si, s) \in R; \ inres \ (c \ s) \ True \ ] \implies fi$   
 $xi \ si \leq \Downarrow R \ (f \ x \ s)$   
**shows**  $monadic\text{-}nfoldli \ li \ ci \ fi \ si \leq \Downarrow R \ (monadic\text{-}nfoldli \ l \ c \ f \ s)$



*<proof>*

**lemma** *monadic-FOREACH-itsl*:

**fixes**  $R I tsl$

**shows**

$do \{ l \leftarrow it\text{-to-sorted-list } R s; monadic\text{-nfoldli } l c f \sigma \}$   
 $\leq monadic\text{-FOREACH } R I s c f \sigma$   
*<proof>*

**lemma** *FOREACHoci-itsl*:

**fixes**  $R I tsl$

**shows**

$do \{ l \leftarrow it\text{-to-sorted-list } R s; nfoldli } l c f \sigma \}$   
 $\leq FOREACHoci } R I s c f \sigma$   
*<proof>*

**lemma** [*def-pat-rules*]:

$FOREACHc \equiv PR\text{-CONST } (FOREACHoci } (\lambda - . True) } (\lambda - . True))$   
 $FOREACHci } I \equiv PR\text{-CONST } (FOREACHoci } (\lambda - . True) } I)$   
 $FOREACHi } I \equiv \lambda_2 s. PR\text{-CONST } (FOREACHoci } (\lambda - . True) } I) \$s \$(\lambda_2 x. True)$   
 $FOREACH \equiv FOREACHi } (\lambda_2 - . True)$   
*<proof>*

**term** *FOREACHoci } R I*

**lemma** *id-FOREACHoci[id-rules]*:  $PR\text{-CONST } (FOREACHoci } R I) ::_i$

$TYPE('c \text{ set} \Rightarrow ('d \Rightarrow bool) \Rightarrow ('c \Rightarrow 'd \Rightarrow 'd \text{ nres}) \Rightarrow 'd \Rightarrow 'd \text{ nres})$   
*<proof>*

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]*:

$PR\text{-CONST } (FOREACHoci } R I) \equiv \lambda_2 s c f \sigma. SP (PR\text{-CONST } (FOREACHoci } R I) \$s \$(\lambda_2 x. c \$x) \$(\lambda_2 x \sigma. f \$x \$\sigma) \$\sigma$   
*<proof>*

**lemma** *FOREACHoci-comb[sepref-monadify-comb]*:

$\bigwedge s c f \sigma. (PR\text{-CONST } (FOREACHoci } R I) \$s \$(\lambda_2 x. c x) \$f \$\sigma \equiv$   
 $Refine\text{-Basic.bind} \$ (EVAL \$s) \$(\lambda_2 s. Refine\text{-Basic.bind} \$ (EVAL \$\sigma) \$(\lambda_2 \sigma. SP (PR\text{-CONST } (monadic\text{-FOREACH } R I) \$s \$(\lambda_2 x. (EVAL \$ (c x))) \$f \$\sigma$   
 $)))$   
*<proof>*

## 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*  $l\ c\ f\ s \equiv \text{heap.fixp-fun } (\lambda D\ (l,s). \text{ case } l \text{ of}$   
 $\quad [] \Rightarrow \text{return } s$   
 $\quad | x\#\!ls \Rightarrow \text{do } \{$   
 $\quad \quad b \leftarrow c\ s;$   
 $\quad \quad \text{if } b \text{ then do } \{ s' \leftarrow f\ x\ s; D\ (ls,s') \}$  *else return*  $s$   
 $\quad \quad \}$   
 $\quad \}$   
 $\quad \})\ (l,s)$

**declare** *imp-nfoldli-def*[*code del*]

**lemma** *imp-nfoldli-simps*[*simp,code*]:  
 $\text{imp-nfoldli } []\ c\ f\ s = \text{return } s$   
 $\text{imp-nfoldli } (x\#\!ls)\ c\ f\ s = (\text{do } \{$   
 $\quad b \leftarrow c\ s;$   
 $\quad \text{if } b \text{ then do } \{$   
 $\quad \quad s' \leftarrow f\ x\ s;$   
 $\quad \quad \text{imp-nfoldli } ls\ c\ f\ s'$   
 $\quad \quad \}$  *else return*  $s$   
 $\quad \})$   
 $\langle \text{proof} \rangle$

**lemma** *monadic-nfoldli-refine-aux*:

**assumes** *c-ref*:  $\bigwedge s\ s'. \text{ hn-refine}$

$(\Gamma * \text{hn-ctxt } Rs\ s'\ s)$

$(c\ s)$

$(\Gamma * \text{hn-ctxt } Rs\ s'\ s)$

*bool-assn*

$(c'\ s')$

**assumes** *f-ref*:  $\bigwedge x\ x'\ s\ s'. \text{ hn-refine}$

$(\Gamma * \text{hn-ctxt } Rl\ x'\ x * \text{hn-ctxt } Rs\ s'\ s)$

$(f\ x\ s)$

$(\Gamma * \text{hn-invalid } Rl\ x'\ x * \text{hn-invalid } Rs\ s'\ s) Rs$

$(f'\ x'\ s')$

**shows** *hn-refine*

$(\Gamma * \text{hn-ctxt } (\text{list-assn } Rl)\ l'\ l * \text{hn-ctxt } Rs\ s'\ s)$

$(\text{imp-nfoldli } l\ c\ f\ s)$

$(\Gamma * \text{hn-invalid } (\text{list-assn } Rl)\ l'\ l * \text{hn-invalid } Rs\ s'\ s) Rs$

$(\text{monadic-nfoldli } l'\ c'\ f'\ s')$

**applyF** (*induct*  $p \equiv Rl\ l'\ l$ )

*arbitrary*:  $s\ s'$

*rule*: *list-assn.induct*)

**applyF** *simp*

$\langle \text{proof} \rangle$

**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)
  <proof>
  apply1 (fr-rot 3, rule fr-refl)
  apply1 (fr-rot 3, rule fr-refl)
  applyS (rule ent-refl)
solved

  <proof>

  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

  <proof>
  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

```

$\langle proof \rangle$

**lemma** *hn-monadic-nfoldli*:

**assumes** *FR*:  $P \Longrightarrow_t \Gamma * hn-ctxt (list-assn\ Rl)\ l'\ l * hn-ctxt\ Rs\ s'\ s$

**assumes** *c-ref*:  $\bigwedge s\ s'.\ hn-refine$

$(\Gamma * hn-ctxt\ Rs\ s'\ s)$

$(c\ s)$

$(\Gamma * hn-ctxt\ Rs\ s'\ s)$

*bool-assn*

$(c'\$s')$

**assumes** *f-ref*:  $\bigwedge x\ x'\ s\ s'.\ hn-refine$

$(\Gamma * hn-ctxt\ Rl\ x'\ x * hn-ctxt\ Rs\ s'\ s)$

$(f\ x\ s)$

$(\Gamma * hn-invalid\ Rl\ x'\ x * hn-invalid\ Rs\ s'\ s)\ Rs$

$(f'\$x'\$s')$

**shows** *hn-refine*

*P*

$(imp-nfoldli\ l\ c\ f\ s)$

$(\Gamma * hn-invalid\ (list-assn\ Rl)\ l'\ l * hn-invalid\ Rs\ s'\ s)$

*Rs*

$(monadic-nfoldli\ l'\$c'\$f'\$s')$

$\langle proof \rangle$

**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*]:

$\bigwedge x\ d.\ mono-Heap\ (\lambda xa.\ B\ x\ xa\ d)$

$\bigwedge Z\ xa.\ mono-Heap\ (\lambda a.\ B\ a\ Z\ xa)$

**shows** *mono-Heap*  $(\lambda x.\ heap.fixp-fun\ (\lambda D\ \sigma.\ B\ x\ D\ \sigma)\ \sigma)$

$\langle proof \rangle$

**lemma** *imp-nfoldli-mono*[*partial-function-mono*]:

**assumes** [*partial-function-mono*]:  $\bigwedge x\ \sigma.\ mono-Heap\ (\lambda fa.\ f\ fa\ x\ \sigma)$

**shows** *mono-Heap*  $(\lambda x.\ imp-nfoldli\ l\ c\ (f\ x)\ \sigma)$

$\langle proof \rangle$

**lemma** *imp-foreach-mono*[*partial-function-mono*]:

**assumes** [*partial-function-mono*]:  $\bigwedge x\ \sigma.\ mono-Heap\ (\lambda fa.\ f\ fa\ x\ \sigma)$

**shows** *mono-Heap*  $(\lambda x.\ imp-foreach\ tsl\ l\ c\ (f\ x)\ \sigma)$

$\langle proof \rangle$

**lemmas** [sepref-opt-simps] = imp-foreach-def

**definition**

*IS-TO-SORTED-LIST*  $\Omega$   $R_s$   $R_k$   $tsl \equiv (tsl, it\text{-to}\text{-sorted}\text{-list } \Omega) \in (R_s)^k \rightarrow_a \text{list}\text{-assn } R_k$

**lemma** *IS-TO-SORTED-LISTI*:

**assumes**  $(tsl, PR\text{-CONST } (it\text{-to}\text{-sorted}\text{-list } \Omega)) \in (R_s)^k \rightarrow_a \text{list}\text{-assn } R_k$   
**shows** *IS-TO-SORTED-LIST*  $\Omega$   $R_s$   $R_k$   $tsl$   
 ⟨proof⟩

**lemma** *hn-monadic-FOREACH*[sepref-comb-rules]:

**assumes** *INDEP*  $R_k$  *INDEP*  $R_s$  *INDEP*  $R\sigma$   
**assumes** *FR*:  $P \Longrightarrow_t \Gamma * hn\text{-ctxt } R_s s' s * hn\text{-ctxt } R\sigma \sigma' \sigma$   
**assumes** *STL*: *GEN-ALGO*  $tsl$  (*IS-TO-SORTED-LIST*  $ordR$   $R_s$   $R_k$ )  
**assumes** *c-ref*:  $\bigwedge \sigma \sigma'. hn\text{-refine}$   
 ( $\Gamma * hn\text{-ctxt } R_s s' s * hn\text{-ctxt } R\sigma \sigma' \sigma$ )  
 (*c*  $\sigma$ )  
 ( $\Gamma c \sigma' \sigma$ )  
*bool-assn*  
 ( $c' \sigma'$ )

**assumes** *C-FR*:

$\bigwedge \sigma' \sigma. TERM \text{ monadic-FOREACH} \Longrightarrow$   
 $\Gamma c \sigma' \sigma \Longrightarrow_t \Gamma * hn\text{-ctxt } R_s s' s * hn\text{-ctxt } R\sigma \sigma' \sigma$

**assumes** *f-ref*:  $\bigwedge x' x \sigma' \sigma. hn\text{-refine}$

( $\Gamma * hn\text{-ctxt } R_s s' s * hn\text{-ctxt } R_k x' x * hn\text{-ctxt } R\sigma \sigma' \sigma$ )  
 (*f*  $x \sigma$ )  
 ( $\Gamma f x' x \sigma' \sigma$ )  $R\sigma$   
 ( $f' x' \sigma'$ )

**assumes** *F-FR*:  $\bigwedge x' x \sigma' \sigma. TERM \text{ monadic-FOREACH} \Longrightarrow \Gamma f x' x \sigma' \sigma \Longrightarrow_t$   
 $\Gamma * hn\text{-ctxt } R_s s' s * hn\text{-ctxt } Pfx x' x * hn\text{-ctxt } Pfs \sigma' \sigma$

**shows** *hn-refine*

*P*  
 (*imp-foreach*  $tsl$   $s$   $c$   $f$   $\sigma$ )  
 ( $\Gamma * hn\text{-ctxt } R_s s' s * hn\text{-invalid } R\sigma \sigma' \sigma$ )  
 $R\sigma$   
 ((*PR-CONST* (*monadic-FOREACH*  $ordR$   $I$ ))  
 $\$s' \$(\lambda_2 \sigma'. c' \sigma') \$(\lambda_2 x' \sigma'. f' x' \sigma') \$\sigma'$   
 )

⟨proof⟩

**applyS** (*tactic* ⟨*Sepref-Frame.frame-tac* ( $K$  ( $K$  *no-tac*)) @{context} 1⟩)  
**applyS** (*tactic* ⟨*Sepref-Frame.frame-tac* ( $K$  ( $K$  *no-tac*)) @{context} 1⟩)

⟨proof⟩

**lemma** *monadic-nfoldli-assert-aux*:

**assumes** *set*  $l \subseteq S$

**shows** *monadic-nfoldli*  $l\ c\ (\lambda x\ s.\ \text{ASSERT } (x \in S) \gg f\ x\ s)\ s = \text{monadic-nfoldli } l\ c\ f\ s$   
 ⟨proof⟩

**lemmas** *monadic-nfoldli-assert* = *monadic-nfoldli-assert-aux*[*OF order-refl*]

**lemma** *nfoldli-arities*[*sepref-monadify-arity*]:  
 $\text{nfoldli} \equiv \lambda_2 s\ c\ f\ \sigma.\ \text{SP } (\text{nfoldli})\ \$s\ (\lambda_2 x.\ c\ \$x)\ (\lambda_2 x\ \sigma.\ f\ \$x\ \$\sigma)\ \$\sigma$   
 ⟨proof⟩

**lemma** *nfoldli-comb*[*sepref-monadify-comb*]:  
 $\bigwedge s\ c\ f\ \sigma.\ (\text{nfoldli})\ \$s\ (\lambda_2 x.\ c\ x)\ \$f\ \$\sigma \equiv$   
 $\text{Refine-Basic.bind}\ \$(\text{EVAL}\ \$s)\ (\lambda_2 s.\ \text{Refine-Basic.bind}\ \$(\text{EVAL}\ \$\sigma)\ (\lambda_2 \sigma.\$   
 $\text{SP } (\text{monadic-nfoldli})\ \$s\ (\lambda_2 x.\ (\text{EVAL}\ \$ (c\ x))))\ \$f\ \$\sigma$   
 ))  
 ⟨proof⟩

**lemma** *monadic-nfoldli-refine-aux'*:  
**assumes** *SS*:  $set\ l' \subseteq S$   
**assumes** *c-ref*:  $\bigwedge s\ s'.$  *hn-refine*  
 $(\Gamma * \text{hn-ctxt } Rs\ s'\ s)$   
 $(c\ s)$   
 $(\Gamma * \text{hn-ctxt } Rs\ s'\ s)$   
*bool-assn*  
 $(c'\ s')$   
**assumes** *f-ref*:  $\bigwedge x\ x'\ s\ s'.$   $\llbracket x' \in S \rrbracket \implies \text{hn-refine}$   
 $(\Gamma * \text{hn-ctxt } Rl\ x'\ x * \text{hn-ctxt } Rs\ s'\ s)$   
 $(f\ x\ s)$   
 $(\Gamma * \text{hn-ctxt } Rl'\ x'\ x * \text{hn-invalid } Rs\ s'\ s)\ Rs$   
 $(f'\ x'\ s')$

**assumes** *merge*[*sepref-frame-merge-rules*]:  $\bigwedge x\ x'.$   $\text{hn-ctxt } Rl'\ x'\ x \vee_A \text{hn-ctxt } Rl\ x'\ x \implies_t \text{hn-ctxt } Rl''\ x'\ x$   
**notes** [*sepref-frame-merge-rules*] = *merge-sat2*[*OF merge*]

**shows** *hn-refine*  
 $(\Gamma * \text{hn-ctxt } (\text{list-assn } Rl)\ l'\ l * \text{hn-ctxt } Rs\ s'\ s)$   
 $(\text{imp-nfoldli } l\ c\ f\ s)$   
 $(\Gamma * \text{hn-ctxt } (\text{list-assn } Rl'')\ l'\ l * \text{hn-invalid } Rs\ s'\ s)\ Rs$   
 $(\text{monadic-nfoldli } l'\ c'\ f'\ s')$

**apply1** (*subst monadic-nfoldli-assert-aux*[*OF SS,symmetric*])

**applyF** (*induct*  $p \equiv Rl\ l'\ l$ )

arbitrary:  $s s'$   
 rule: *list-assn.induct*)

**applyF** *simp*  
 ⟨*proof*⟩  
**solved**

⟨*proof*⟩

**applyF** (*rule hn-refine-frame*)  
**applyS** *rprems*

**focus**  
 ⟨*proof*⟩  
**solved**  
**solved**

**focus** (*simp add: assn-assoc*)  
 ⟨*proof*⟩  
**solved**

**apply1** (*rule hn-refine-frame*[*OF hnr-RETURN-pass*])  
**applyS** (*tactic* ⟨*Sepref-Frame.frame-tac* ( $K (K \text{ no-tac})$ ) @{*context*} 1⟩)

**apply1** (*simp add: assn-assoc*)  
**applyS** (*tactic* ⟨*Sepref-Frame.merge-tac* ( $K (K \text{ no-tac})$ ) @{*context*} 1⟩)

⟨*proof*⟩  
**solved**  
 ⟨*proof*⟩

**lemma** *hn-monadic-nfoldli-rl'*[*sepref-comb-rules*]:

**assumes** *INDEP Rk INDEP Rσ*

**assumes** *FR*:  $P \implies_t \Gamma * \text{hn-ctxt} (\text{list-assn } Rk) s' s * \text{hn-ctxt } R\sigma \sigma' \sigma$

**assumes** *c-ref*:  $\bigwedge \sigma \sigma'. \text{hn-refine}$

( $\Gamma * \text{hn-ctxt } R\sigma \sigma' \sigma$ )

(*c*  $\sigma$ )

( $\Gamma c \sigma' \sigma$ )

*bool-assn*

(*c'*  $\sigma'$ )

**assumes** *C-FR*:

$\bigwedge \sigma' \sigma. \text{TERM monadic-nfoldli} \implies$

$\Gamma c \sigma' \sigma \implies_t \Gamma * \text{hn-ctxt } R\sigma \sigma' \sigma$

**assumes** *f-ref*:  $\bigwedge x' x \sigma' \sigma. \llbracket x' \in \text{set } s \rrbracket \implies \text{hn-refine}$

( $\Gamma * \text{hn-ctxt } Rk x' x * \text{hn-ctxt } R\sigma \sigma' \sigma$ )

(*f*  $x \sigma$ )

( $\Gamma f x' x \sigma' \sigma$ ) *Rσ*

(*f'*  $x' \sigma'$ )

**assumes** *F-FR*:  $\bigwedge x' x \sigma' \sigma. \text{TERM monadic-nfoldli} \implies \Gamma f x' x \sigma' \sigma \implies_t$   
 $\Gamma * \text{hn-ctxt} Rk' x' x * \text{hn-ctxt} Pf \sigma \sigma' \sigma$

**assumes** *MERGE*:  $\bigwedge x x'. \text{hn-ctxt} Rk' x' x \vee_A \text{hn-ctxt} Rk x' x \implies_t \text{hn-ctxt} Rk''$   
 $x' x$

**shows** *hn-refine*

*P*  
*(imp-nfoldli s c f σ)*  
 $(\Gamma * \text{hn-ctxt} (\text{list-assn } Rk'') s' s * \text{hn-invalid } R\sigma \sigma' \sigma)$   
 $R\sigma$   
 $((\text{monadic-nfoldli})$   
 $\$s' \$(\lambda_2 \sigma'. c' \sigma') \$(\lambda_2 x' \sigma'. f' x' \sigma') \$\sigma'$   
 $)$   
 $\langle \text{proof} \rangle$   
**apply1** (*rule hn-refine-cons-pre*[*OF FR*])  
**apply1** *weaken-hnr-post*  
**applyF** (*rule hn-refine-cons*[*rotated*])  
**applyF** (*rule monadic-nfoldli-refine-aux'*[*OF order-refl*])  
**focus**  
 $\langle \text{proof} \rangle$   
**applyS** (*rule c-ref*)  
**apply1** (*rule entt-trans*[*OF C-FR*[*OF TERMI*]])  
**applyS** (*rule entt-refl*)  
**solved**  
  
**apply1** *weaken-hnr-post*  
**applyF** (*rule hn-refine-cons-post*)  
**applyS** (*rule f-ref; simp*)  
  
**apply1** (*rule entt-trans*[*OF F-FR*[*OF TERMI*]])  
**applyS** (*tactic*  $\langle \text{Sepref-Frame.frame-tac} (K (K \text{no-tac})) @\{\text{context}\} 1 \rangle$ )  
**solved**  
  
 $\langle \text{proof} \rangle$   
**solved**

**applyS** (*tactic*  $\langle \text{Sepref-Frame.frame-tac} (K (K \text{no-tac})) @\{\text{context}\} 1 \rangle$ )  
**applyS** (*tactic*  $\langle \text{Sepref-Frame.frame-tac} (K (K \text{no-tac})) @\{\text{context}\} 1 \rangle$ )  
**applyS** (*tactic*  $\langle \text{Sepref-Frame.frame-tac} (K (K \text{no-tac})) @\{\text{context}\} 1 \rangle$ )  
**solved**  
 $\langle \text{proof} \rangle$

**lemma** *nfoldli-assert*:

**assumes** *set*  $l \subseteq S$

**shows** *nfoldli l c*  $(\lambda x s. \text{ASSERT} (x \in S) \gg f x s) s = \text{nfoldli } l c f s$

$\langle \text{proof} \rangle$

**lemmas** *nfoldli-assert'* = *nfoldli-assert*[*OF order.refl*]



**lemma** *fold-eq-nfoldli*:

$RETURN (fold\ f\ l\ s) = nfoldli\ l\ (\lambda\cdot.\ True) (\lambda x\ s.\ RETURN (f\ x\ s))\ s$   
 $\langle proof \rangle$

**lemma** *fold-eq-nfoldli-assert*:

$RETURN (fold\ f\ l\ s) = nfoldli\ l\ (\lambda\cdot.\ True) (\lambda x\ s.\ ASSERT (x \in set\ l) \gg RETURN (f\ x\ s))\ s$   
 $\langle proof \rangle$

**lemma** *fold-arity[sepref-monadify-arity]*:  $fold \equiv \lambda_2 f\ l\ s.\ SP\ fold\ (\lambda_2 x\ s.\ f\ \$x\ \$s)\ \$l\ \$s$   
 $\langle proof \rangle$

**lemma** *monadify-plain-fold[sepref-monadify-comb]*:

$EVAL\ \$ (fold\ (\lambda_2 x\ s.\ f\ x\ s)\ \$l\ \$s) \equiv (\gg)\ \$ (EVAL\ \$l)\ \$ (\lambda_2 l.\ (\gg)\ \$ (EVAL\ \$s)\ \$ (\lambda_2 s.\ nfoldli\ \$l\ (\lambda_2\cdot.\ True)\ \$ (\lambda_2 x\ s.\ EVAL\ \$ (f\ x\ s))\ \$s))$   
 $\langle proof \rangle$

**lemma** *monadify-plain-fold-old-rl*:

$EVAL\ \$ (fold\ (\lambda_2 x\ s.\ f\ x\ s)\ \$l\ \$s) \equiv (\gg)\ \$ (EVAL\ \$l)\ \$ (\lambda_2 l.\ (\gg)\ \$ (EVAL\ \$s)\ \$ (\lambda_2 s.\ nfoldli\ \$l\ (\lambda_2\cdot.\ True)\ \$ (\lambda_2 x\ s.\ PR-CONST (op-ASSERT-bind (x \in set\ l))\ \$ (EVAL\ \$ (f\ x\ s))\ \$s))$   
 $\langle proof \rangle$

foldli

**lemma** *foldli-eq-nfoldli*:

$RETURN (foldli\ l\ c\ f\ s) = nfoldli\ l\ c\ (\lambda x\ s.\ RETURN (f\ x\ s))\ s$   
 $\langle proof \rangle$

**lemma** *foldli-arities[sepref-monadify-arity]*:

$foldli \equiv \lambda_2 s\ c\ f\ \sigma.\ SP (foldli)\ \$s\ (\lambda_2 x.\ c\ \$x)\ (\lambda_2 x\ \sigma.\ f\ \$x\ \$\sigma)\ \$\sigma$   
 $\langle proof \rangle$

**lemma** *monadify-plain-foldli[sepref-monadify-comb]*:

$EVAL\ \$ (foldli\ \$l\ \$c\ (\lambda_2 x\ s.\ f\ x\ s)\ \$s) \equiv (\gg)\ \$ (EVAL\ \$l)\ \$ (\lambda_2 l.\ (\gg)\ \$ (EVAL\ \$s)\ \$ (\lambda_2 s.\ nfoldli\ \$l\ \$c\ (\lambda_2 x\ s.\ (EVAL\ \$ (f\ x\ s))\ \$s))$   
 $\langle proof \rangle$

## Deforestation

**lemma** *nfoldli-filter-deforestation*:

$nfoldli (filter\ P\ xs)\ c\ f\ s = nfoldli\ xs\ c\ (\lambda x\ s.\ if\ P\ x\ then\ f\ x\ s\ else\ RETURN\ s)\ s$   
 $\langle proof \rangle$

**lemma** *extend-list-of-filtered-set*:

**assumes** [simp, intro!]: *finite S*

**and**  $A$ : *distinct*  $xs'$  set  $xs' = \{x \in S. P x\}$   
**obtains**  $xs$  **where**  $xs' = \text{filter } P \text{ } xs$  *distinct*  $xs$  set  $xs = S$   
 $\langle \text{proof} \rangle$

**lemma** *FOREACHc-filter-deforestation*:  
**assumes**  $FIN[simp, intro!]$ : *finite*  $S$   
**shows**  $(FOREACHc \{x \in S. P x\} c f s)$   
 $= FOREACHc S c (\lambda x s. \text{if } P x \text{ then } f x s \text{ else } RETURN s) s$   
 $\langle \text{proof} \rangle$   
**applyS**  $simp$   
**applyS** ( $simp \text{ add: } nfoldli\text{-filter-deforestation}$ )  
 $\langle \text{proof} \rangle$

**lemma** *FOREACHc-filter-deforestation2*:  
**assumes**  $[simp]$ : *distinct*  $xs$   
**shows**  $(FOREACHc (\text{set } (\text{filter } P \text{ } xs)) c f s)$   
 $= FOREACHc (\text{set } xs) c (\lambda x s. \text{if } P x \text{ then } f x s \text{ else } RETURN s) s$   
 $\langle \text{proof} \rangle$

## 2.2.2 For Loops

**partial-function**  $(heap) \text{ imp-for} :: \text{nat} \Rightarrow \text{nat} \Rightarrow ('a \Rightarrow \text{bool } Heap) \Rightarrow (\text{nat} \Rightarrow 'a \Rightarrow 'a Heap) \Rightarrow 'a \Rightarrow 'a Heap$  **where**  
 $\text{imp-for } i \ u \ c \ f \ s = (\text{if } i \geq u \text{ then } \text{return } s \text{ else } \text{do } \{ctn \leftarrow c \ s; \text{if } ctn \text{ then } f \ i \ s \gg \gg \text{imp-for } (i + 1) \ u \ c \ f \ \text{else } \text{return } s\})$

**declare**  $\text{imp-for.simps}[code]$

**lemma**  $[simp]$ :  
 $i \geq u \implies \text{imp-for } i \ u \ c \ f \ s = \text{return } s$   
 $i < u \implies \text{imp-for } i \ u \ c \ f \ s = \text{do } \{ctn \leftarrow c \ s; \text{if } ctn \text{ then } f \ i \ s \gg \gg \text{imp-for } (i + 1) \ u \ c \ f \ \text{else } \text{return } s\}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{imp-nfoldli-deforest}[sepref-opt-simps]$ :  
 $\text{imp-nfoldli } [l..<u] \ c = \text{imp-for } l \ u \ c$   
 $\langle \text{proof} \rangle$

**partial-function**  $(heap) \text{ imp-for}' :: \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \Rightarrow 'a \Rightarrow 'a Heap) \Rightarrow 'a \Rightarrow 'a Heap$  **where**  
 $\text{imp-for}' \ i \ u \ f \ s = (\text{if } i \geq u \text{ then } \text{return } s \text{ else } f \ i \ s \gg \gg \text{imp-for}' (i + 1) \ u \ f)$

**declare**  $\text{imp-for}'.simps}[code]$

**lemma**  $[simp]$ :  
 $i \geq u \implies \text{imp-for}' \ i \ u \ f \ s = \text{return } s$   
 $i < u \implies \text{imp-for}' \ i \ u \ f \ s = f \ i \ s \gg \gg \text{imp-for}' (i + 1) \ u \ f$   
 $\langle \text{proof} \rangle$

**lemma** *imp-for-imp-for'*[*sepref-opt-simps*]:  
*imp-for* *i u* ( $\lambda$  -. *return True*) = *imp-for'* *i u*  
 ⟨*proof*⟩

**partial-function** (*heap*) *imp-for-down* :: *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  (*'a*  $\Rightarrow$  *bool Heap*)  $\Rightarrow$  (*nat*  
 $\Rightarrow$  *'a*  $\Rightarrow$  *'a Heap*)  $\Rightarrow$  *'a*  $\Rightarrow$  *'a Heap* **where**  
*imp-for-down* *l i c f s* = *do* {  
   *let* *i* = *i* - 1;  
   *ctn*  $\leftarrow$  *c s*;  
   *if* *ctn* *then do* {  
     *s*  $\leftarrow$  *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*[*sepref-opt-simps*]:  
*imp-nfoldli* (*rev* [*l..<u*]) *c* =  
 ( $\lambda$  *f s*. *if* *u*  $\leq$  *l* *then return* *s* *else imp-for-down* *l u c f s*)  
 ⟨*proof*⟩

**context begin**

**private fun** *imp-for-down-induction-scheme* :: *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  *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* ()  
 )

**partial-function** (*heap*) *imp-for-down'* :: *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  (*nat*  $\Rightarrow$  *'a*  $\Rightarrow$  *'a Heap*)  $\Rightarrow$   
*'a*  $\Rightarrow$  *'a Heap* **where**  
*imp-for-down'* *l i f s* = *do* {  
   *let* *i* = *i* - 1;  
   *s*  $\leftarrow$  *f i s*;  
   *if* *i* > *l* *then imp-for-down'* *l i f s* *else return* *s*  
 }

**declare** *imp-for-down'.simps*[*code*]

**lemma** *imp-for-down-no-cond*[*sepref-opt-simps*]:  
*imp-for-down* *l u* ( $\lambda$  -. *return True*) = *imp-for-down'* *l u*  
 ⟨*proof*⟩

**end**

```

lemma imp-for'-rule:
  assumes LESS:  $l \leq u$ 
  assumes PRE:  $P \implies_A I\ l\ s$ 
  assumes STEP:  $\bigwedge i\ s. \llbracket l \leq i; i < u \rrbracket \implies \langle I\ i\ s \rangle\ f\ i\ s\ \langle I\ (i+1) \rangle$ 
  shows  $\langle P \rangle\ \text{imp-for}'\ l\ u\ f\ s\ \langle I\ u \rangle$ 
  \langle proof \rangle

```

This lemma is used to manually convert a fold to a loop over indices.

```

lemma fold-idx-conv:  $\text{fold}\ f\ l\ s = \text{fold}\ (\lambda i. f\ (!i))\ [0..<length\ l]\ s$ 
\langle proof \rangle

```

**end**

## 2.3 Ad-Hoc Solutions

```

theory Sepref-Improper

```

```

imports

```

```

  Sepref-Tool

```

```

  Sepref-HOL-Bindings

```

```

  Sepref-Foreach

```

```

  Sepref-Intf-Util

```

```

begin

```

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 pho-apply :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  'a  $\Rightarrow$  'b where [code-unfold,simp]: pho-apply
f x = f x

```

```

sepref-register pho-apply

```

```

lemmas fold-pho-apply = pho-apply-def[symmetric]

```

```

lemma pure-fun-refine[sepref-fr-rules]: hn-refine

```

```

  (hn-val (A  $\rightarrow$  B) f fi * hn-val A x xi)

```

```

  (return (pho-apply $ fi $ xi))

```

```

  (hn-val (A  $\rightarrow$  B) f fi * hn-val A x xi)

```

```

  (pure B)

```

```

  (RETURN $(pho-apply $ f $ x))

```

```

  \langle proof \rangle

```

```
end  
theory Sepref  
imports  
  Sepref-Tool  
  Sepref-HOL-Bindings  
  
  Sepref-Foreach  
  Sepref-Intf-Util  
  Separation-Logic-Imperative-HOL.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**  $[simp]$ :  $op\text{-}set\text{-}is\text{-}empty\ s \equiv s = \{\}$

**lemma**  $op\text{-}set\text{-}is\text{-}empty\text{-}param[param]$ :  $(op\text{-}set\text{-}is\text{-}empty, op\text{-}set\text{-}is\text{-}empty) \in \langle A \rangle set\text{-}rel \rightarrow bool\text{-}rel$   $\langle proof \rangle$

**definition**  $op\text{-}set\text{-}copy :: 'a\ set \Rightarrow 'a\ set$  **where**  $[simp]$ :  $op\text{-}set\text{-}copy\ s \equiv s$

**context**

**notes**  $[simp] = IS\text{-}LEFT\text{-}UNIQUE\text{-}def$   
**begin**

**sepref-decl-op**  $(no\text{-}def)$   $set\text{-}copy$ :  $op\text{-}set\text{-}copy :: \langle A \rangle set\text{-}rel \rightarrow \langle A \rangle set\text{-}rel$  **where**  $A = Id$   $\langle proof \rangle$

**sepref-decl-op**  $set\text{-}empty$ :  $\{\} :: \langle A \rangle set\text{-}rel$   $\langle proof \rangle$

**sepref-decl-op**  $(no\text{-}def)$   $set\text{-}is\text{-}empty$ :  $op\text{-}set\text{-}is\text{-}empty :: \langle A \rangle set\text{-}rel \rightarrow bool\text{-}rel$   $\langle proof \rangle$

**sepref-decl-op**  $set\text{-}member$ :  $(\in) :: A \rightarrow \langle A \rangle set\text{-}rel \rightarrow bool\text{-}rel$  **where**  $IS\text{-}LEFT\text{-}UNIQUE\ A\ IS\text{-}RIGHT\text{-}UNIQUE\ A$   $\langle proof \rangle$

**sepref-decl-op**  $set\text{-}insert$ :  $Set.insert :: A \rightarrow \langle A \rangle set\text{-}rel \rightarrow \langle A \rangle set\text{-}rel$  **where**  $IS\text{-}RIGHT\text{-}UNIQUE\ A$   $\langle proof \rangle$

**sepref-decl-op**  $set\text{-}delete$ :  $\lambda x\ s.\ s - \{x\} :: A \rightarrow \langle A \rangle set\text{-}rel \rightarrow \langle A \rangle set\text{-}rel$

**where** *IS-LEFT-UNIQUE A IS-RIGHT-UNIQUE A*  $\langle proof \rangle$   
**sepref-decl-op** *set-union*:  $(\cup) :: \langle A \rangle set-rel \rightarrow \langle A \rangle set-rel \rightarrow \langle A \rangle set-rel$   $\langle proof \rangle$   
**sepref-decl-op** *set-inter*:  $(\cap) :: \langle A \rangle set-rel \rightarrow \langle A \rangle set-rel \rightarrow \langle A \rangle set-rel$  **where** *IS-LEFT-UNIQUE A IS-RIGHT-UNIQUE A*  $\langle proof \rangle$   
**sepref-decl-op** *set-diff*:  $(-) ::- set \Rightarrow - :: \langle A \rangle set-rel \rightarrow \langle A \rangle set-rel \rightarrow \langle A \rangle set-rel$   
**where** *IS-LEFT-UNIQUE A IS-RIGHT-UNIQUE A*  $\langle proof \rangle$   
**sepref-decl-op** *set-subseteq*:  $(\subseteq) :: \langle A \rangle set-rel \rightarrow \langle A \rangle set-rel \rightarrow bool-rel$  **where** *IS-LEFT-UNIQUE A IS-RIGHT-UNIQUE A*  $\langle proof \rangle$   
**sepref-decl-op** *set-subset*:  $(\subset) :: \langle A \rangle set-rel \rightarrow \langle A \rangle set-rel \rightarrow bool-rel$  **where** *IS-LEFT-UNIQUE A IS-RIGHT-UNIQUE A*  $\langle proof \rangle$

**sepref-decl-op** *set-pick*:  $RES :: [\lambda s. s \neq \{\}]_f \langle K \rangle set-rel \rightarrow K$   $\langle proof \rangle$

**sepref-decl-op** *set-size*:  $(card) :: \langle A \rangle set-rel \rightarrow nat-rel$  **where** *IS-LEFT-UNIQUE A IS-RIGHT-UNIQUE A*  $\langle proof \rangle$

**end**

### 3.1.2 Patterns

**lemma** *pat-set*[*def-pat-rules*]:

$\{\} \equiv op-set-empty$   
 $(\in) \equiv op-set-member$   
 $Set.insert \equiv op-set-insert$   
 $(\cup) \equiv op-set-union$   
 $(\cap) \equiv op-set-inter$   
 $(-) \equiv op-set-diff$   
 $(\subseteq) \equiv op-set-subseteq$   
 $(\subset) \equiv op-set-subset$   
 $\langle proof \rangle$

**lemma** *pat-set2*[*pat-rules*]:

$(=) \$s\{\} \equiv op-set-is-empty\$s$   
 $(=) \{\}\$s \equiv op-set-is-empty\$s$   
 $(-) \$s\$(Set.insert\$x\{\}) \equiv op-set-delete\$x\$s$   
 $SPEC\$(\lambda_2 x. (\in) \$x\$s) \equiv op-set-pick s$   
 $RES\$s \equiv op-set-pick s$   
 $\langle proof \rangle$

**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* :: 'a set

```

lemma fold-custom-empty:
  {} = op-custom-empty
  op-set-empty = op-custom-empty
  mop-set-empty = RETURN op-custom-empty
  ⟨proof⟩
end

end

```

## 3.2 Sets by Lists that Own their Elements

```

theory IICF-List-SetO
imports ../Intf/IICF-Set
begin

```

Minimal implementation, only supporting a few operations

```

definition lso-assn  $A \equiv hr\text{-comp } (list\text{-assn } A) (br\text{ set } (\lambda\cdot. True))$ 
lemmas [fcomp-norm-unfold] = lso-assn-def[symmetric]
lemma lso-is-pure[safe-constraint-rules]:  $is\text{-pure } A \implies is\text{-pure } (lso\text{-assn } A)$ 
  ⟨proof⟩

lemma lso-empty-aref:  $(uncurry0 (RETURN []), uncurry0 (RETURN op\text{-set-empty}))$ 
  ∈  $unit\text{-rel} \rightarrow_f \langle br\text{ set } (\lambda\cdot. True) \rangle nres\text{-rel}$ 
  ⟨proof⟩

lemma lso-ins-aref:  $(uncurry (RETURN oo ((\#) )), uncurry (RETURN oo op\text{-set-insert}))$ 
  ∈  $Id \times_r br\text{ set } (\lambda\cdot. True) \rightarrow_f \langle br\text{ set } (\lambda\cdot. True) \rangle nres\text{-rel}$ 
  ⟨proof⟩

sepref-decl-impl (no-register) lso-empty:  $hn\text{-Nil}[to\text{-href}]$  uses lso-empty-aref
  ⟨proof⟩
definition [simp]:  $op\text{-lso-empty} \equiv op\text{-set-empty}$ 
lemma lso-fold-custom-empty:
  {} = op-lso-empty
  op-set-empty = op-lso-empty
  ⟨proof⟩
lemmas [sepref-fr-rules] = lso-empty-hnr[folded op-lso-empty-def]

sepref-decl-impl lso-insert:  $hn\text{-Cons}[to\text{-href}]$  uses lso-ins-aref ⟨proof⟩

thm  $hn\text{-Cons}[FCOMP\ lso\text{-ins-aref}]$ 

definition [simp]:  $op\text{-lso-bex } P\ S \equiv \exists x \in S. P\ x$ 
lemma fold-lso-bex:  $Bex \equiv \lambda s\ P. op\text{-lso-bex } P\ s$  ⟨proof⟩

```



**definition** [simp]:  $mop\text{-}lso\text{-}bex\ P\ S \equiv ASSERT\ (\forall x \in S. \exists y. P\ x = RETURN\ y) \gg RETURN\ (\exists x \in S. P\ x = RETURN\ True)$

**lemma** *op-mop-lso-bex*:  $RETURN\ (op\text{-}lso\text{-}bex\ P\ S) = mop\text{-}lso\text{-}bex\ (RETURN\ o\ P)\ S$  *<proof>*

**sempref-register** *op-lso-bex*

**lemma** *lso-bex-arity*[*sempref-monadify-arity*]:

$op\text{-}lso\text{-}bex \equiv \lambda_2 P\ s. SP\ op\text{-}lso\text{-}bex\ (\lambda_2 x. P\ \$x)\ \$s$  *<proof>*

**lemma** *op-lso-bex-monadify*[*sempref-monadify-comb*]:

$EVAL\ \$ (op\text{-}lso\text{-}bex\ (\lambda_2 x. P\ x)\ \$s) \equiv (\gg) \$ (EVAL\ \$s)\ (\lambda_2 s. mop\text{-}lso\text{-}bex\ (\lambda_2 x. EVAL\ \$ P\ x)\ \$s)$  *<proof>*

**definition** *lso-abex*  $P\ l \equiv nfoldli\ l\ (Not)\ (\lambda x\ -. P\ x)\ False$

**lemma** *lso-abex-to-set*:  $lso\text{-}abex\ P\ l \leq mop\text{-}lso\text{-}bex\ P\ (set\ l)$  *<proof>*

**applyS** *simp*

**applyS** (*clarsimp simp add: pw-le-iff refine-pw-simps; blast*)

*<proof>*

**locale** *lso-bex-impl-loc* =

**fixes** *Pi* **and** *P* :: 'a  $\Rightarrow$  bool nres

**fixes** *li* :: 'c list **and** *l* :: 'a list

**fixes** *A* :: 'a  $\Rightarrow$  'c  $\Rightarrow$  assn

**fixes** *F* :: assn

**assumes** *Prl*:  $\bigwedge x\ xi. \llbracket x \in set\ l \rrbracket \Longrightarrow hn\text{-refine}\ (F\ *\ hn\text{-ctxt}\ A\ x\ xi)\ (Pi\ xi)\ (F\ *\ hn\text{-ctxt}\ A\ x\ xi)\ bool\text{-assn}\ (P\ x)$

**begin**

**sempref-register** *l*

**sempref-register** *P*

**lemma** [*sempref-comb-rules*]:

**assumes**  $\Gamma \Longrightarrow_t F' * F * hn\text{-ctxt}\ A\ x\ xi$

**assumes**  $x \in set\ l$

**shows**  $hn\text{-refine}\ \Gamma\ (Pi\ xi)\ (F' * F * hn\text{-ctxt}\ A\ x\ xi)\ bool\text{-assn}\ (P\ \$x)$

*<proof>*

**schematic-goal** *lso-bex-impl*:

$hn\text{-refine}\ (hn\text{-ctxt}\ (list\text{-assn}\ A)\ l\ li * F)\ (?c)\ (F * hn\text{-ctxt}\ (list\text{-assn}\ A)\ l\ li)\ bool\text{-assn}\ (lso\text{-abex}\ P\ l)$

*<proof>*

**end**

**concrete-definition** *lso-bex-impl* **uses** *lso-bex-impl-loc.lso-bex-impl*

**lemma** *hn-lso-bex*[*sempref-prep-comb-rule,sempref-comb-rules*]:

```

assumes FR:  $\Gamma \Longrightarrow_t \text{hn-ctxt } (lso\text{-assn } A) s li * F$ 
assumes Prl:  $\bigwedge x xi. \llbracket x \in s \rrbracket \Longrightarrow \text{hn-refine } (F * \text{hn-ctxt } A x xi) (Pi xi) (F * \text{hn-ctxt } A x xi) \text{bool-assn } (P x)$ 
notes [simp del] = mop-lso-bex-def
shows  $\text{hn-refine } \Gamma (lso\text{-bex-impl } Pi li) (F * \text{hn-ctxt } (lso\text{-assn } A) s li) \text{bool-assn } (mop\text{-lso-bex } (\lambda_2 x. P x) s)$ 
  <proof>
applyS (simp add: hn-ctxt-def; rule entt-refl)
apply1 unfold-locales apply1 (rule Prl') applyS simp
applyS (sep-auto intro!: enttI simp: hn-ctxt-def)
applyS (rule entt-refl)
  <proof>

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')
  <proof>

```

```

lemma rel-mset-single:
assumes A x y
shows rel-mset A {#x#} {#y#}
  <proof>

```

```

lemma rel-mset-Minus:
assumes BIU: bi-unique A
shows  $\llbracket \text{rel-mset } A m n; A x y \rrbracket \Longrightarrow \text{rel-mset } A (m - \{#x\}) (n - \{#y\})$ 
  <proof>

```

```

lemma rel-mset-Minus-gen:
assumes BIU: bi-unique A
assumes rel-mset A m m'
assumes rel-mset A n n'
shows rel-mset A (m-n) (m'-n')
  <proof>

```

```

lemma pcr-count:
assumes bi-unique A
shows rel-fun (rel-mset A) (rel-fun A (=)) count count

```

$\langle proof \rangle$

### 3.3.2 Parametricity Setup

**definition**  $[to-relAPP]$ :  $mset-rel\ A \equiv p2rel\ (rel-mset\ (rel2p\ A))$

**lemma**  $rel2p-mset[rel2p]$ :  $rel2p\ (\langle A \rangle mset-rel) = rel-mset\ (rel2p\ A)$   
 $\langle proof \rangle$

**lemma**  $p2re-mset[p2rel]$ :  $p2rel\ (rel-mset\ A) = \langle p2rel\ A \rangle mset-rel$   
 $\langle proof \rangle$

**lemma**  $mset-rel-empty[simp]$ :  
 $(a, \{\#\}) \in \langle A \rangle mset-rel \longleftrightarrow a = \{\#\}$   
 $(\{\#\}, b) \in \langle A \rangle mset-rel \longleftrightarrow b = \{\#\}$   
 $\langle proof \rangle$

**lemma**  $param-mset-empty[param]$ :  $(\{\#\}, \{\#\}) \in \langle A \rangle mset-rel$   
 $\langle proof \rangle$

**lemma**  $param-mset-Plus[param]$ :  $((+), (+)) \in \langle A \rangle mset-rel \rightarrow \langle A \rangle mset-rel \rightarrow \langle A \rangle mset-rel$   
 $\langle proof \rangle$

**lemma**  $param-mset-add[param]$ :  $(add-mset, add-mset) \in A \rightarrow \langle A \rangle mset-rel \rightarrow \langle A \rangle mset-rel$   
 $\langle proof \rangle$

**lemma**  $param-mset-minus[param]$ :  $\llbracket single-valued\ A; single-valued\ (A^{-1}) \rrbracket$   
 $\implies ((-), (-)) \in \langle A \rangle mset-rel \rightarrow \langle A \rangle mset-rel \rightarrow \langle A \rangle mset-rel$   
 $\langle proof \rangle$

**lemma**  $param-count[param]$ :  $\llbracket single-valued\ A; single-valued\ (A^{-1}) \rrbracket \implies (count, count) \in \langle A \rangle mset-rel$   
 $\rightarrow A \rightarrow nat-rel$   
 $\langle proof \rangle$

**lemma**  $param-set-mset[param]$ :  
**shows**  $(set-mset, set-mset) \in \langle A \rangle mset-rel \rightarrow \langle A \rangle set-rel$   
 $\langle proof \rangle$

**definition**  $[simp]$ :  $mset-is-empty\ m \equiv m = \{\#\}$

**lemma**  $mset-is-empty-param[param]$ :  $(mset-is-empty, mset-is-empty) \in \langle A \rangle mset-rel$   
 $\rightarrow bool-rel$   
 $\langle proof \rangle$

### 3.3.3 Operations

**sepref-decl-op** *mset-empty*:  $\{\#\} :: \langle A \rangle \text{mset-rel} \langle \text{proof} \rangle$

**sepref-decl-op** *mset-is-empty*:  $\lambda m. m = \{\#\} :: \langle A \rangle \text{mset-rel} \rightarrow \text{bool-rel} \langle \text{proof} \rangle$

**sepref-decl-op** *mset-insert*:  $\text{add-mset} :: A \rightarrow \langle A \rangle \text{mset-rel} \rightarrow \langle A \rangle \text{mset-rel} \langle \text{proof} \rangle$

**sepref-decl-op** *mset-delete*:  $\lambda x m. m - \{\#x\# \} :: A \rightarrow \langle A \rangle \text{mset-rel} \rightarrow \langle A \rangle \text{mset-rel}$   
**where** *single-valued A single-valued (A<sup>-1</sup>)*  $\langle \text{proof} \rangle$

**sepref-decl-op** *mset-plus*:  $(+) :: \text{multiset} \Rightarrow - :: \langle A \rangle \text{mset-rel} \rightarrow \langle A \rangle \text{mset-rel} \rightarrow \langle A \rangle \text{mset-rel} \langle \text{proof} \rangle$

**sepref-decl-op** *mset-minus*:  $(-) :: \text{multiset} \Rightarrow - :: \langle A \rangle \text{mset-rel} \rightarrow \langle A \rangle \text{mset-rel} \rightarrow \langle A \rangle \text{mset-rel}$   
**where** *single-valued A single-valued (A<sup>-1</sup>)*  $\langle \text{proof} \rangle$

**sepref-decl-op** *mset-contains*:  $(\in\#) :: A \rightarrow \langle A \rangle \text{mset-rel} \rightarrow \text{bool-rel}$   
**where** *single-valued A single-valued (A<sup>-1</sup>)*  $\langle \text{proof} \rangle$

**sepref-decl-op** *mset-count*:  $\lambda x y. \text{count } y \ x :: A \rightarrow \langle A \rangle \text{mset-rel} \rightarrow \text{nat-rel}$   
**where** *single-valued A single-valued (A<sup>-1</sup>)*  $\langle \text{proof} \rangle$

**sepref-decl-op** *mset-pick*:  $\lambda m. \text{SPEC } (\lambda(x,m'). m = \{\#x\# \} + m') ::$   
 $[\lambda m. m \neq \{\#\}]_f \langle A \rangle \text{mset-rel} \rightarrow A \times_r \langle A \rangle \text{mset-rel}$   
 $\langle \text{proof} \rangle$   
**apply1** (*rule ccontr; clarsimp*)  
**applyS** (*metis mset-rel-invL rel2p-def rel2p-mset union-ac(2)*)  
**applyS** *parametricity*  
 $\langle \text{proof} \rangle$

### 3.3.4 Patterns

**lemma** [*def-pat-rules*]:

$\{\#\} \equiv \text{op-mset-empty}$   
 $\text{add-mset} \equiv \text{op-mset-insert}$   
 $(=) \ \$b\ \{\#\} \equiv \text{op-mset-is-empty}\ \$b$   
 $(=) \ \{\#\} \ \$b \equiv \text{op-mset-is-empty}\ \$b$   
 $(+) \ \$a\ \$b \equiv \text{op-mset-plus}\ \$a\ \$b$   
 $(-) \ \$a\ \$b \equiv \text{op-mset-minus}\ \$a\ \$b$   
 $\langle \text{proof} \rangle$

**lemma** [*def-pat-rules*]:

$(+) \ \$b\ (\text{add-mset}\ \$x\ \{\#\}) \equiv \text{op-mset-insert}\ \$x\ \$b$   
 $(+) \ (\text{add-mset}\ \$x\ \{\#\})\ \$b \equiv \text{op-mset-insert}\ \$x\ \$b$

```

(-) $b$(add-mset$x${#}) ≡ op-mset-delete$x$b
(<) $0$(count$a$x) ≡ op-mset-contains$x$a
(∈) $x$(set-mset$a) ≡ op-mset-contains$x$a
⟨proof⟩

```

```

locale mset-custom-empty =
  fixes rel empty and op-custom-empty :: 'a multiset
  assumes customize-hnr-aux: (uncurry0 empty,uncurry0 (RETURN (op-mset-empty::'a
multiset))) ∈ unit-assnk →a rel
  assumes op-custom-empty-def: op-custom-empty = op-mset-empty
begin
  sepref-register op-custom-empty :: 'ax multiset

  lemma fold-custom-empty:
    {#} = op-custom-empty
    op-mset-empty = op-custom-empty
    mop-mset-empty = RETURN op-custom-empty
    ⟨proof⟩

  lemmas custom-hnr[sepref-fr-rules] = customize-hnr-aux[folded op-custom-empty-def]
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.

```

context
  fixes prio:: 'a ⇒ 'b::linorder
begin
  definition mop-prio-pop-min b = ASSERT (b≠{#}) ≫ SPEC (λ(v,b').
    v ∈# b
    ∧ b'=b - {#v#}
    ∧ (∀ v'∈set-mset b. prio v ≤ prio v'))

  definition mop-prio-peek-min b ≡ ASSERT (b≠{#}) ≫ SPEC (λv.

```

$v \in \# b$   
 $\wedge (\forall v' \in \text{set-mset } b. \text{prio } v \leq \text{prio } v')$

**end**

**lemma** *param-mop-prio-pop-min*[*param*]:  
**assumes** [*param*]:  $(\text{prio}', \text{prio}) \in A \rightarrow B$   
**assumes** [*param*]:  $((\leq), (\leq)) \in B \rightarrow B \rightarrow \text{bool-rel}$   
**shows**  $(\text{mop-prio-pop-min } \text{prio}', \text{mop-prio-pop-min } \text{prio}) \in \langle A \rangle \text{mset-rel} \rightarrow \langle A \rangle \times_r \langle A \rangle \text{mset-rel} \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**lemma** *param-mop-prio-peek-min*[*param*]:  
**assumes** [*param*]:  $(\text{prio}', \text{prio}) \in A \rightarrow B$   
**assumes** [*param*]:  $((\leq), (\leq)) \in B \rightarrow B \rightarrow \text{bool-rel}$   
**shows**  $(\text{mop-prio-peek-min } \text{prio}', \text{mop-prio-peek-min } \text{prio}) \in \langle A \rangle \text{mset-rel} \rightarrow \langle A \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**context fixes** *prio* :: 'a  $\Rightarrow$  'b::linorder **and** *A* :: ('a  $\times$  'a) set **begin**  
**sepref-decl-op** (no-def,no-mop) *prio-pop-min*:  
*PR-CONST* (*mop-prio-pop-min prio*) ::  $\langle A \rangle \text{mset-rel} \rightarrow_f \langle A \times_r \langle A \rangle \text{mset-rel} \rangle \text{nres-rel}$   
**where** *IS-BELOW-ID A*  
 $\langle \text{proof} \rangle$   
**sepref-decl-op** (no-def,no-mop) *prio-peek-min*:  
*PR-CONST* (*mop-prio-peek-min prio*) ::  $\langle A \rangle \text{mset-rel} \rightarrow_f \langle A \rangle \text{nres-rel}$   
**where** *IS-BELOW-ID A*  
 $\langle \text{proof} \rangle$   
**end**

### 3.4.2 Patterns

**lemma** [*def-pat-rules*]:  
*mop-prio-pop-min*\$*prio*  $\equiv$  *UNPROTECT* (*mop-prio-pop-min prio*)  
*mop-prio-peek-min*\$*prio*  $\equiv$  *UNPROTECT* (*mop-prio-peek-min prio*)  
 $\langle \text{proof} \rangle$

**end**

## 3.5 Multisets by Lists

**theory** *IICF-List-Mset*  
**imports** ../Intf/IICF-Multiset  
**begin**

### 3.5.1 Abstract Operations

**definition**  $list\text{-}mset\text{-}rel \equiv br\ mset\ (\lambda\cdot.\ True)$

**lemma**  $lms\text{-}empty\text{-}aref: (\ [],\ op\text{-}mset\text{-}empty) \in list\text{-}mset\text{-}rel$   
 $\langle proof \rangle$

**lemma**  $lms\text{-}is\text{-}empty\text{-}aref: (is\text{-}Nil,\ op\text{-}mset\text{-}is\text{-}empty) \in list\text{-}mset\text{-}rel \rightarrow bool\text{-}rel$   
 $\langle proof \rangle$

**lemma**  $lms\text{-}insert\text{-}aref: ((\ #),\ op\text{-}mset\text{-}insert) \in Id \rightarrow list\text{-}mset\text{-}rel \rightarrow list\text{-}mset\text{-}rel$   
 $\langle proof \rangle$

**lemma**  $lms\text{-}union\text{-}aref: ((\ @),\ op\text{-}mset\text{-}plus) \in list\text{-}mset\text{-}rel \rightarrow list\text{-}mset\text{-}rel \rightarrow list\text{-}mset\text{-}rel$   
 $\langle proof \rangle$

**lemma**  $lms\text{-}pick\text{-}aref: (\lambda x\#l \Rightarrow RETURN\ (x,l),\ mop\text{-}mset\text{-}pick) \in list\text{-}mset\text{-}rel$   
 $\rightarrow \langle Id \times_r\ list\text{-}mset\text{-}rel \rangle nres\text{-}rel$

$\langle proof \rangle$   
**apply1**  $(refine\text{-}vcg\ nres\text{-}relI\ fun\text{-}relI)$   
**apply1**  $(clarsimp\ simp: in\text{-}br\text{-}conv\ neq\text{-}Nil\text{-}conv)$   
**apply1**  $(refine\text{-}vcg\ RETURN\text{-}SPEC\text{-}refine)$   
**applyS**  $(clarsimp\ simp: in\text{-}br\text{-}conv\ algebra\text{-}simps)$   
 $\langle proof \rangle$

**definition**  $list\text{-}contains\ x\ l \equiv list\text{-}ex\ ((=)\ x)\ l$

**lemma**  $lms\text{-}contains\text{-}aref: (list\text{-}contains,\ op\text{-}mset\text{-}contains) \in Id \rightarrow list\text{-}mset\text{-}rel$   
 $\rightarrow bool\text{-}rel$   
 $\langle proof \rangle$

**fun**  $list\text{-}remove1 :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$  **where**  
 $list\text{-}remove1\ x\ [] = []$   
 $| list\text{-}remove1\ x\ (y\#\ys) = (if\ x=y\ then\ ys\ else\ y\#\ list\text{-}remove1\ x\ ys)$

**lemma**  $mset\text{-}list\text{-}remove1[simp]: mset\ (list\text{-}remove1\ x\ l) = mset\ l - \{x\}$   
 $\langle proof \rangle$   
**applyS**  $simp$   
 $\langle proof \rangle$

**lemma**  $lms\text{-}remove\text{-}aref: (list\text{-}remove1,\ op\text{-}mset\text{-}delete) \in Id \rightarrow list\text{-}mset\text{-}rel \rightarrow list\text{-}mset\text{-}rel$   
 $\langle proof \rangle$

**fun**  $list\text{-}count :: 'a \Rightarrow 'a\ list \Rightarrow nat$  **where**  
 $list\text{-}count\ -\ [] = 0$   
 $| list\text{-}count\ x\ (y\#\ys) = (if\ x=y\ then\ 1 + list\text{-}count\ x\ ys\ else\ list\text{-}count\ x\ ys)$

**lemma** *mset-list-count*[simp]: *list-count x ys = count (mset ys) x*  
 ⟨proof⟩

**lemma** *lms-count-aref*:  $(\text{list-count}, \text{op-mset-count}) \in \text{Id} \rightarrow \text{list-mset-rel} \rightarrow \text{nat-rel}$   
 ⟨proof⟩

**definition** *list-remove-all* :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  'a list **where**

*list-remove-all xs ys*  $\equiv$  *fold list-remove1 ys xs*

**lemma** *list-remove-all-mset*[simp]: *mset (list-remove-all xs ys) = mset xs - mset ys*  
 ⟨proof⟩

**lemma** *lms-minus-aref*:  $(\text{list-remove-all}, \text{op-mset-minus}) \in \text{list-mset-rel} \rightarrow \text{list-mset-rel} \rightarrow \text{list-mset-rel}$   
 ⟨proof⟩

### 3.5.2 Declaration of Implementations

**definition** *list-mset-assn A*  $\equiv$  *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)* ⟨proof⟩

**sepref-decl-impl** (*no-register*) *lms-empty*: *lms-empty-aref*[sepref-param] ⟨proof⟩

**definition** [simp]: *op-list-mset-empty*  $\equiv$  *op-mset-empty*

**lemma** *lms-fold-custom-empty*:

{#} = *op-list-mset-empty*

*op-mset-empty* = *op-list-mset-empty*

⟨proof⟩

**sepref-register** *op-list-mset-empty*

**lemmas** [sepref-fr-rules] = *lms-empty-hnr*[folded *op-list-mset-empty-def*]

**sepref-decl-impl** *lms-is-empty*: *lms-is-empty-aref*[sepref-param] ⟨proof⟩

**sepref-decl-impl** *lms-insert*: *lms-insert-aref*[sepref-param] ⟨proof⟩

**sepref-decl-impl** *lms-union*: *lms-union-aref*[sepref-param] ⟨proof⟩

**lemma** *lms-pick-aref'*:

$(\lambda x \# l \Rightarrow \text{return } (x, l), \text{mop-mset-pick}) \in (\text{pure list-mset-rel})^k \rightarrow_a \text{prod-assn id-assn (pure list-mset-rel)}$   
 ⟨proof⟩

**sepref-decl-impl** (*ismop*) *lms-pick*: *lms-pick-aref'* ⟨proof⟩

**sepref-decl-impl** *lms-contains*: *lms-contains-aref*[sepref-param] ⟨proof⟩

**sepref-decl-impl** *lms-remove*: *lms-remove-aref*[sepref-param] ⟨proof⟩

**sepref-decl-impl** *lms-count*: *lms-count-aref*[sepref-param] ⟨proof⟩

**sepref-decl-impl** *lms-minus*: *lms-minus-aref*[sepref-param] ⟨proof⟩



**end**  
**theory** *IICF-List-MsetO*  
**imports** *../Intf/IICF-Multiset*  
**begin**

**definition** *lmsso-assn*  $A \equiv \text{hr-comp } (\text{list-assn } A) (\text{br mset } (\lambda-. \text{True}))$   
**lemmas** *[fcomp-norm-unfold]* = *lmsso-assn-def[symmetric]*

**lemma** *lmsso-is-pure[safe-constraint-rules]*: *is-pure*  $A \implies \text{is-pure } (\text{lmsso-assn } A)$   
 $\langle \text{proof} \rangle$

**lemma** *lmsso-empty-aref*:  $(\text{uncurry0 } (\text{RETURN } []), \text{uncurry0 } (\text{RETURN } \text{op-mset-empty}))$   
 $\in \text{unit-rel} \rightarrow_f \langle \text{br mset } (\lambda-. \text{True}) \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**lemma** *lmsso-is-empty-aref*:  $(\text{RETURN } o \text{List.null}, \text{RETURN } o \text{op-mset-is-empty})$   
 $\in \text{br mset } (\lambda-. \text{True}) \rightarrow_f \langle \text{bool-rel} \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**lemma** *lmsso-insert-aref*:  $(\text{uncurry } (\text{RETURN } oo (\#)), \text{uncurry } (\text{RETURN } oo \text{op-mset-insert})) \in (\text{Id } \times_r \text{br mset } (\lambda-. \text{True})) \rightarrow_f \langle \text{br mset } (\lambda-. \text{True}) \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**definition** *[simp]*: *hd-tl*  $l \equiv (\text{hd } l, \text{tl } l)$

**lemma** *hd-tl-opt[sepref-opt-simps]*: *hd-tl*  $l = (\text{case } l \text{ of } (x\#xs) \Rightarrow (x,xs) \mid - \Rightarrow \text{CODE-ABORT } (\lambda-. (\text{hd } l, \text{tl } l)))$   
 $\langle \text{proof} \rangle$

**lemma** *lmsso-pick-aref*:  $(\text{RETURN } o \text{hd-tl}, \text{op-mset-pick}) \in [\lambda m. m \neq \{\#\}]_f \text{br mset } (\lambda-. \text{True}) \rightarrow \langle \text{Id } \times_r \text{br mset } (\lambda-. \text{True}) \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**lemma** *hd-tl-hnr*:  $(\text{return } o \text{hd-tl}, \text{RETURN } o \text{hd-tl}) \in [\lambda l. \neg \text{is-Nil } l]_a (\text{list-assn } A)^d \rightarrow \text{prod-assn } A (\text{list-assn } A)$   
 $\langle \text{proof} \rangle$

**sepref-decl-impl** *(no-register)* *lmsso-empty*: *hn-Nil[to-hfref]* **uses** *lmsso-empty-aref*  
 $\langle \text{proof} \rangle$

**definition** *[simp]*: *op-lmsso-empty*  $\equiv \text{op-mset-empty}$   
**sepref-register** *op-lmsso-empty*  
**lemma** *lmsso-fold-custom-empty*:  
 $\{\#\} = \text{op-lmsso-empty}$

```

    op-mset-empty = op-lmso-empty
    mop-mset-empty = RETURN op-lmso-empty
    <proof>
lemmas [sepref-fr-rules] = lmso-empty-hnr[folded op-lmso-empty-def]

```

```

lemma list-null-hnr: (return o List.null, RETURN o List.null) ∈ (list-assn A)k
→a bool-assn
    <proof>

```

```

sepref-decl-impl lmso-is-empty: list-null-hnr uses lmso-is-empty-aref <proof>

```

```

sepref-decl-impl lmso-insert: hn-Cons[to-hfref] uses lmso-insert-aref <proof>

```

```

context notes [simp] = in-br-conv and [split] = list.splits begin

```

Dummy lemma, to exploit *sepref-decl-impl* automation without parametricity stuff.

```

    private lemma op-mset-pick-dummy-param: (op-mset-pick, op-mset-pick) ∈ Id
→f <Id>nres-rel
    <proof>

```

```

    sepref-decl-impl lmso-pick: hd-tl-hnr[FCOMP lmso-pick-aref] uses op-mset-pick-dummy-param
    <proof>
end

```

```

end

```

```

theory IICF-List

```

```

imports

```

```

    ../Sepref

```

```

    List-Index.List-Index

```

```

begin

```

```

lemma param-index[param]:

```

```

    [[single-valued A; single-valued (A-1)] ⇒ (index, index) ∈ <A>list-rel → A →
nat-rel
    <proof>

```

### 3.5.3 Swap two elements of a list, by index

```

definition swap l i j ≡ l[i := !j, j:=!i]

```

```

lemma swap-nth[simp]: [[i < length l; j < length l; k < length l] ⇒

```

```

    swap l i j!k = (
        if k=i then !j
        else if k=j then !i
        else !k
    )

```

$\langle \text{proof} \rangle$

**lemma** *swap-set*[simp]:  $\llbracket i < \text{length } l; j < \text{length } l \rrbracket \implies \text{set } (\text{swap } l \ i \ j) = \text{set } l$   
 $\langle \text{proof} \rangle$

**lemma** *swap-multiset*[simp]:  $\llbracket i < \text{length } l; j < \text{length } l \rrbracket \implies \text{mset } (\text{swap } l \ i \ j) = \text{mset } l$   
 $\langle \text{proof} \rangle$

**lemma** *swap-length*[simp]:  $\text{length } (\text{swap } l \ i \ j) = \text{length } l$   
 $\langle \text{proof} \rangle$

**lemma** *swap-same*[simp]:  $\text{swap } l \ i \ i = l$   
 $\langle \text{proof} \rangle$

**lemma** *distinct-swap*[simp]:  
 $\llbracket i < \text{length } l; j < \text{length } l \rrbracket \implies \text{distinct } (\text{swap } l \ i \ j) = \text{distinct } l$   
 $\langle \text{proof} \rangle$

**lemma** *map-swap*:  $\llbracket i < \text{length } l; j < \text{length } l \rrbracket$   
 $\implies \text{map } f \ (\text{swap } l \ i \ j) = \text{swap } (\text{map } f \ l) \ i \ j$   
 $\langle \text{proof} \rangle$

**lemma** *swap-param*[param]:  $\llbracket i < \text{length } l; j < \text{length } l; (l', l) \in \langle A \rangle \text{list-rel}; (i', i) \in \text{nat-rel}; (j', j) \in \text{nat-rel} \rrbracket$   
 $\implies (\text{swap } l' \ i' \ j', \text{swap } l \ i \ j) \in \langle A \rangle \text{list-rel}$   
 $\langle \text{proof} \rangle$

**lemma** *swap-param-fref*:  $(\text{uncurry2 } \text{swap}, \text{uncurry2 } \text{swap}) \in$   
 $[\lambda((l, i), j). i < \text{length } l \wedge j < \text{length } l]_f (\langle A \rangle \text{list-rel} \times_r \text{nat-rel}) \times_r \text{nat-rel} \rightarrow \langle A \rangle \text{list-rel}$   
 $\langle \text{proof} \rangle$

**lemma** *param-list-null*[param]:  $(\text{List.null}, \text{List.null}) \in \langle A \rangle \text{list-rel} \rightarrow \text{bool-rel}$   
 $\langle \text{proof} \rangle$

### 3.5.4 Operations

**sepref-decl-op** *list-empty*:  $\llbracket \rrbracket :: \langle A \rangle \text{list-rel} \langle \text{proof} \rangle$

**context notes** [simp] = *eq-Nil-null* **begin**

**sepref-decl-op** *list-is-empty*:  $\lambda l. l = \llbracket \rrbracket :: \langle A \rangle \text{list-rel} \rightarrow_f \text{bool-rel} \langle \text{proof} \rangle$

**end**

**sepref-decl-op** *list-replicate*:  $\text{replicate} :: \text{nat-rel} \rightarrow A \rightarrow \langle A \rangle \text{list-rel} \langle \text{proof} \rangle$

**definition** *op-list-copy* ::  $'a \ \text{list} \Rightarrow 'a \ \text{list}$  **where** [simp]:  $\text{op-list-copy } l \equiv l$

**sepref-decl-op** (*no-def*) *list-copy*:  $\text{op-list-copy} :: \langle A \rangle \text{list-rel} \rightarrow \langle A \rangle \text{list-rel} \langle \text{proof} \rangle$

**sepref-decl-op** *list-prepend*:  $(\#) :: A \rightarrow \langle A \rangle \text{list-rel} \rightarrow \langle A \rangle \text{list-rel} \langle \text{proof} \rangle$

**sepref-decl-op** *list-append*:  $\lambda xs \ x. xs @ [x] :: \langle A \rangle \text{list-rel} \rightarrow A \rightarrow \langle A \rangle \text{list-rel} \langle \text{proof} \rangle$

**sepref-decl-op** *list-concat*:  $(@) :: \langle A \rangle \text{list-rel} \rightarrow \langle A \rangle \text{list-rel} \rightarrow \langle A \rangle \text{list-rel} \langle \text{proof} \rangle$

**sepref-decl-op** *list-length*:  $\text{length} :: \langle A \rangle \text{list-rel} \rightarrow \text{nat-rel} \langle \text{proof} \rangle$

**sepref-decl-op** *list-get*:  $nth :: [\lambda(l,i). i < \text{length } l]_f \langle A \rangle \text{list-rel} \times_r \text{nat-rel} \rightarrow A \langle \text{proof} \rangle$   
**sepref-decl-op** *list-set*:  $list\text{-update} :: [\lambda((l,i),-). i < \text{length } l]_f (\langle A \rangle \text{list-rel} \times_r \text{nat-rel}) \times_r A \rightarrow \langle A \rangle \text{list-rel} \langle \text{proof} \rangle$   
**context notes** [*simp*] = *eq-Nil-null* **begin**  
**sepref-decl-op** *list-hd*:  $hd :: [\lambda l. l \neq []]_f \langle A \rangle \text{list-rel} \rightarrow A \langle \text{proof} \rangle$   
**sepref-decl-op** *list-tl*:  $tl :: [\lambda l. l \neq []]_f \langle A \rangle \text{list-rel} \rightarrow \langle A \rangle \text{list-rel} \langle \text{proof} \rangle$   
**sepref-decl-op** *list-last*:  $last :: [\lambda l. l \neq []]_f \langle A \rangle \text{list-rel} \rightarrow A \langle \text{proof} \rangle$   
**sepref-decl-op** *list-butlast*:  $butlast :: [\lambda l. l \neq []]_f \langle A \rangle \text{list-rel} \rightarrow \langle A \rangle \text{list-rel} \langle \text{proof} \rangle$   
**end**  
**sepref-decl-op** *list-contains*:  $\lambda x l. x \in \text{set } l :: A \rightarrow \langle A \rangle \text{list-rel} \rightarrow \text{bool-rel}$   
**where** *single-valued*  $A$  *single-valued*  $(A^{-1}) \langle \text{proof} \rangle$   
**sepref-decl-op** *list-swap*:  $swap :: [\lambda((l,i),j). i < \text{length } l \wedge j < \text{length } l]_f (\langle A \rangle \text{list-rel} \times_r \text{nat-rel}) \times_r \text{nat-rel} \rightarrow \langle A \rangle \text{list-rel} \langle \text{proof} \rangle$   
**sepref-decl-op** *list-rotate1*:  $rotate1 :: \langle A \rangle \text{list-rel} \rightarrow \langle A \rangle \text{list-rel} \langle \text{proof} \rangle$   
**sepref-decl-op** *list-rev*:  $rev :: \langle A \rangle \text{list-rel} \rightarrow \langle A \rangle \text{list-rel} \langle \text{proof} \rangle$   
**sepref-decl-op** *list-index*:  $index :: \langle A \rangle \text{list-rel} \rightarrow A \rightarrow \text{nat-rel}$   
**where** *single-valued*  $A$  *single-valued*  $(A^{-1}) \langle \text{proof} \rangle$

### 3.5.5 Patterns

**lemma** [*def-pat-rules*]:

$[] \equiv \text{op-list-empty}$   
 $(=) \$l\$[] \equiv \text{op-list-is-empty}\$l$   
 $(=) \$[]\$l \equiv \text{op-list-is-empty}\$l$   
 $\text{replicate}\$n\$v \equiv \text{op-list-replicate}\$n\$v$   
 $\text{Cons}\$x\$xs \equiv \text{op-list-prepend}\$x\$xs$   
 $(@) \$xs\$(\text{Cons}\$x\$[]) \equiv \text{op-list-append}\$xs\$x$   
 $(@) \$xs\$ys \equiv \text{op-list-concat}\$xs\$ys$   
 $\text{op-list-concat}\$xs\$(\text{Cons}\$x\$[]) \equiv \text{op-list-append}\$xs\$x$   
 $\text{length}\$xs \equiv \text{op-list-length}\$xs$   
 $\text{nth}\$l\$i \equiv \text{op-list-get}\$l\$i$   
 $\text{list-update}\$l\$i\$x \equiv \text{op-list-set}\$l\$i\$x$   
 $\text{hd}\$l \equiv \text{op-list-hd}\$l$   
 $\text{hd}\$l \equiv \text{op-list-hd}\$l$   
 $\text{tl}\$l \equiv \text{op-list-tl}\$l$   
 $\text{tl}\$l \equiv \text{op-list-tl}\$l$   
 $\text{last}\$l \equiv \text{op-list-last}\$l$   
 $\text{butlast}\$l \equiv \text{op-list-butlast}\$l$   
 $(\in) \$x\$(\text{set}\$l) \equiv \text{op-list-contains}\$x\$l$   
 $\text{swap}\$l\$i\$j \equiv \text{op-list-swap}\$l\$i\$j$   
 $\text{rotate1}\$l \equiv \text{op-list-rotate1}\$l$   
 $\text{rev}\$l \equiv \text{op-list-rev}\$l$   
 $\text{index}\$l\$x \equiv \text{op-list-index}\$l\$x$   
 $\langle \text{proof} \rangle$

Standard preconditions are preserved by list-relation. These lemmas are used for simplification of preconditions after composition.

**lemma** *list-rel-pres-neq-nil*[*fcomp-prenorm-simps*]:  $(x',x) \in \langle A \rangle \text{list-rel} \implies x' \neq [] \longleftrightarrow$

```

x≠[] ⟨proof⟩
lemma list-rel-pres-length[fcomp-prenorm-simps]: (x',x)∈⟨A⟩list-rel ⇒ length x'
= length x ⟨proof⟩

locale list-custom-empty =
  fixes rel empty and op-custom-empty :: 'a list
  assumes customize-hnr-aux: (uncurry0 empty,uncurry0 (RETURN (op-list-empty::'a
list))) ∈ unit-assnk →a rel
  assumes op-custom-empty-def: op-custom-empty = op-list-empty
begin
  sepref-register op-custom-empty :: 'c list

  lemma fold-custom-empty:
    [] = op-custom-empty
    op-list-empty = op-custom-empty
    mop-list-empty = RETURN op-custom-empty
    ⟨proof⟩

  lemmas custom-hnr[sepref-fr-rules] = customize-hnr-aux[folded op-custom-empty-def]
end

lemma gen-mop-list-swap: mop-list-swap l i j = do {
  xi ← mop-list-get l i;
  xj ← mop-list-get l j;
  l ← mop-list-set l i xj;
  l ← mop-list-set l j xi;
  RETURN l
}
⟨proof⟩

end

```

### 3.6 Heap Implementation On Lists

```

theory IICF-Abs-Heap
imports
  HOL-Library.Multiset
  ../ ../ ../ Sepref
  List-Index.List-Index
  ../ ../ Intf/IICF-List
  ../ ../ Intf/IICF-Prio-Bag
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**  $'a \text{ heap} = 'a \text{ list}$

**locale**  $\text{heapstruct} =$

**fixes**  $\text{prio} :: 'a \Rightarrow 'b::\text{linorder}$

**begin**

**definition**  $\text{valid} :: 'a \text{ heap} \Rightarrow \text{nat} \Rightarrow \text{bool}$

**where**  $\text{valid } h \ i \equiv i > 0 \wedge i \leq \text{length } h$

**abbreviation**  $\alpha :: 'a \text{ heap} \Rightarrow 'a \text{ multiset}$  **where**  $\alpha \equiv \text{mset}$

**lemma**  $\text{valid-empty}[simp]: \neg \text{valid } [] \ i \ \langle \text{proof} \rangle$

**lemma**  $\text{valid0}[simp]: \neg \text{valid } h \ 0 \ \langle \text{proof} \rangle$

**lemma**  $\text{valid-glen}[simp]: i > \text{length } h \Longrightarrow \neg \text{valid } h \ i \ \langle \text{proof} \rangle$

**lemma**  $\text{valid-len}[simp]: h \neq [] \Longrightarrow \text{valid } h \ (\text{length } h) \ \langle \text{proof} \rangle$

**lemma**  $\text{validI}: 0 < i \Longrightarrow i \leq \text{length } h \Longrightarrow \text{valid } h \ i$   
 $\langle \text{proof} \rangle$

**definition**  $\text{val-of} :: 'a \text{ heap} \Rightarrow \text{nat} \Rightarrow 'a$  **where**  $\text{val-of } l \ i \equiv l!(i-1)$

**abbreviation**  $\text{prio-of} :: 'a \text{ heap} \Rightarrow \text{nat} \Rightarrow 'b$  **where**

$\text{prio-of } l \ i \equiv \text{prio } (\text{val-of } l \ i)$

#### Navigating the tree

**definition**  $\text{parent} :: \text{nat} \Rightarrow \text{nat}$  **where**  $\text{parent } i \equiv i \text{ div } 2$

**definition**  $\text{left} :: \text{nat} \Rightarrow \text{nat}$  **where**  $\text{left } i \equiv 2*i$

**definition**  $\text{right} :: \text{nat} \Rightarrow \text{nat}$  **where**  $\text{right } i \equiv 2*i + 1$

**abbreviation**  $\text{has-parent } h \ i \equiv \text{valid } h \ (\text{parent } i)$

**abbreviation**  $\text{has-left } h \ i \equiv \text{valid } h \ (\text{left } i)$

**abbreviation**  $\text{has-right } h \ i \equiv \text{valid } h \ (\text{right } i)$

**abbreviation**  $\text{vparent } h \ i == \text{val-of } h \ (\text{parent } i)$

**abbreviation**  $\text{vleft } h \ i == \text{val-of } h \ (\text{left } i)$

**abbreviation**  $\text{vright } h \ i == \text{val-of } h \ (\text{right } i)$

**abbreviation**  $\text{pparent } h \ i == \text{prio-of } h \ (\text{parent } i)$

**abbreviation**  $\text{pleft } h \ i == \text{prio-of } h \ (\text{left } i)$

**abbreviation**  $\text{pright } h \ i == \text{prio-of } h \ (\text{right } i)$

**lemma**  $\text{parent-left-id}[simp]: \text{parent } (\text{left } i) = i$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{parent-right-id}[simp]: \text{parent } (\text{right } i) = i$   
 $\langle \text{proof} \rangle$

**lemma** *child-of-parentD*:  
 $has\text{-}parent\ l\ i \implies left\ (parent\ i) = i \vee right\ (parent\ i) = i$   
 $\langle proof \rangle$

**lemma** *rc-imp-lc*:  $\llbracket valid\ h\ i; has\text{-}right\ h\ i \rrbracket \implies has\text{-}left\ h\ i$   
 $\langle proof \rangle$

**lemma** *plr-corner-cases[simp]*:

**assumes**  $0 < i$

**shows**

$i \neq parent\ i$

$i \neq left\ i$

$i \neq right\ i$

$parent\ i \neq i$

$left\ i \neq i$

$right\ i \neq i$

$\langle proof \rangle$

**lemma** *i-eq-parent-conv[simp]*:  $i = parent\ i \longleftrightarrow i = 0$   
 $\langle proof \rangle$

## 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*  $l$   
 $\equiv \forall i. valid\ l\ i \longrightarrow has\text{-}parent\ l\ i \longrightarrow pparent\ l\ i \leq prio\text{-}of\ l\ i$

**definition** *heap-rel1*  $\equiv br\ \alpha\ heap\text{-}invar$

**lemma** *heap-invar-empty[simp]*:  $heap\text{-}invar\ []$   
 $\langle proof \rangle$

**function** *heap-induction-scheme* ::  $nat \Rightarrow unit$  **where**  
*heap-induction-scheme*  $i =$  (  
  if  $i > 1$  then *heap-induction-scheme* (parent  $i$ ) else ())  
 $\langle proof \rangle$

**termination**

$\langle proof \rangle$

**lemma**

*heap-parent-le*:  $\llbracket heap\text{-}invar\ l; valid\ l\ i; has\text{-}parent\ l\ i \rrbracket$   
 $\implies pparent\ l\ i \leq prio\text{-}of\ l\ i$   
 $\langle proof \rangle$

**lemma** *heap-min-prop*:

**assumes**  $H$ : *heap-invar*  $h$   
**assumes**  $V$ : *valid*  $h$   $i$   
**shows** *prio-of*  $h$  (*Suc* 0)  $\leq$  *prio-of*  $h$   $i$   
 $\langle$ *proof* $\rangle$

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*  $h$   $p$   $i$   $\equiv$   
 $(\textit{has-left } h \ i \ \longrightarrow \ p \leq \textit{pleft } h \ i)$   
 $\wedge (\textit{has-right } h \ i \ \longrightarrow \ p \leq \textit{pright } h \ i)$

**definition** *heap-invar'*  $h$   $\equiv \forall i. \textit{valid } h \ i \ \longrightarrow \ \textit{children-ge } h \ (\textit{prio-of } h \ i) \ i$

**lemma** *heap-eq-heap'*:  
**shows** *heap-invar*  $h \ \longleftrightarrow \ \textit{heap-invar}' \ h$   
 $\langle$ *proof* $\rangle$

### 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*  $l$   $i$   $\equiv \textit{valid } l \ i$   
**definition** *val-of-op*  $:: 'a \ \textit{heap} \ \Rightarrow \ \textit{nat} \ \Rightarrow \ 'a \ \textit{nres}$   
**where** *val-of-op*  $l \ i \ \equiv \ \textit{ASSERT } (i > 0) \ \gg \ \textit{mop-list-get } l \ (i-1)$   
**lemma** *val-of-correct*[*refine-vcg*]:  
 $\textit{val-of-pre } l \ i \ \Longrightarrow \ \textit{val-of-op } l \ i \ \leq \ \textit{SPEC } (\lambda r. \ r = \textit{val-of } l \ i)$   
 $\langle$ *proof* $\rangle$

**abbreviation** (*input*) *prio-of-pre*  $\equiv \textit{val-of-pre}$   
**definition** *prio-of-op*  $l \ i \ \equiv \ \textit{do } \{v \leftarrow \textit{val-of-op } l \ i; \ \textit{RETURN } (\textit{prio } v)\}$   
**lemma** *prio-of-op-correct*[*refine-vcg*]:  
 $\textit{prio-of-pre } l \ i \ \Longrightarrow \ \textit{prio-of-op } l \ i \ \leq \ \textit{SPEC } (\lambda r. \ r = \textit{prio-of } l \ i)$   
 $\langle$ *proof* $\rangle$

#### Update

**abbreviation** *update-pre*  $h \ i \ v \ \equiv \ \textit{valid } h \ i$   
**definition** *update*  $:: 'a \ \textit{heap} \ \Rightarrow \ \textit{nat} \ \Rightarrow \ 'a \ \Rightarrow \ 'a \ \textit{heap}$   
**where** *update*  $h \ i \ v \ \equiv \ h[i - 1 := v]$   
**definition** *update-op*  $:: 'a \ \textit{heap} \ \Rightarrow \ \textit{nat} \ \Rightarrow \ 'a \ \Rightarrow \ 'a \ \textit{heap} \ \textit{nres}$   
**where** *update-op*  $h \ i \ v \ \equiv \ \textit{ASSERT } (i > 0) \ \gg \ \textit{mop-list-set } h \ (i-1) \ v$   
**lemma** *update-correct*[*refine-vcg*]:  
 $\textit{update-pre } h \ i \ v \ \Longrightarrow \ \textit{update-op } h \ i \ v \ \leq \ \textit{SPEC}(\lambda r. \ r = \textit{update } h \ i \ v)$   
 $\langle$ *proof* $\rangle$

**lemma** *update-valid*[*simp*]:  $\textit{valid } (\textit{update } h \ i \ v) \ j \ \longleftrightarrow \ \textit{valid } h \ j$



$\langle \text{proof} \rangle$

**lemma** *val-of-update[simp]*:  $\llbracket \text{update-pre } h \ i \ v; \text{ valid } h \ j \rrbracket \implies \text{val-of } (\text{update } h \ i \ v) \ j =$   
(  
  if  $i=j$  then  $v$  else  $\text{val-of } h \ j$ )  
   $\langle \text{proof} \rangle$ )

**lemma** *length-update[simp]*:  $\text{length } (\text{update } l \ i \ v) = \text{length } l$   
 $\langle \text{proof} \rangle$

## Exchange

Exchange two elements

**definition** *exch* ::  $'a \ \text{heap} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \ \text{heap}$  **where**  
   $\text{exch } l \ i \ j \equiv \text{swap } l \ (i - 1) \ (j - 1)$   
**abbreviation** *exch-pre*  $l \ i \ j \equiv \text{valid } l \ i \wedge \text{valid } l \ j$

**definition** *exch-op* ::  $'a \ \text{list} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \ \text{list nres}$   
**where** *exch-op*  $l \ i \ j \equiv \text{do } \{$   
   $\text{ASSERT } (i > 0 \wedge j > 0);$   
   $l \leftarrow \text{mop-list-swap } l \ (i - 1) \ (j - 1);$   
   $\text{RETURN } l$   
   $\}$

**lemma** *exch-op-alt*:  $\text{exch-op } l \ i \ j = \text{do } \{$   
   $vi \leftarrow \text{val-of-op } l \ i;$   
   $vj \leftarrow \text{val-of-op } l \ j;$   
   $l \leftarrow \text{update-op } l \ i \ vj;$   
   $l \leftarrow \text{update-op } l \ j \ vi;$   
   $\text{RETURN } l \}$   
 $\langle \text{proof} \rangle$

**lemma** *exch-op-correct[refine-vcg]*:  
   $\text{exch-pre } l \ i \ j \implies \text{exch-op } l \ i \ j \leq \text{SPEC } (\lambda r. r = \text{exch } l \ i \ j)$   
 $\langle \text{proof} \rangle$

**lemma** *valid-exch[simp]*:  $\text{valid } (\text{exch } l \ i \ j) \ k = \text{valid } l \ k$   
 $\langle \text{proof} \rangle$

**lemma** *val-of-exch[simp]*:  $\llbracket \text{valid } l \ i; \text{ valid } l \ j; \text{ valid } l \ k \rrbracket \implies$   
 $\text{val-of } (\text{exch } l \ i \ j) \ k =$   
  (  
    if  $k=i$  then  $\text{val-of } l \ j$   
    else if  $k=j$  then  $\text{val-of } l \ i$   
    else  $\text{val-of } l \ k$   
  )  
 $\langle \text{proof} \rangle$

**lemma** *exch-eq[simp]*:  $\text{exch } h \ i \ i = h$   
 $\langle \text{proof} \rangle$

**lemma**  $\alpha$ -*exch*[*simp*]:  $\llbracket \text{valid } l \ i; \text{ valid } l \ j \rrbracket$   
 $\implies \alpha (\text{exch } l \ i \ j) = \alpha l$   
 $\langle \text{proof} \rangle$

**lemma** *length-exch*[*simp*]:  $\text{length } (\text{exch } l \ i \ j) = \text{length } l$   
 $\langle \text{proof} \rangle$

## Butlast

Remove last element

**abbreviation** *butlast-pre*  $l \equiv l \neq []$

**definition** *butlast-op* ::  $'a \ \text{heap} \Rightarrow 'a \ \text{heap} \ \text{nres}$   
**where** *butlast-op*  $l \equiv \text{mop-list-butlast } l$

**lemma** *butlast-op-correct*[*refine-vcg*]:  
 $\text{butlast-pre } l \implies \text{butlast-op } l \leq \text{SPEC } (\lambda r. r = \text{butlast } l)$   
 $\langle \text{proof} \rangle$

**lemma** *valid-butlast-conv*[*simp*]:  $\text{valid } (\text{butlast } h) \ i \longleftrightarrow \text{valid } h \ i \wedge i < \text{length } h$   
 $\langle \text{proof} \rangle$

**lemma** *valid-butlast*:  $\text{valid } (\text{butlast } h) \ i \implies \text{valid } h \ i$   
 $\langle \text{proof} \rangle$

**lemma** *val-of-butlast*[*simp*]:  $\llbracket \text{valid } h \ i; \ i < \text{length } h \rrbracket$   
 $\implies \text{val-of } (\text{butlast } h) \ i = \text{val-of } h \ i$   
 $\langle \text{proof} \rangle$

**lemma** *val-of-butlast'*[*simp*]:  
 $\text{valid } (\text{butlast } h) \ i \implies \text{val-of } (\text{butlast } h) \ i = \text{val-of } h \ i$   
 $\langle \text{proof} \rangle$

**lemma**  $\alpha$ -*butlast*[*simp*]:  $\llbracket \text{length } h \neq 0 \rrbracket$   
 $\implies \alpha (\text{butlast } h) = \alpha h - \{\# \text{val-of } h \ (\text{length } h) \# \}$   
 $\langle \text{proof} \rangle$

**lemma** *heap-invar-butlast*[*simp*]:  $\text{heap-invar } h \implies \text{heap-invar } (\text{butlast } h)$   
 $\langle \text{proof} \rangle$

## Append

**definition** *append-op* ::  $'a \ \text{heap} \Rightarrow 'a \Rightarrow 'a \ \text{heap} \ \text{nres}$   
**where** *append-op*  $l \ v \equiv \text{mop-list-append } l \ v$

**lemma** *append-op-correct*[*refine-vcg*]:  
 $\text{append-op } l \ v \leq \text{SPEC } (\lambda r. r = l @ [v])$   
 $\langle \text{proof} \rangle$

**lemma** *valid-append*[*simp*]:  $\text{valid } (l @ [v]) \ i \longleftrightarrow \text{valid } l \ i \vee i = \text{length } l + 1$

⟨proof⟩

**lemma** *val-of-append[simp]*:  $\text{valid } (l@[v]) \ i \implies$   
 $\text{val-of } (l@[v]) \ i = (\text{if valid } l \ i \ \text{then val-of } l \ i \ \text{else } v)$   
⟨proof⟩

**lemma** *α-append[simp]*:  $\alpha \ (l@[v]) = \alpha \ l + \{\#v\#}$   
⟨proof⟩

### 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* ::  $'a \ \text{heap} \Rightarrow \text{nat} \Rightarrow \text{bool}$   
**where** *swim-invar*  $h \ i \equiv$   
 $\text{valid } h \ i$   
 $\wedge (\forall j. \text{valid } h \ j \wedge \text{has-parent } h \ j \wedge j \neq i \longrightarrow \text{pparent } h \ j \leq \text{prio-of } h \ j)$   
 $\wedge (\text{has-parent } h \ i \longrightarrow$   
 $(\forall j. \text{valid } h \ j \wedge \text{has-parent } h \ j \wedge \text{parent } j = i$   
 $\longrightarrow \text{pparent } h \ i \leq \text{prio-of } h \ j))$

Move up an element that is too small, until it fits

**definition** *swim-op* ::  $'a \ \text{heap} \Rightarrow \text{nat} \Rightarrow 'a \ \text{heap} \ \text{nres}$  **where**  
*swim-op*  $h \ i \equiv \text{do } \{$   
 $\text{RECT } (\lambda \text{swim } (h,i). \text{do } \{$   
 $\text{ASSERT } (\text{valid } h \ i \wedge \text{swim-invar } h \ i);$   
 $\text{if has-parent } h \ i \ \text{then do } \{$   
 $\text{ppi} \leftarrow \text{prio-of-op } h \ (\text{parent } i);$   
 $\text{pi} \leftarrow \text{prio-of-op } h \ i;$   
 $\text{if } (\neg \text{ppi} \leq \text{pi}) \ \text{then do } \{$   
 $\text{h} \leftarrow \text{exch-op } h \ i \ (\text{parent } i);$   
 $\text{swim } (h, \text{parent } i)$   
 $\} \ \text{else}$   
 $\text{RETURN } h$   
 $\} \ \text{else}$   
 $\text{RETURN } h$   
 $\}) \ (h,i)$   
 $\}$

**lemma** *swim-invar-valid*:  $\text{swim-invar } h \ i \implies \text{valid } h \ i$   
⟨proof⟩

**lemma** *swim-invar-exit1*:  $\neg \text{has-parent } h \ i \implies \text{swim-invar } h \ i \implies \text{heap-invar } h$

⟨proof⟩

**lemma** *swim-invar-exit2*:  $pparent\ h\ i \leq prio-of\ h\ i \implies swim-invar\ h\ i \implies$   
*heap-invar\ h*  
⟨proof⟩

**lemma** *swim-invar-pres*:  
**assumes** *HPI*: *has-parent\ h\ i*  
**assumes** *VIOLATED*:  $pparent\ h\ i > prio-of\ h\ i$   
**and** *INV*: *swim-invar\ h\ i*  
**defines**  $h' \equiv exch\ h\ i\ (parent\ i)$   
**shows** *swim-invar\ h'\ (parent\ i)*  
⟨proof⟩

**lemma** *swim-invar-decr*:  
**assumes** *INV*: *heap-invar\ h*  
**assumes** *V*: *valid\ h\ i*  
**assumes** *DECR*:  $prio\ v \leq prio-of\ h\ i$   
**shows** *swim-invar\ (update\ h\ i\ v)\ i*  
⟨proof⟩

**lemma** *swim-op-correct*[*refine-vcg*]:  
[[*swim-invar\ h\ i*]]  $\implies$   
*swim-op\ h\ i*  $\leq SPEC\ (\lambda h'. \alpha\ h' = \alpha\ h \wedge heap-invar\ h' \wedge length\ h' = length$   
*h*)  
⟨proof⟩

## Sink

Move down an element that is too big, until it fits in

**definition** *sink-op* :: '*a heap*  $\Rightarrow$  *nat*  $\Rightarrow$  '*a heap nres* **where**  
*sink-op\ h\ i*  $\equiv do\ \{\$   
  *RECT*  $(\lambda sink\ (h,i). do\ \{\$   
    *ASSERT*  $(valid\ h\ i)$ ;  
    *if\ has-right\ h\ i\ then\ do\ \{\  
      *ASSERT*  $(has-left\ h\ i)$ ;  
       $lp \leftarrow prio-of-op\ h\ (left\ i)$ ;  
       $rp \leftarrow prio-of-op\ h\ (right\ i)$ ;  
       $p \leftarrow prio-of-op\ h\ i$ ;  
      *if*  $(lp < p \wedge rp \geq lp)$  *then\ do\ \{\  
         $h \leftarrow exch-op\ h\ i\ (left\ i)$ ;  
        *sink*  $(h, left\ i)$   
      } *else\ if*  $(rp < lp \wedge rp < p)$  *then\ do\ \{\  
         $h \leftarrow exch-op\ h\ i\ (right\ i)$ ;  
        *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);
      sink (h, left i)
    } else
      RETURN h

  } else
    RETURN h
}) (h, i)
}

```

This invariant expresses that the heap has a single defect, which can be repaired by sinking

**definition** *sink-invar*  $l\ i \equiv$   
 $valid\ l\ i$   
 $\wedge (\forall j. valid\ l\ j \wedge j \neq i \longrightarrow children\text{-}ge\ l\ (prio\text{-}of\ l\ j)\ j)$   
 $\wedge (has\text{-}parent\ l\ i \longrightarrow children\text{-}ge\ l\ (pparent\ l\ i)\ i)$

**lemma** *sink-invar-valid*:  $sink\text{-}invar\ l\ i \implies valid\ l\ i$   
 $\langle proof \rangle$

**lemma** *sink-invar-exit*:  $\llbracket sink\text{-}invar\ l\ i; children\text{-}ge\ l\ (prio\text{-}of\ l\ i)\ i \rrbracket$   
 $\implies heap\text{-}invar'\ l$   
 $\langle proof \rangle$

**lemma** *sink-aux1*:  $\neg (2 * i \leq length\ h) \implies \neg has\text{-}left\ h\ i \wedge \neg has\text{-}right\ h\ i$   
 $\langle proof \rangle$

**lemma** *sink-invar-pres1*:  
**assumes**  $sink\text{-}invar\ h\ i$   
**assumes**  $has\text{-}left\ h\ i\ has\text{-}right\ h\ i$   
**assumes**  $prio\text{-}of\ h\ i \geq pleft\ h\ i$   
**assumes**  $pleft\ h\ i \geq pright\ h\ i$   
**shows**  $sink\text{-}invar\ (exch\ h\ i\ (right\ i))\ (right\ i)$   
 $\langle proof \rangle$

**lemma** *sink-invar-pres2*:  
**assumes**  $sink\text{-}invar\ h\ i$   
**assumes**  $has\text{-}left\ h\ i\ has\text{-}right\ h\ i$   
**assumes**  $prio\text{-}of\ h\ i \geq pleft\ h\ i$   
**assumes**  $pleft\ h\ i \leq pright\ h\ i$   
**shows**  $sink\text{-}invar\ (exch\ h\ i\ (left\ i))\ (left\ i)$   
 $\langle proof \rangle$

**lemma** *sink-invar-pres3*:  
**assumes**  $sink\text{-}invar\ h\ i$   
**assumes**  $has\text{-}left\ h\ i\ has\text{-}right\ h\ i$

**assumes**  $\text{prio-of } h \ i \geq \text{pright } h \ i$   
**assumes**  $\text{pleft } h \ i \leq \text{pright } h \ i$   
**shows**  $\text{sink-invar } (\text{exch } h \ i \ (\text{left } i)) \ (\text{left } i)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{sink-invar-pres4}$ :  
**assumes**  $\text{sink-invar } h \ i$   
**assumes**  $\text{has-left } h \ i \ \text{has-right } h \ i$   
**assumes**  $\text{prio-of } h \ i \geq \text{pright } h \ i$   
**assumes**  $\text{pleft } h \ i \geq \text{pright } h \ i$   
**shows**  $\text{sink-invar } (\text{exch } h \ i \ (\text{right } i)) \ (\text{right } i)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{sink-invar-pres5}$ :  
**assumes**  $\text{sink-invar } h \ i$   
**assumes**  $\text{has-left } h \ i \ \neg \text{has-right } h \ i$   
**assumes**  $\text{prio-of } h \ i \geq \text{pleft } h \ i$   
**shows**  $\text{sink-invar } (\text{exch } h \ i \ (\text{left } i)) \ (\text{left } i)$   
 $\langle \text{proof} \rangle$

**lemmas**  $\text{sink-invar-pres} =$   
 $\text{sink-invar-pres1}$   
 $\text{sink-invar-pres2}$   
 $\text{sink-invar-pres3}$   
 $\text{sink-invar-pres4}$   
 $\text{sink-invar-pres5}$

**lemma**  $\text{sink-invar-incr}$ :  
**assumes**  $\text{INV: heap-invar } h$   
**assumes**  $V: \text{valid } h \ i$   
**assumes**  $\text{INCR: } \text{prio } v \geq \text{prio-of } h \ i$   
**shows**  $\text{sink-invar } (\text{update } h \ i \ v) \ i$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{sink-op-correct}[\text{refine-vcg}]$ :  
 $\llbracket \text{sink-invar } h \ i \rrbracket \implies$   
 $\text{sink-op } h \ i \leq \text{SPEC } (\lambda h'. \alpha \ h' = \alpha \ h \wedge \text{heap-invar } h' \wedge \text{length } h' = \text{length } h)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{sink-op-swim-rule}$ :  
 $\text{swim-invar } h \ i \implies \text{sink-op } h \ i \leq \text{SPEC } (\lambda h'. \ h' = h)$   
 $\langle \text{proof} \rangle$

**definition**  $\text{sink-op-opt}$   
— Sink operation as presented in Sedgewick et al. Algs4 reference implementation

**where**

```
sink-op-opt h k  $\equiv$  RECT ( $\lambda D (h,k)$ ). do {  
  ASSERT ( $k > 0 \wedge k \leq \text{length } h$ );  
  let len = length h;  
  if ( $2 * k \leq \text{len}$ ) then do {  
    let j =  $2 * k$ ;  
    pj  $\leftarrow$  prio-of-op h j;  
  
    j  $\leftarrow$  (  
      if  $j < \text{len}$  then do {  
        psj  $\leftarrow$  prio-of-op h (Suc j);  
        if  $pj > psj$  then RETURN (j+1) else RETURN j  
      } else RETURN j);  
  
    pj  $\leftarrow$  prio-of-op h j;  
    pk  $\leftarrow$  prio-of-op h k;  
    if ( $pk > pj$ ) then do {  
      h  $\leftarrow$  exch-op h k j;  
      D (h,j)  
    } else  
      RETURN h  
    } else RETURN h  
  } (h,k)
```

**lemma** *sink-op-opt-eq*: *sink-op-opt* *h k* = *sink-op* *h k*  
*<proof>*

## 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.

```
definition repair-op h i  $\equiv$  do {  
  h  $\leftarrow$  sink-op h i;  
  h  $\leftarrow$  swim-op h i;  
  RETURN h  
}
```

**lemma** *update-sink-swim-cases*:

```
assumes heap-invar h  
assumes valid h i  
obtains swim-invar (update h i v) i | sink-invar (update h i v) i  
<proof>
```

**lemma** *heap-invar-imp-swim-invar*:  $\llbracket \text{heap-invar } h; \text{valid } h \ i \rrbracket \implies \text{swim-invar } h$   
*i*  
*<proof>*

**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'  $\wedge$   $\alpha$  h' =  $\alpha$  (update h i v)  $\wedge$  length h' = length h*)  
 $\langle$ *proof* $\rangle$

### 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 = {#}  $\wedge$  *heap-invar r*)  
 $\langle$ *proof* $\rangle$

#### 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*  $\Longrightarrow$  *is-empty-op h*  $\leq$  *SPEC* ( $\lambda r$ .  $r \longleftrightarrow \alpha$  h = {#})  
 $\langle$ *proof* $\rangle$

#### 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*  
}

**lemma** *swim-invar-insert*: *heap-invar l*  $\Longrightarrow$  *swim-invar (l@[x]) (Suc (length l))*  
 $\langle$ *proof* $\rangle$

**lemma**  
(*insert-op,RETURN oo op-mset-insert*)  $\in$  *Id*  $\rightarrow$  *heap-rel1*  $\rightarrow$   $\langle$ *heap-rel1* $\rangle$ *nres-rel*  
 $\langle$ *proof* $\rangle$

**lemma** *insert-op-correct*:  
*heap-invar h*  $\Longrightarrow$  *insert-op v h*  $\leq$  *SPEC* ( $\lambda h'$ . *heap-invar h'  $\wedge$   $\alpha$  h' =  $\alpha$  h +*  
{#v#})



$\langle \text{proof} \rangle$   
**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  
 $\langle \text{proof} \rangle$

**lemma** right-one-conv[simp]: right j = Suc 0  $\longleftrightarrow$  j=0  
 $\langle \text{proof} \rangle$

**lemma** parent-one-conv[simp]: parent (Suc 0) = 0  
 $\langle \text{proof} \rangle$

**lemma** sink-invar-init:  
**assumes** I: heap-invar h  
**assumes** NE: length h > 1  
**shows** sink-invar (butlast (exch h (Suc 0) (length h))) (Suc 0)  
 $\langle \text{proof} \rangle$

**lemma** in-set-conv-val: v  $\in$  set h  $\longleftrightarrow$  ( $\exists$  i. valid h i  $\wedge$  v = val-of h i)  
 $\langle \text{proof} \rangle$

**lemma** pop-min-op-correct:  
**assumes** heap-invar h  $\alpha$  h  $\neq$  {#}  
**shows** pop-min-op h  $\leq$  SPEC ( $\lambda(v,h').$  heap-invar h'  $\wedge$   
v  $\in$  {#}  $\alpha$  h  $\wedge$  h' =  $\alpha$  h - {#v#}  $\wedge$  ( $\forall v' \in$  set-mset ( $\alpha$  h). prio v  $\leq$  prio v'))  
 $\langle \text{proof} \rangle$

**lemmas** [refine-vcg] = pop-min-op-correct

## Peek minimum element

**definition** *peek-min-op* :: 'a heap  $\Rightarrow$  'a nres **where**  
  *peek-min-op* h  $\equiv$  do {  
    ASSERT (heap-invar h);  
    ASSERT (valid h 1);  
    val-of-op h 1  
  }

**lemma** *peek-min-op-correct*:  
  **assumes** heap-invar h  $\alpha$  h  $\neq$  {#}  
  **shows** *peek-min-op* h  $\leq$  SPEC ( $\lambda v.$   
     $v \in \# \alpha h \wedge (\forall v' \in \text{set-mset } (\alpha h). \text{prio } v \leq \text{prio } v')$ )  
  <proof>

**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*)  $\in$  (*heap-rel1*) nres-rel  
  <proof>

**lemma** *is-empty-op-refine*: (*is-empty-op*, RETURN *o op-mset-is-empty*)  $\in$  *heap-rel1*  
 $\rightarrow$  (*bool-rel*) nres-rel  
  <proof>

**lemma** *insert-op-refine*: (*insert-op*, RETURN *oo op-mset-insert*)  $\in$  Id  $\rightarrow$  *heap-rel1*  
 $\rightarrow$  (*heap-rel1*) nres-rel  
  <proof>

**lemma** *pop-min-op-refine*:  
  (*pop-min-op*, PR-CONST (*mop-prio-pop-min prio*))  $\in$  *heap-rel1*  $\rightarrow$  (Id  $\times_r$   
*heap-rel1*) nres-rel  
  <proof>

**lemma** *peek-min-op-refine*:  
  (*peek-min-op*, PR-CONST (*mop-prio-peek-min prio*))  $\in$  *heap-rel1*  $\rightarrow$  (Id) nres-rel  
  <proof>

**end**

**end**

**theory** *IICF-HOL-List*  
**imports** ../Intf/IICF-List  
**begin**

**context**  
**begin**

**private lemma** *id-take-nth-drop-rl*:  
**assumes**  $i < \text{length } l$   
**assumes**  $\bigwedge l1 \ x \ l2. \llbracket l = l1 @ x \# l2; i = \text{length } l1 \rrbracket \implies P (l1 @ x \# l2)$   
**shows**  $P \ l$   
 $\langle \text{proof} \rangle$  **lemma** *list-set-entails-aux*:  
**shows**  $\text{list-assn } A \ l \ li * A \ x \ xi \implies_A \text{list-assn } A \ (l[i := x]) \ (li[i := xi]) * \text{true}$   
 $\langle \text{proof} \rangle$  **lemma** *list-set-hd-tl-aux*:  
 $a \neq [] \implies \text{list-assn } R \ a \ c \implies_A \ R \ (\text{hd } a) \ (\text{hd } c) * \text{true}$   
 $a \neq [] \implies \text{list-assn } R \ a \ c \implies_A \text{list-assn } R \ (\text{tl } a) \ (\text{tl } c) * \text{true}$   
 $\langle \text{proof} \rangle$  **lemma** *list-set-last-butlast-aux*:  
 $a \neq [] \implies \text{list-assn } R \ a \ c \implies_A \ R \ (\text{last } a) \ (\text{last } c) * \text{true}$   
 $a \neq [] \implies \text{list-assn } R \ a \ c \implies_A \text{list-assn } R \ (\text{butlast } a) \ (\text{butlast } c) * \text{true}$   
 $\langle \text{proof} \rangle$  **lemma** *swap-decomp-simp*[simp]:  
 $\text{swap } (l1 @ x \# c21' @ xa \# l2a) \ (\text{length } l1) \ (\text{Suc } (\text{length } l1 + \text{length } c21')) =$   
 $l1 @ xa \# c21' @ x \# l2a$   
 $\text{swap } (l1 @ x \# c21' @ xa \# l2a) \ (\text{Suc } (\text{length } l1 + \text{length } c21')) \ (\text{length } l1) =$   
 $l1 @ xa \# c21' @ x \# l2a$   
 $\langle \text{proof} \rangle$  **lemma** *list-swap-aux*:  $\llbracket i < \text{length } l; j < \text{length } l \rrbracket \implies \text{list-assn } A \ l \ li$   
 $\implies_A \text{list-assn } A \ (\text{swap } l \ i \ j) \ (\text{swap } li \ i \ j) * \text{true}$   
 $\langle \text{proof} \rangle$  **lemma** *list-rotate1-aux*:  $\text{list-assn } A \ a \ c \implies_A \text{list-assn } A \ (\text{rotate1 } a)$   
 $(\text{rotate1 } c) * \text{true}$   
 $\langle \text{proof} \rangle$  **lemma** *list-rev-aux*:  $\text{list-assn } A \ a \ c \implies_A \text{list-assn } A \ (\text{rev } a) \ (\text{rev } c) * \text{true}$   
 $\langle \text{proof} \rangle$

**lemma** *mod-starE*:  
**assumes**  $h \models A * B$   
**obtains**  $h1 \ h2$  **where**  $h1 \models A \ h2 \models B$   
 $\langle \text{proof} \rangle$  **lemma** *CONSTRAINT-is-pureE*:  
**assumes** *CONSTRAINT is-pure*  $A$   
**obtains**  $R$  **where**  $A = \text{pure } R$   
 $\langle \text{proof} \rangle$  **method** *solve-dbg* =  
 $( (\text{elim } \text{CONSTRAINT-is-pureE}; (\text{simp only: list-assn-pure-conv the-pure-pure})?) ?;$   
 $\text{sep-auto}$   
 $\text{simp: pure-def hn-ctxt-def invalid-assn-def list-assn-aux-eqlen-simp}$   
 $\text{intro!}: \text{hn-refineI}[\text{THEN } \text{hn-refine-preI}] \ \text{hhrefI}$   
 $\text{elim!}: \text{mod-starE}$   
 $\text{intro: list-set-entails-aux list-set-hd-tl-aux list-set-last-butlast-aux}$   
 $\text{list-swap-aux list-rotate1-aux list-rev-aux}$   
 $;$   
 $((\text{rule entails-preI}; \text{sep-auto simp: list-assn-aux-eqlen-simp} \mid (\text{parametricity};$   
 $\text{simp; fail}))?)$   
 $)$

**private method** *solve* = *solve-dbg; fail*

**lemma** *HOL-list-empty-hnr-aux*:  $(\text{uncurry0 } (\text{return op-list-empty}), \text{uncurry0 } (\text{RETURN}$

$op\text{-list-empty}) \in unit\text{-assn}^k \rightarrow_a (list\text{-assn } A) \langle proof \rangle$   
**lemma** *HOL-list-is-empty-hnr*[sepref-fr-rules]:  $(return \circ op\text{-list-is-empty}, RETURN \circ op\text{-list-is-empty}) \in (list\text{-assn } A)^k \rightarrow_a bool\text{-assn} \langle proof \rangle$   
**lemma** *HOL-list-prepend-hnr*[sepref-fr-rules]:  $(uncurry (return \circ op\text{-list-prepend}), uncurry (RETURN \circ op\text{-list-prepend})) \in A^d *_a (list\text{-assn } A)^d \rightarrow_a list\text{-assn } A \langle proof \rangle$   
**lemma** *HOL-list-append-hnr*[sepref-fr-rules]:  $(uncurry (return \circ op\text{-list-append}), uncurry (RETURN \circ op\text{-list-append})) \in (list\text{-assn } A)^d *_a A^d \rightarrow_a list\text{-assn } A \langle proof \rangle$   
**lemma** *HOL-list-concat-hnr*[sepref-fr-rules]:  $(uncurry (return \circ op\text{-list-concat}), uncurry (RETURN \circ op\text{-list-concat})) \in (list\text{-assn } A)^d *_a (list\text{-assn } A)^d \rightarrow_a list\text{-assn } A \langle proof \rangle$   
**lemma** *HOL-list-length-hnr*[sepref-fr-rules]:  $(return \circ op\text{-list-length}, RETURN \circ op\text{-list-length}) \in (list\text{-assn } A)^k \rightarrow_a nat\text{-assn} \langle proof \rangle$   
**lemma** *HOL-list-set-hnr*[sepref-fr-rules]:  $(uncurry2 (return \circ op\text{-list-set}), uncurry2 (RETURN \circ op\text{-list-set})) \in (list\text{-assn } A)^d *_a nat\text{-assn}^k *_a A^d \rightarrow_a list\text{-assn } A \langle proof \rangle$   
**lemma** *HOL-list-hd-hnr*[sepref-fr-rules]:  $(return \circ op\text{-list-hd}, RETURN \circ op\text{-list-hd}) \in [\lambda y. y \neq []]_a (list\text{-assn } R)^d \rightarrow R \langle proof \rangle$   
**lemma** *HOL-list-tl-hnr*[sepref-fr-rules]:  $(return \circ op\text{-list-tl}, RETURN \circ op\text{-list-tl}) \in [\lambda y. y \neq []]_a (list\text{-assn } A)^d \rightarrow list\text{-assn } A \langle proof \rangle$   
**lemma** *HOL-list-last-hnr*[sepref-fr-rules]:  $(return \circ op\text{-list-last}, RETURN \circ op\text{-list-last}) \in [\lambda y. y \neq []]_a (list\text{-assn } R)^d \rightarrow R \langle proof \rangle$   
**lemma** *HOL-list-butlast-hnr*[sepref-fr-rules]:  $(return \circ op\text{-list-butlast}, RETURN \circ op\text{-list-butlast}) \in [\lambda y. y \neq []]_a (list\text{-assn } A)^d \rightarrow list\text{-assn } A \langle proof \rangle$   
**lemma** *HOL-list-swap-hnr*[sepref-fr-rules]:  $(uncurry2 (return \circ op\text{-list-swap}), uncurry2 (RETURN \circ op\text{-list-swap})) \in [\lambda((a, b), ba). b < length a \wedge ba < length a]_a (list\text{-assn } A)^d *_a nat\text{-assn}^k *_a nat\text{-assn}^k \rightarrow list\text{-assn } A \langle proof \rangle$   
**lemma** *HOL-list-rotate1-hnr*[sepref-fr-rules]:  $(return \circ op\text{-list-rotate1}, RETURN \circ op\text{-list-rotate1}) \in (list\text{-assn } A)^d \rightarrow_a list\text{-assn } A \langle proof \rangle$   
**lemma** *HOL-list-rev-hnr*[sepref-fr-rules]:  $(return \circ op\text{-list-rev}, RETURN \circ op\text{-list-rev}) \in (list\text{-assn } A)^d \rightarrow_a list\text{-assn } A \langle proof \rangle$   
  
**lemma** *HOL-list-replicate-hnr*[sepref-fr-rules]:  $CONSTRAINT\ is\text{-pure } A \implies (uncurry (return \circ op\text{-list-replicate}), uncurry (RETURN \circ op\text{-list-replicate})) \in nat\text{-assn}^k *_a A^k \rightarrow_a list\text{-assn } A \langle proof \rangle$   
**lemma** *HOL-list-get-hnr*[sepref-fr-rules]:  $CONSTRAINT\ is\text{-pure } A \implies (uncurry (return \circ op\text{-list-get}), uncurry (RETURN \circ op\text{-list-get})) \in [\lambda(a, b). b < length a]_a (list\text{-assn } A)^k *_a nat\text{-assn}^k \rightarrow A \langle proof \rangle$  **lemma** *bool-by-paramE*:  $\llbracket a; (a, b) \in Id \rrbracket \implies b \langle proof \rangle$  **lemma** *bool-by-paramE'*:  $\llbracket a; (b, a) \in Id \rrbracket \implies b \langle proof \rangle$   
  
**lemma** *HOL-list-contains-hnr*[sepref-fr-rules]:  $\llbracket CONSTRAINT\ is\text{-pure } A; single\text{-valued } (the\text{-pure } A); single\text{-valued } ((the\text{-pure } A)^{-1}) \rrbracket \implies (uncurry (return \circ op\text{-list-contains}), uncurry (RETURN \circ op\text{-list-contains})) \in A^k *_a (list\text{-assn } A)^k \rightarrow_a bool\text{-assn} \langle proof \rangle$

**lemmas**  $HOL\text{-list-empty-hnr-mop} = HOL\text{-list-empty-hnr-aux}[FCOMP\ mk\text{-mop-rl0-np}[OF\ mop\text{-list-empty-alt}]]$   
**lemmas**  $HOL\text{-list-is-empty-hnr-mop}[sepref\text{-fr-rules}] = HOL\text{-list-is-empty-hnr}[FCOMP\ mk\text{-mop-rl1-np}[OF\ mop\text{-list-is-empty-alt}]]$   
**lemmas**  $HOL\text{-list-prepend-hnr-mop}[sepref\text{-fr-rules}] = HOL\text{-list-prepend-hnr}[FCOMP\ mk\text{-mop-rl2-np}[OF\ mop\text{-list-prepend-alt}]]$   
**lemmas**  $HOL\text{-list-append-hnr-mop}[sepref\text{-fr-rules}] = HOL\text{-list-append-hnr}[FCOMP\ mk\text{-mop-rl2-np}[OF\ mop\text{-list-append-alt}]]$   
**lemmas**  $HOL\text{-list-concat-hnr-mop}[sepref\text{-fr-rules}] = HOL\text{-list-concat-hnr}[FCOMP\ mk\text{-mop-rl2-np}[OF\ mop\text{-list-concat-alt}]]$   
**lemmas**  $HOL\text{-list-length-hnr-mop}[sepref\text{-fr-rules}] = HOL\text{-list-length-hnr}[FCOMP\ mk\text{-mop-rl1-np}[OF\ mop\text{-list-length-alt}]]$   
**lemmas**  $HOL\text{-list-set-hnr-mop}[sepref\text{-fr-rules}] = HOL\text{-list-set-hnr}[FCOMP\ mk\text{-mop-rl3}[OF\ mop\text{-list-set-alt}]]$   
**lemmas**  $HOL\text{-list-hd-hnr-mop}[sepref\text{-fr-rules}] = HOL\text{-list-hd-hnr}[FCOMP\ mk\text{-mop-rl1}[OF\ mop\text{-list-hd-alt}]]$   
**lemmas**  $HOL\text{-list-tl-hnr-mop}[sepref\text{-fr-rules}] = HOL\text{-list-tl-hnr}[FCOMP\ mk\text{-mop-rl1}[OF\ mop\text{-list-tl-alt}]]$   
**lemmas**  $HOL\text{-list-last-hnr-mop}[sepref\text{-fr-rules}] = HOL\text{-list-last-hnr}[FCOMP\ mk\text{-mop-rl1}[OF\ mop\text{-list-last-alt}]]$   
**lemmas**  $HOL\text{-list-butlast-hnr-mop}[sepref\text{-fr-rules}] = HOL\text{-list-butlast-hnr}[FCOMP\ mk\text{-mop-rl1}[OF\ mop\text{-list-butlast-alt}]]$   
**lemmas**  $HOL\text{-list-swap-hnr-mop}[sepref\text{-fr-rules}] = HOL\text{-list-swap-hnr}[FCOMP\ mk\text{-mop-rl3}[OF\ mop\text{-list-swap-alt}]]$   
**lemmas**  $HOL\text{-list-rotate1-hnr-mop}[sepref\text{-fr-rules}] = HOL\text{-list-rotate1-hnr}[FCOMP\ mk\text{-mop-rl1-np}[OF\ mop\text{-list-rotate1-alt}]]$   
**lemmas**  $HOL\text{-list-rev-hnr-mop}[sepref\text{-fr-rules}] = HOL\text{-list-rev-hnr}[FCOMP\ mk\text{-mop-rl1-np}[OF\ mop\text{-list-rev-alt}]]$   
**lemmas**  $HOL\text{-list-replicate-hnr-mop}[sepref\text{-fr-rules}] = HOL\text{-list-replicate-hnr}[FCOMP\ mk\text{-mop-rl2-np}[OF\ mop\text{-list-replicate-alt}]]$   
**lemmas**  $HOL\text{-list-get-hnr-mop}[sepref\text{-fr-rules}] = HOL\text{-list-get-hnr}[FCOMP\ mk\text{-mop-rl2}[OF\ mop\text{-list-get-alt}]]$   
**lemmas**  $HOL\text{-list-contains-hnr-mop}[sepref\text{-fr-rules}] = HOL\text{-list-contains-hnr}[FCOMP\ mk\text{-mop-rl2-np}[OF\ mop\text{-list-contains-alt}]]$

**lemmas**  $HOL\text{-list-empty-hnr} = HOL\text{-list-empty-hnr-aux}\ HOL\text{-list-empty-hnr-mop}$

**end**

**definition**  $[simp]$ :  $op\text{-HOL-list-empty} \equiv op\text{-list-empty}$

**interpretation**  $HOL\text{-list}$ :  $list\text{-custom-empty}\ list\text{-assn}\ A\ return\ []\ op\text{-HOL-list-empty}$   
 $\langle proof \rangle$

**schematic-goal**

**notes**  $[sepref\text{-fr-rules}] = HOL\text{-list-empty-hnr}$

**shows**

$hn\text{-refine}\ (emp)\ (?c::?'c\ Heap)\ ?\Gamma'\ ?R\ (do\ \{\$   
 $x \leftarrow RETURN\ [1,2,3::nat];$

```

    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)
  })
  ⟨proof⟩

```

**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*  $l \equiv \lambda(a,n). \exists_A l'. a \mapsto_a l' * \uparrow(n \leq \text{length } l' \wedge l = \text{take } n \text{ } l' \wedge \text{length } l' > 0)$

**lemma** *is-array-list-prec[safe-constraint-rules]*: *precise is-array-list*  
 ⟨proof⟩

**definition** *initial-capacity*  $\equiv 16::\text{nat}$

**definition** *minimum-capacity*  $\equiv 16::\text{nat}$

**definition** *arl-empty*  $\equiv \text{do } \{$   
    $a \leftarrow \text{Array.new initial-capacity default};$   
    $\text{return } (a,0)$   
 $\}$

**definition** *arl-empty-sz init-cap*  $\equiv \text{do } \{$   
    $a \leftarrow \text{Array.new (max init-cap minimum-capacity) default};$   
    $\text{return } (a,0)$   
 $\}$

**definition** *arl-append*  $\equiv \lambda(a,n) x. \text{do } \{$   
    $\text{len} \leftarrow \text{Array.len } a;$

$\text{if } n < \text{len} \text{ then do } \{$   
      $a \leftarrow \text{Array.upd } n \ x \ a;$   
      $\text{return } (a, n+1)$   
    $\}$  *else do*  $\{$   
      $\text{let newcap} = 2 * \text{len};$   
      $a \leftarrow \text{array-grow } a \ \text{newcap} \ \text{default};$   
      $a \leftarrow \text{Array.upd } n \ x \ a;$   
      $\text{return } (a, n+1)$   
    $\}$   
 $\}$

**definition** *arl-copy*  $\equiv \lambda(a,n). \text{ do } \{$   
 $\quad a \leftarrow \text{array-copy } a;$   
 $\quad \text{return } (a,n)$   
 $\}$

**definition** *arl-length*  $:: 'a::\text{heap array-list} \Rightarrow \text{nat Heap where}$   
 $\text{arl-length} \equiv \lambda(a,n). \text{ return } (n)$

**definition** *arl-is-empty*  $:: 'a::\text{heap array-list} \Rightarrow \text{bool Heap where}$   
 $\text{arl-is-empty} \equiv \lambda(a,n). \text{ return } (n=0)$

**definition** *arl-last*  $:: 'a::\text{heap array-list} \Rightarrow 'a \text{ Heap where}$   
 $\text{arl-last} \equiv \lambda(a,n). \text{ do } \{$   
 $\quad \text{Array.nth } a \ (n - 1)$   
 $\}$

**definition** *arl-butlast*  $:: 'a::\text{heap array-list} \Rightarrow 'a \text{ array-list Heap where}$   
 $\text{arl-butlast} \equiv \lambda(a,n). \text{ do } \{$   
 $\quad \text{let } n = n - 1;$   
 $\quad \text{len} \leftarrow \text{Array.len } a;$   
 $\quad \text{if } (n*4 < \text{len} \wedge n*2 \geq \text{minimum-capacity}) \text{ then do } \{$   
 $\quad \quad a \leftarrow \text{array-shrink } a \ (n*2);$   
 $\quad \quad \text{return } (a,n)$   
 $\quad \}$  else  
 $\quad \text{return } (a,n)$   
 $\}$

**definition** *arl-get*  $:: 'a::\text{heap array-list} \Rightarrow \text{nat} \Rightarrow 'a \text{ Heap where}$   
 $\text{arl-get} \equiv \lambda(a,n) \ i. \text{ Array.nth } a \ i$

**definition** *arl-set*  $:: 'a::\text{heap array-list} \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow 'a \text{ array-list Heap where}$   
 $\text{arl-set} \equiv \lambda(a,n) \ i \ x. \text{ do } \{ a \leftarrow \text{Array.upd } i \ x \ a; \text{ return } (a,n) \}$

**lemma** *arl-empty-rule*[sep-heap-rules]:  $\langle \text{emp} \rangle \text{ arl-empty} \langle \text{is-array-list } [] \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *arl-empty-sz-rule*[sep-heap-rules]:  $\langle \text{emp} \rangle \text{ arl-empty-sz } N \langle \text{is-array-list} \ [] \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *arl-copy-rule*[sep-heap-rules]:  $\langle \text{is-array-list } l \ a \rangle \text{ arl-copy } a \langle \lambda r. \text{ is-array-list } l \ a * \text{ is-array-list } l \ r \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *arl-append-rule*[sep-heap-rules]:  
 $\langle \text{is-array-list } l \ a \rangle$   
 $\text{arl-append } a \ x$

$\langle \lambda a. \text{is-array-list } (l@[x]) \ a \ \rangle_t$   
 $\langle \text{proof} \rangle$

**lemma** *arl-length-rule*[*sep-heap-rules*]:  
 $\langle \text{is-array-list } l \ a \rangle$   
 $\text{arl-length } a$   
 $\langle \lambda r. \text{is-array-list } l \ a \ * \ \uparrow(r=\text{length } l) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *arl-is-empty-rule*[*sep-heap-rules*]:  
 $\langle \text{is-array-list } l \ a \rangle$   
 $\text{arl-is-empty } a$   
 $\langle \lambda r. \text{is-array-list } l \ a \ * \ \uparrow(r \longleftrightarrow (l=[])) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *arl-last-rule*[*sep-heap-rules*]:  
 $l \neq [] \implies$   
 $\langle \text{is-array-list } l \ a \rangle$   
 $\text{arl-last } a$   
 $\langle \lambda r. \text{is-array-list } l \ a \ * \ \uparrow(r=\text{last } l) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *arl-butlast-rule*[*sep-heap-rules*]:  
 $l \neq [] \implies$   
 $\langle \text{is-array-list } l \ a \rangle$   
 $\text{arl-butlast } a$   
 $\langle \text{is-array-list } (\text{butlast } l) \ \rangle_t$   
 $\langle \text{proof} \rangle$

**lemma** *arl-get-rule*[*sep-heap-rules*]:  
 $i < \text{length } l \implies$   
 $\langle \text{is-array-list } l \ a \rangle$   
 $\text{arl-get } a \ i$   
 $\langle \lambda r. \text{is-array-list } l \ a \ * \ \uparrow(r=!i) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *arl-set-rule*[*sep-heap-rules*]:  
 $i < \text{length } l \implies$   
 $\langle \text{is-array-list } l \ a \rangle$   
 $\text{arl-set } a \ i \ x$   
 $\langle \text{is-array-list } (l[i:=x]) \rangle$   
 $\langle \text{proof} \rangle$

**definition** *arl-assn*  $A \equiv \text{hr-comp } \text{is-array-list } (\langle \text{the-pure } A \rangle \text{list-rel})$

**lemmas** [*safe-constraint-rules*] = *CN-FALSEI*[*of is-pure arl-assn A for A*]

**lemma** *arl-assn-comp*:  $\text{is-pure } A \implies \text{hr-comp } (\text{arl-assn } A) (\langle B \rangle \text{list-rel}) = \text{arl-assn}$   
 $(\text{hr-comp } A \ B)$



$\langle proof \rangle$

**lemma** *arl-assn-comp'*:  $hr\text{-}comp (arl\text{-}assn\ id\text{-}assn) (\langle B \rangle list\text{-}rel) = arl\text{-}assn (pure\ B)$   
 $\langle proof \rangle$

**context**

**notes** [*fcomp-norm-unfold*] =  $arl\text{-}assn\text{-}def[symmetric] arl\text{-}assn\text{-}comp'$

**notes** [*intro!*] =  $hfrefI\ hn\text{-}refineI[THEN\ hn\text{-}refine\text{-}preI]$

**notes** [*simp*] =  $pure\text{-}def\ hn\text{-}ctxt\text{-}def\ invalid\text{-}assn\text{-}def$

**begin**

**lemma** *arl-empty-hnr-aux*:  $(uncurry0\ arl\text{-}empty, uncurry0 (RETURN\ op\text{-}list\text{-}empty)) \in unit\text{-}assn^k \rightarrow_a is\text{-}array\text{-}list$   
 $\langle proof \rangle$

**sempref-decl-impl** (*no-register*) *arl-empty*: *arl-empty-hnr-aux*  $\langle proof \rangle$

**lemma** *arl-empty-sz-hnr-aux*:  $(uncurry0 (arl\text{-}empty\text{-}sz\ N), uncurry0 (RETURN\ op\text{-}list\text{-}empty)) \in unit\text{-}assn^k \rightarrow_a is\text{-}array\text{-}list$   
 $\langle proof \rangle$

**sempref-decl-impl** (*no-register*) *arl-empty-sz*: *arl-empty-sz-hnr-aux*  $\langle proof \rangle$

**definition** *op-arl-empty*  $\equiv op\text{-}list\text{-}empty$

**definition** *op-arl-empty-sz* (*N::nat*)  $\equiv op\text{-}list\text{-}empty$

**lemma** *arl-copy-hnr-aux*:  $(arl\text{-}copy, RETURN\ o\ op\text{-}list\text{-}copy) \in is\text{-}array\text{-}list^k \rightarrow_a is\text{-}array\text{-}list$   
 $\langle proof \rangle$

**sempref-decl-impl** *arl-copy*: *arl-copy-hnr-aux*  $\langle proof \rangle$

**lemma** *arl-append-hnr-aux*:  $(uncurry\ arl\text{-}append, uncurry (RETURN\ oo\ op\text{-}list\text{-}append)) \in (is\text{-}array\text{-}list^d * _a\ id\text{-}assn^k) \rightarrow_a is\text{-}array\text{-}list$   
 $\langle proof \rangle$

**sempref-decl-impl** *arl-append*: *arl-append-hnr-aux*  $\langle proof \rangle$

**lemma** *arl-length-hnr-aux*:  $(arl\text{-}length, RETURN\ o\ op\text{-}list\text{-}length) \in is\text{-}array\text{-}list^k \rightarrow_a nat\text{-}assn$   
 $\langle proof \rangle$

**sempref-decl-impl** *arl-length*: *arl-length-hnr-aux*  $\langle proof \rangle$

**lemma** *arl-is-empty-hnr-aux*:  $(arl\text{-}is\text{-}empty, RETURN\ o\ op\text{-}list\text{-}is\text{-}empty) \in is\text{-}array\text{-}list^k \rightarrow_a bool\text{-}assn$   
 $\langle proof \rangle$

**sempref-decl-impl** *arl-is-empty*: *arl-is-empty-hnr-aux*  $\langle proof \rangle$

**lemma** *arl-last-hnr-aux*:  $(arl\text{-}last, RETURN\ o\ op\text{-}list\text{-}last) \in [pre\text{-}list\text{-}last]_a\ is\text{-}array\text{-}list^k \rightarrow id\text{-}assn$   
 $\langle proof \rangle$

```

sepref-decl-impl arl-last: arl-last-hnr-aux ⟨proof⟩

lemma arl-butlast-hnr-aux: (arl-butlast, RETURN o op-list-butlast) ∈ [pre-list-butlast]a
is-array-listd → is-array-list
  ⟨proof⟩
sepref-decl-impl arl-butlast: arl-butlast-hnr-aux ⟨proof⟩

lemma arl-get-hnr-aux: (uncurry2 arl-get, uncurry2 (RETURN oo op-list-get)) ∈
[λ(l, i). i < length l]a (is-array-listk *a nat-assnk) → id-assn
  ⟨proof⟩
sepref-decl-impl arl-get: arl-get-hnr-aux ⟨proof⟩

lemma arl-set-hnr-aux: (uncurry2 arl-set, uncurry2 (RETURN ooo op-list-set))
∈ [λ((l, i), -). i < length l]a (is-array-listd *a nat-assnk *a id-assnk) → is-array-list
  ⟨proof⟩
sepref-decl-impl arl-set: arl-set-hnr-aux ⟨proof⟩

sepref-definition arl-swap is uncurry2 mop-list-swap :: ((arl-assn id-assn)d *a
nat-assnk *a nat-assnk →a arl-assn id-assn)
  ⟨proof⟩
sepref-decl-impl (ismop) arl-swap: arl-swap.refine ⟨proof⟩
end

interpretation arl: list-custom-empty arl-assn A arl-empty op-arl-empty
  ⟨proof⟩

lemma [def-pat-rules]: op-arl-empty-sz $N ≡ UNPROTECT (op-arl-empty-sz N)
  ⟨proof⟩
interpretation arl-sz: list-custom-empty arl-assn A arl-empty-sz N PR-CONST
(op-arl-empty-sz N)
  ⟨proof⟩

end

```

### 3.7 Implementation of Heaps with Arrays

```

theory IICF-Impl-Heap
imports
  IICF-Abs-Heap
  ../IICF-HOL-List
  ../IICF-Array-List
  HOL-Library.Rewrite
begin

```

We implement the heap data structure by an array. The implementation is automatically synthesized by the Sepref-tool.

### 3.7.1 Setup of the Sepref-Tool

```

context
  fixes prio :: 'a::{heap,default} ⇒ 'b::linorder
begin
  interpretation heapstruct prio ⟨proof⟩
  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 ⟨proof⟩

abbreviation rel ≡ arl-assn id-assn

sepref-register prio
lemma [sepref-import-param]: (prio,prio) ∈ Id → Id ⟨proof⟩

lemma [sepref-import-param]:
  ((≤), (≤)::'b ⇒ -) ∈ Id → Id → bool-rel
  ((<), (<)::'b ⇒ -) ∈ Id → Id → bool-rel
  ⟨proof⟩

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
  ⟨proof⟩

end

```

**context**  
**fixes**  $prio :: 'a::\{heap,default\} \Rightarrow 'b::linorder$   
**begin**

**interpretation**  $heapstruct-impl\ prio \langle proof \rangle$

### 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.

**sepref-definition**  $update-impl$  **is**  $uncurry2\ update-op :: rel^d *_{a}\ nat-assn^k *_{a}\ id-assn^k \rightarrow_a rel$   
 $\langle proof \rangle$   
**lemmas**  $[sepref-fr-rules] = update-impl.refine$

**sepref-definition**  $val-of-impl$  **is**  $uncurry\ val-of-op :: rel^k *_{a}\ nat-assn^k \rightarrow_a id-assn$   
 $\langle proof \rangle$   
**lemmas**  $[sepref-fr-rules] = val-of-impl.refine$

**sepref-definition**  $exch-impl$  **is**  $uncurry2\ exch-op :: rel^d *_{a}\ nat-assn^k *_{a}\ nat-assn^k \rightarrow_a rel$   
 $\langle proof \rangle$   
**lemmas**  $[sepref-fr-rules] = exch-impl.refine$

**sepref-definition**  $valid-impl$  **is**  $uncurry\ (RETURN\ oo\ valid) :: rel^k *_{a}\ nat-assn^k \rightarrow_a bool-assn$   
 $\langle proof \rangle$   
**lemmas**  $[sepref-fr-rules] = valid-impl.refine$

**sepref-definition**  $prio-of-impl$  **is**  $uncurry\ (PR-CONST\ prio-of-op) :: rel^k *_{a}\ nat-assn^k \rightarrow_a id-assn$   
 $\langle proof \rangle$   
**lemmas**  $[sepref-fr-rules] = prio-of-impl.refine$

**sepref-definition**  $swim-impl$  **is**  $uncurry\ (PR-CONST\ swim-op) :: rel^d *_{a}\ nat-assn^k \rightarrow_a rel$   
 $\langle proof \rangle$

**lemmas**  $[sepref-fr-rules] = swim-impl.refine$

**sepref-definition**  $sink-impl$  **is**  $uncurry\ (PR-CONST\ sink-op) :: rel^d *_{a}\ nat-assn^k \rightarrow_a rel$   
 $\langle proof \rangle$   
**lemmas**  $[sepref-fr-rules] = sink-impl.refine$

**lemmas**  $[fcomp-norm-unfold] = heap-rel-def[symmetric]$

**sepref-definition**  $empty-impl$  **is**  $uncurry0\ empty-op :: unit-assn^k \rightarrow_a rel$

*<proof>*

**sepref-decl-impl** (*no-register*) *heap-empty*: *empty-impl.refine*[*FCOMP empty-op-refine*]  
*<proof>*

**sepref-definition** *is-empty-impl* **is** *is-empty-op* ::  $rel^k \rightarrow_a bool\text{-}assn$   
*<proof>*

**sepref-decl-impl** *heap-is-empty*: *is-empty-impl.refine*[*FCOMP is-empty-op-refine*]  
*<proof>*

**sepref-definition** *insert-impl* **is** *uncurry insert-op* ::  $id\text{-}assn^k *_a rel^d \rightarrow_a rel$   
*<proof>*

**sepref-decl-impl** *heap-insert*: *insert-impl.refine*[*FCOMP insert-op-refine*] *<proof>*

**sepref-definition** *pop-min-impl* **is** *pop-min-op* ::  $rel^d \rightarrow_a prod\text{-}assn\ id\text{-}assn\ rel$   
*<proof>*

**sepref-decl-impl** (*no-mop*) *heap-pop-min*: *pop-min-impl.refine*[*FCOMP pop-min-op-refine*]  
*<proof>*

**sepref-definition** *peek-min-impl* **is** *peek-min-op* ::  $rel^k \rightarrow_a id\text{-}assn$   
*<proof>*

**sepref-decl-impl** (*no-mop*) *heap-peek-min*: *peek-min-impl.refine*[*FCOMP peek-min-op-refine*]  
*<proof>*

**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*  
*<proof>*

### 3.7.3 Regression Test

**export-code** *empty-impl is-empty-impl insert-impl pop-min-impl peek-min-impl*  
**checking** *SML*

**definition** *sort-by-prio prio l*  $\equiv$  *do* {  
  *q*  $\leftarrow$  *nfoldli l* ( $\lambda\cdot. True$ ) ( $\lambda x q. mop\text{-}mset\text{-}insert\ x\ q$ ) *heap-custom-empty*;  
  (*l, q*)  $\leftarrow$  *WHILET* ( $\lambda(l, q). \neg op\text{-}mset\text{-}is\text{-}empty\ q$ ) ( $\lambda(l, q). do$  {  
    (*x, q*)  $\leftarrow$  *mop-prio-pop-min prio q*;  
    *RETURN* (*l@[x], q*)  
  }) (*op-arl-empty, q*);  
  *RETURN l*  
}

```

context fixes prio:: 'a::{default,heap} ⇒ 'b::linorder begin
sepredef-definition sort-impl is
  sort-by-prio prio :: (list-assn (id-assn::'a::{default,heap} ⇒ -))k →a arl-assn
  id-assn
  ⟨proof⟩
end
definition sort-impl-nat ≡ sort-impl (id::nat⇒nat)

export-code sort-impl checking SML

⟨ML⟩

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 V]] ⇒ ⟨K,V⟩map-rel = K → ⟨V⟩option-rel
  ⟨proof⟩

```

```

lemma map-rel-Id[simp]: ⟨Id,Id⟩map-rel = Id
  ⟨proof⟩

```

```

lemma map-rel-empty1-simp[simp]:
  (Map.empty,m) ∈ ⟨K,V⟩map-rel ⇔ m = Map.empty
  ⟨proof⟩

```

```

lemma map-rel-empty2-simp[simp]:
  (m,Map.empty) ∈ ⟨K,V⟩map-rel ⇔ m = Map.empty
  ⟨proof⟩

```

```

lemma map-rel-obtain1:
  assumes 1: (m,n) ∈ ⟨K,V⟩map-rel
  assumes 2: n l = Some w
  obtains k v where m k = Some v (k,l) ∈ K (v,w) ∈ V
  ⟨proof⟩

```

**lemma** *map-rel-obtain2*:

**assumes** 1:  $(m,n) \in \langle K, V \rangle \text{map-rel}$

**assumes** 2:  $m \ k = \text{Some } v$

**obtains**  $l \ w$  **where**  $n \ l = \text{Some } w \ (k,l) \in K \ (v,w) \in V$

$\langle \text{proof} \rangle$

**lemma** *param-dom[param]*:  $(\text{dom}, \text{dom}) \in \langle K, V \rangle \text{map-rel} \rightarrow \langle K \rangle \text{set-rel}$

$\langle \text{proof} \rangle$

### 3.8.2 Interface Type

**sempref-decl-intf**  $(k, v)$  *i-map* **is**  $k \rightarrow v$

**lemma** [*synth-rules*]:  $\llbracket \text{INTF-OF-REL } K \text{ TYPE}(k); \text{INTF-OF-REL } V \text{ TYPE}(v) \rrbracket$

$\implies \text{INTF-OF-REL } (\langle K, V \rangle \text{map-rel}) \text{ TYPE}((k, v) \text{ i-map}) \langle \text{proof} \rangle$

### 3.8.3 Operations

**sempref-decl-op** *map-empty*:  $\text{Map.empty} :: \langle K, V \rangle \text{map-rel} \langle \text{proof} \rangle$

**sempref-decl-op** *map-is-empty*:  $(=) \text{Map.empty} :: \langle K, V \rangle \text{map-rel} \rightarrow \text{bool-rel} \langle \text{proof} \rangle$

**sempref-decl-op** *map-update*:  $\lambda k \ v \ m. \ m(k \mapsto v) :: K \rightarrow V \rightarrow \langle K, V \rangle \text{map-rel} \rightarrow \langle K, V \rangle \text{map-rel}$

**where** *single-valued*  $K$  *single-valued*  $(K^{-1})$

$\langle \text{proof} \rangle$

**sempref-decl-op** *map-delete*:  $\lambda k \ m. \ \text{fun-upd } m \ k \ \text{None} :: K \rightarrow \langle K, V \rangle \text{map-rel} \rightarrow \langle K, V \rangle \text{map-rel}$

**where** *single-valued*  $K$  *single-valued*  $(K^{-1})$

$\langle \text{proof} \rangle$

**sempref-decl-op** *map-lookup*:  $\lambda k \ (m :: k \rightarrow v). \ m \ k :: K \rightarrow \langle K, V \rangle \text{map-rel} \rightarrow \langle V \rangle \text{option-rel} \langle \text{proof} \rangle$

**lemma** *in-dom-alt*:  $k \in \text{dom } m \iff \neg \text{is-None } (m \ k) \langle \text{proof} \rangle$

**sempref-decl-op** *map-contains-key*:  $\lambda k \ m. \ k \in \text{dom } m :: K \rightarrow \langle K, V \rangle \text{map-rel} \rightarrow \text{bool-rel}$

$\langle \text{proof} \rangle$

### 3.8.4 Patterns

**lemma** *pat-map-empty[pat-rules]*:  $\lambda_2 \cdot \text{None} \equiv \text{op-map-empty} \langle \text{proof} \rangle$

**lemma** *pat-map-is-empty[pat-rules]*:

$(=) \ \$m \ (\lambda_2 \cdot \text{None}) \equiv \text{op-map-is-empty} \ \$m$

(=)  $(\lambda_2-. \text{None})m \equiv \text{op-map-is-empty}m$   
 (=)  $(\text{dom}m)\{\} \equiv \text{op-map-is-empty}m$   
 (=)  $\{\}(\text{dom}m) \equiv \text{op-map-is-empty}m$   
 $\langle \text{proof} \rangle$

**lemma** *pat-map-update*[pat-rules]:

$\text{fun-upd}m\$k\$(\text{Some}v) \equiv \text{op-map-update}'k\$'v\$'m$   
 $\langle \text{proof} \rangle$

**lemma** *pat-map-lookup*[pat-rules]:  $m\$k \equiv \text{op-map-lookup}'k\$'m$

$\langle \text{proof} \rangle$

**lemma** *op-map-delete-pat*[pat-rules]:

$(|\wedge) \$ m \$ (\text{uminus} \$ (\text{insert} \$ k \$ \{\})) \equiv \text{op-map-delete}'k\$'m$   
 $\text{fun-upd}m\$k\$\text{None} \equiv \text{op-map-delete}'k\$'m$   
 $\langle \text{proof} \rangle$

**lemma** *op-map-contains-key*[pat-rules]:

$(\in) \$ k \$ (\text{dom}m) \equiv \text{op-map-contains-key}'k\$'m$   
 $\text{Not}(\text{op-map-contains-key}'k\$'m) \equiv \text{op-map-contains-key}'k\$'m$   
 $\langle \text{proof} \rangle$

### 3.8.5 Parametricity

**locale** *map-custom-empty* =

**fixes** *op-custom-empty* ::  $'k \rightarrow 'v$

**assumes** *op-custom-empty-def*:  $\text{op-custom-empty} = \text{op-map-empty}$

**begin**

**sempref-register** *op-custom-empty* ::  $('kx, 'vx) \text{ i-map}$

**lemma** *fold-custom-empty*:

$\text{Map.empty} = \text{op-custom-empty}$

$\text{op-map-empty} = \text{op-custom-empty}$

$\text{mop-map-empty} = \text{RETURN op-custom-empty}$

$\langle \text{proof} \rangle$

**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*:

$(\text{uncurry } f, \text{uncurry } g) \in A \times_r B \rightarrow R \longleftrightarrow (f, g) \in A \rightarrow B \rightarrow R$

$\langle \text{proof} \rangle$



**lemma** *uncurry0-fun-rel-conv*:

$(\text{uncurry0 } f, \text{uncurry0 } g) \in \text{unit-rel} \rightarrow R \longleftrightarrow (f, g) \in R$   
 $\langle \text{proof} \rangle$

**lemma** *RETURN-rel-conv0*:  $(\text{RETURN } f, \text{RETURN } g) \in \langle A \rangle \text{nres-rel} \longleftrightarrow (f, g) \in A$   
 $\langle \text{proof} \rangle$

**lemma** *RETURN-rel-conv1*:  $(\text{RETURN } o f, \text{RETURN } o g) \in A \rightarrow \langle B \rangle \text{nres-rel}$   
 $\longleftrightarrow (f, g) \in A \rightarrow B$   
 $\langle \text{proof} \rangle$

**lemma** *RETURN-rel-conv2*:  $(\text{RETURN } oo f, \text{RETURN } oo g) \in A \rightarrow B \rightarrow \langle R \rangle \text{nres-rel}$   
 $\longleftrightarrow (f, g) \in A \rightarrow B \rightarrow R$   
 $\langle \text{proof} \rangle$

**lemma** *RETURN-rel-conv3*:  $(\text{RETURN } ooo f, \text{RETURN } ooo g) \in A \rightarrow B \rightarrow C \rightarrow \langle R \rangle \text{nres-rel}$   
 $\longleftrightarrow (f, g) \in A \rightarrow B \rightarrow C \rightarrow R$   
 $\langle \text{proof} \rangle$

**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-ncD, 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* ::  $[\lambda((k, v), m). k \notin \text{dom } m]_f (K \times_r V) \times_r \langle K, V \rangle \text{map-rel}$   
 $\rightarrow \langle K, V \rangle \text{map-rel}$

**where** *single-valued K single-valued (K<sup>-1</sup>)*  $\langle \text{proof} \rangle$

**sepref-decl-op** *map-update-ex*: *op-map-update* ::  $[\lambda((k, v), m). k \in \text{dom } m]_f (K \times_r V) \times_r \langle K, V \rangle \text{map-rel}$   
 $\rightarrow \langle K, V \rangle \text{map-rel}$

**where** *single-valued K single-valued (K<sup>-1</sup>)*  $\langle \text{proof} \rangle$

**sepref-decl-op** *map-delete-ex*: *op-map-delete* ::  $[\lambda(k, m). k \in \text{dom } m]_f K \times_r \langle K, V \rangle \text{map-rel}$   
 $\rightarrow \langle K, V \rangle \text{map-rel}$

**where** *single-valued K single-valued (K<sup>-1</sup>)*  $\langle \text{proof} \rangle$

**context**

**fixes** *prio* :: 'v  $\Rightarrow$  'p::linorder

**begin**

**sepref-decl-op** *pm-decrease-key: op-map-update*  
 $:: [\lambda((k,v),m). k \in \text{dom } m \wedge \text{prio } v \leq \text{prio } (the (m k))]_f (K \times_r V) \times_r \langle K, V \rangle \text{map-rel}$   
 $\rightarrow \langle K, (V :: ('v \times 'v) \text{ set}) \rangle \text{map-rel}$   
**where** *single-valued K single-valued (K<sup>-1</sup>) IS-BELOW-ID V*  
 $\langle \text{proof} \rangle$

**sepref-decl-op** *pm-increase-key: op-map-update*  
 $:: [\lambda((k,v),m). k \in \text{dom } m \wedge \text{prio } v \geq \text{prio } (the (m k))]_f (K \times_r V) \times_r \langle K, V \rangle \text{map-rel}$   
 $\rightarrow \langle K, (V :: ('v \times 'v) \text{ set}) \rangle \text{map-rel}$   
**where** *single-valued K single-valued (K<sup>-1</sup>) IS-BELOW-ID V*  
 $\langle \text{proof} \rangle$

**lemma** *IS-BELOW-ID-D: (a,b) ∈ R ⇒ IS-BELOW-ID R ⇒ a=b*  $\langle \text{proof} \rangle$

**sepref-decl-op** *pm-peek-min: λm. SPEC (λ(k,v).*  
 $m k = \text{Some } v \wedge (\forall k' v'. m k' = \text{Some } v' \longrightarrow \text{prio } v \leq \text{prio } v')$   
 $:: [\text{Not } o \text{ op-map-is-empty}]_f \langle K, V \rangle \text{map-rel} \rightarrow K \times_r (V :: ('v \times 'v) \text{ set})$   
**where** *IS-BELOW-ID V*  
 $\langle \text{proof} \rangle$   
**apply1** (*intro exI conjI allI impI; assumption?*)  
 $\langle \text{proof} \rangle$

**sepref-decl-op** *pm-pop-min: λm. SPEC (λ((k,v),m').*  
 $m k = \text{Some } v$   
 $\wedge m' = \text{op-map-delete } k m$   
 $\wedge (\forall k' v'. m k' = \text{Some } v' \longrightarrow \text{prio } v \leq \text{prio } v')$   
 $) :: [\text{Not } o \text{ op-map-is-empty}]_f \langle K, V \rangle \text{map-rel} \rightarrow (K \times_r (V :: ('v \times 'v) \text{ set})) \times_r \langle K, V \rangle \text{map-rel}$   
**where** *single-valued K single-valued (K<sup>-1</sup>) IS-BELOW-ID V*  
 $\langle \text{proof} \rangle$   
**applyS** *parametricity*  
 $\langle \text{proof} \rangle$   
**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**  $('k, 'v) \text{ ahm} = 'k \text{ list} \times ('k \rightarrow 'v)$

### 3.10.1 Basic Setup

First, we define a mapping to list-based heaps

**definition**  $\text{hmr-}\alpha :: ('k, 'v) \text{ ahm} \Rightarrow 'v \text{ heap}$  **where**

$hmr-\alpha \equiv \lambda(pq,m). \text{map } (the \ o \ m) \ pq$

**definition**  $hmr-invar \equiv \lambda(pq,m). \text{distinct } pq \wedge \text{dom } m = \text{set } pq$

**definition**  $hmr-rel \equiv br \ hmr-\alpha \ hmr-invar$

**lemmas**  $hmr-rel-defs = hmr-rel-def \ br-def \ hmr-\alpha-def \ hmr-invar-def$

**lemma**  $hmr-empty-invar[simp]: hmr-invar \ (\[], \text{Map.empty})$   
 $\langle proof \rangle$

**locale**  $hmstruct = h: \text{heapstruct } prio \ \text{for } prio :: 'v \Rightarrow 'b::\text{linorder}$   
**begin**

Next, we define a mapping to priority maps.

**definition**  $heapmap-\alpha :: ('k, 'v) \text{ahm} \Rightarrow ('k \rightarrow 'v) \ \text{where}$   
 $heapmap-\alpha \equiv \lambda(pq,m). \ m$

**definition**  $heapmap-invar :: ('k, 'v) \text{ahm} \Rightarrow \text{bool} \ \text{where}$   
 $heapmap-invar \equiv \lambda hm. \ hmr-invar \ hm \wedge h.\text{heap-invar} \ (hmr-\alpha \ hm)$

**definition**  $heapmap-rel \equiv br \ heapmap-\alpha \ heapmap-invar$

**lemmas**  $heapmap-rel-defs = heapmap-rel-def \ br-def \ heapmap-\alpha-def \ heapmap-invar-def$

**lemma**  $[\text{refine-dref-RELATES}]: \text{RELATES } hmr-rel \ \langle proof \rangle$

**lemma**  $h\text{-heap-invarI}[simp]: \text{heapmap-invar } hm \Longrightarrow h.\text{heap-invar} \ (hmr-\alpha \ hm)$   
 $\langle proof \rangle$

**lemma**  $hmr-invarI[simp]: \text{heapmap-invar } hm \Longrightarrow hmr-invar \ hm$   
 $\langle proof \rangle$

**lemma**  $set-hmr-\alpha[simp]: hmr-invar \ hm \Longrightarrow \text{set} \ (hmr-\alpha \ hm) = \text{ran} \ (\text{heapmap-\alpha} \ hm)$   
 $\langle proof \rangle$

**lemma**  $in-h-hmr-\alpha\text{-conv}[simp]: hmr-invar \ hm \Longrightarrow x \in\# \ h.\alpha \ (hmr-\alpha \ hm) \longleftrightarrow$   
 $x \in \text{ran} \ (\text{heapmap-\alpha} \ hm)$   
 $\langle proof \rangle$

### 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**  $hm\text{-length} :: ('k, 'v) \text{ahm} \Rightarrow \text{nat}$  **where**  
 $hm\text{-length} \equiv \lambda(pq, -). \text{length } pq$

**lemma**  $hm\text{-length-refine}: (hm\text{-length}, \text{length}) \in hmr\text{-rel} \rightarrow \text{nat-rel}$   
 $\langle \text{proof} \rangle$

**lemma**  $hm\text{-length-hmr-}\alpha[simp]: \text{length } (hmr\text{-}\alpha \text{ hm}) = hm\text{-length } hm$   
 $\langle \text{proof} \rangle$

**lemmas**  $[refine] = hm\text{-length-refine}[param\text{-}fo]$

#### Valid

Check whether index is valid

**definition**  $hm\text{-valid } hm \ i \equiv i > 0 \wedge i \leq hm\text{-length } hm$

**lemma**  $hm\text{-valid-refine}: (hm\text{-valid}, h.\text{valid}) \in hmr\text{-rel} \rightarrow \text{nat-rel} \rightarrow \text{bool-rel}$   
 $\langle \text{proof} \rangle$

**lemma**  $hm\text{-valid-hmr-}\alpha[simp]: h.\text{valid } (hmr\text{-}\alpha \text{ hm}) = hm\text{-valid } hm$   
 $\langle \text{proof} \rangle$

#### Key-Of

**definition**  $hm\text{-key-of} :: ('k, 'v) \text{ahm} \Rightarrow \text{nat} \Rightarrow 'k$  **where**  
 $hm\text{-key-of} \equiv \lambda(pq, m) \ i. pq!(i - 1)$

**definition**  $hm\text{-key-of-op} :: ('k, 'v) \text{ahm} \Rightarrow \text{nat} \Rightarrow 'k \ \text{nres}$  **where**  
 $hm\text{-key-of-op} \equiv \lambda(pq, m) \ i. \text{ASSERT } (i > 0) \gg mop\text{-list-get } pq \ (i - 1)$

**lemma**  $hm\text{-key-of-op-unfold}:$

**shows**  $hm\text{-key-of-op } hm \ i = \text{ASSERT } (hm\text{-valid } hm \ i) \gg \text{RETURN } (hm\text{-key-of } hm \ i)$   
 $\langle \text{proof} \rangle$

**lemma**  $val\text{-of-hmr-}\alpha[simp]: hm\text{-valid } hm \ i \Longrightarrow h.\text{val-of } (hmr\text{-}\alpha \text{ hm}) \ i$   
 $= \text{the } (heapmap\text{-}\alpha \text{ hm } (hm\text{-key-of } hm \ i))$   
 $\langle \text{proof} \rangle$

**lemma**  $hm\text{-}\alpha\text{-key-ex}[simp]:$

$\llbracket \text{hmr-invar } hm; \text{hm-valid } hm \ i \rrbracket \implies (\text{heapmap-}\alpha \text{ } hm \ (\text{hm-key-of } hm \ i) \neq \text{None})$   
 $\langle \text{proof} \rangle$

## Lookup

**abbreviation**  $(input) \text{ hm-lookup}$  **where**  $\text{hm-lookup} \equiv \text{heapmap-}\alpha$

**definition**  $\text{hm-the-lookup-op } hm \ k \equiv$   
 $\text{ASSERT } (\text{heapmap-}\alpha \text{ } hm \ k \neq \text{None} \wedge \text{hmr-invar } hm)$   
 $\gg \text{RETURN } (\text{the } (\text{heapmap-}\alpha \text{ } hm \ k))$

## Exchange

Exchange two indices

**definition**  $\text{hm-exch-op} \equiv \lambda(pq,m) \ i \ j. \text{ do } \{$   
 $\text{ASSERT } (\text{hm-valid } (pq,m) \ i);$   
 $\text{ASSERT } (\text{hm-valid } (pq,m) \ j);$   
 $\text{ASSERT } (\text{hmr-invar } (pq,m));$   
 $pq \leftarrow \text{mop-list-swap } pq \ (i - 1) \ (j - 1);$   
 $\text{RETURN } (pq,m)$   
 $\}$

**lemma**  $\text{hm-exch-op-invar}: \text{hm-exch-op } hm \ i \ j \leq_n \text{ SPEC } \text{hmr-invar}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{hm-exch-op-refine}: (\text{hm-exch-op}, \text{h.exch-op}) \in \text{hmr-rel} \rightarrow \text{nat-rel} \rightarrow$   
 $\text{nat-rel} \rightarrow \langle \text{hmr-rel} \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**lemmas**  $\text{hm-exch-op-refine}'[\text{refine}] = \text{hm-exch-op-refine}[\text{param-fo}, \text{THEN nres-relD}]$

**definition**  $\text{hm-exch} :: ('k, 'v) \text{ ahm} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow ('k, 'v) \text{ ahm}$   
**where**  $\text{hm-exch} \equiv \lambda(pq,m) \ i \ j. (\text{swap } pq \ (i-1) \ (j-1), m)$

**lemma**  $\text{hm-exch-op-}\alpha\text{-correct}: \text{hm-exch-op } hm \ i \ j \leq_n \text{ SPEC } (\lambda hm'.$   
 $\text{hm-valid } hm \ i \wedge \text{hm-valid } hm \ j \wedge \text{hm}' = \text{hm-exch } hm \ i \ j$   
 $)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{hm-exch-}\alpha[\text{simp}]: \text{heapmap-}\alpha \ (\text{hm-exch } hm \ i \ j) = (\text{heapmap-}\alpha \text{ } hm)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{hm-exch-valid}[\text{simp}]: \text{hm-valid } (\text{hm-exch } hm \ i \ j) = \text{hm-valid } hm$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{hm-exch-length}[\text{simp}]: \text{hm-length } (\text{hm-exch } hm \ i \ j) = \text{hm-length } hm$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{hm-exch-same}[\text{simp}]: \text{hm-exch } hm \ i \ i = hm$   
 $\langle \text{proof} \rangle$

**lemma** *hm-key-of-exch-conv*[simp]:  
 $\llbracket \text{hm-valid hm } i; \text{hm-valid hm } j; \text{hm-valid hm } k \rrbracket \implies$   
 $\text{hm-key-of (hm-exch hm } i \ j) \ k = ($   
 $\text{if } k=i \text{ then hm-key-of hm } j$   
 $\text{else if } k=j \text{ then hm-key-of hm } i$   
 $\text{else hm-key-of hm } k$   
 $)$   
 $\langle \text{proof} \rangle$

**lemma** *hm-key-of-exch-matching*[simp]:  
 $\llbracket \text{hm-valid hm } i; \text{hm-valid hm } j \rrbracket \implies \text{hm-key-of (hm-exch hm } i \ j) \ i = \text{hm-key-of}$   
 $\text{hm } j$   
 $\llbracket \text{hm-valid hm } i; \text{hm-valid hm } j \rrbracket \implies \text{hm-key-of (hm-exch hm } i \ j) \ j = \text{hm-key-of}$   
 $\text{hm } i$   
 $\langle \text{proof} \rangle$

## Index

Obtaining the index of a key

**definition** *hm-index*  $\equiv \lambda(pq,m) \ k. \text{index } pq \ k + 1$

**lemma** *hm-index-valid*[simp]:  $\llbracket \text{hmr-invar hm}; \text{heapmap-}\alpha \ \text{hm } k \neq \text{None} \rrbracket \implies$   
 $\text{hm-valid hm (hm-index hm } k)$   
 $\langle \text{proof} \rangle$

**lemma** *hm-index-key-of*[simp]:  $\llbracket \text{hmr-invar hm}; \text{heapmap-}\alpha \ \text{hm } k \neq \text{None} \rrbracket \implies$   
 $\text{hm-key-of hm (hm-index hm } k) = k$   
 $\langle \text{proof} \rangle$

**definition** *hm-index-op*  $\equiv \lambda(pq,m) \ k.$   
 $do \{$   
 $\text{ASSERT (hmr-invar } (pq,m) \wedge \text{heapmap-}\alpha \ (pq,m) \ k \neq \text{None});$   
 $i \leftarrow \text{mop-list-index } pq \ k;$   
 $\text{RETURN } (i+1)$   
 $\}$

**lemma** *hm-index-op-correct*:  
**assumes** *hmr-invar hm*  
**assumes** *heapmap- $\alpha$  hm k  $\neq$  None*  
**shows** *hm-index-op hm k  $\leq$  SPEC ( $\lambda r. r = \text{hm-index hm } k$ )*  
 $\langle \text{proof} \rangle$

**lemmas** [*refine-vcg*] = *hm-index-op-correct*

## Update

Updating the heap at an index

**definition**  $hm\text{-update-op} :: ('k, 'v) \text{ ahm} \Rightarrow \text{nat} \Rightarrow 'v \Rightarrow ('k, 'v) \text{ ahm nres}$  **where**  
 $hm\text{-update-op} \equiv \lambda(pq, m) i v. \text{ do } \{$   
 $\text{ASSERT } (hm\text{-valid } (pq, m) i \wedge hmr\text{-invar } (pq, m));$   
 $k \leftarrow mop\text{-list-get } pq (i - 1);$   
 $\text{RETURN } (pq, m(k \mapsto v))$   
 $\}$

**lemma**  $hm\text{-update-op-invar}: hm\text{-update-op } hm \ k \ v \ \leq_n \ \text{SPEC } hmr\text{-invar}$   
 $\langle \text{proof} \rangle$

**lemma**  $hm\text{-update-op-refine}: (hm\text{-update-op}, h.\text{update-op}) \in hmr\text{-rel} \rightarrow \text{nat-rel}$   
 $\rightarrow Id \rightarrow \langle hmr\text{-rel} \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**lemmas**  $[\text{refine}] = hm\text{-update-op-refine}[\text{param-fo}, \text{THEN nres-relD}]$

**lemma**  $hm\text{-update-op-}\alpha\text{-correct}:$   
**assumes**  $hmr\text{-invar } hm$   
**assumes**  $heapmap\text{-}\alpha \ hm \ k \neq \text{None}$   
**shows**  $hm\text{-update-op } hm \ (hm\text{-index } hm \ k) \ v \ \leq_n \ \text{SPEC } (\lambda hm'. heapmap\text{-}\alpha \ hm'$   
 $= (heapmap\text{-}\alpha \ hm)(k \mapsto v))$   
 $\langle \text{proof} \rangle$

## Butlast

Remove last element

**definition**  $hm\text{-butlast-op} :: ('k, 'v) \text{ ahm} \Rightarrow ('k, 'v) \text{ ahm nres}$  **where**  
 $hm\text{-butlast-op} \equiv \lambda(pq, m). \text{ do } \{$   
 $\text{ASSERT } (hmr\text{-invar } (pq, m));$   
 $k \leftarrow mop\text{-list-get } pq (\text{length } pq - 1);$   
 $pq \leftarrow mop\text{-list-butlast } pq;$   
 $\text{let } m = m(k := \text{None});$   
 $\text{RETURN } (pq, m)$   
 $\}$

**lemma**  $hm\text{-butlast-op-refine}: (hm\text{-butlast-op}, h.\text{butlast-op}) \in hmr\text{-rel} \rightarrow \langle hmr\text{-rel} \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**lemmas**  $[\text{refine}] = hm\text{-butlast-op-refine}[\text{param-fo}, \text{THEN nres-relD}]$

**lemma**  $hm\text{-butlast-op-}\alpha\text{-correct}: hm\text{-butlast-op } hm \ \leq_n \ \text{SPEC } ($   
 $\lambda hm'. heapmap\text{-}\alpha \ hm' = (heapmap\text{-}\alpha \ hm)(hm\text{-key-of } hm \ (hm\text{-length } hm) :=$   
 $\text{None} \ ))$   
 $\langle \text{proof} \rangle$

## Append

Append new element at end of heap

**definition**  $hm\text{-append-op} :: ('k, 'v) \text{ahm} \Rightarrow 'k \Rightarrow 'v \Rightarrow ('k, 'v) \text{ahm nres}$   
**where**  $hm\text{-append-op} \equiv \lambda(pq, m) k v. \text{do} \{$   
   $ASSERT (k \notin \text{dom } m);$   
   $ASSERT (hmr\text{-invar } (pq, m));$   
   $pq \leftarrow mop\text{-list-append } pq k;$   
   $\text{let } m = m (k \mapsto v);$   
   $RETURN (pq, m)$   
 $\}$

**lemma**  $hm\text{-append-op-invar}: hm\text{-append-op } hm k v \leq_n SPEC hmr\text{-invar}$   
 $\langle proof \rangle$

**lemma**  $hm\text{-append-op-refine}: \llbracket heapmap\text{-}\alpha \text{ } hm k = None; (hm, h) \in hmr\text{-rel} \rrbracket$   
 $\implies (hm\text{-append-op } hm k v, h.\text{append-op } h v) \in \langle hmr\text{-rel} \rangle nres\text{-rel}$   
 $\langle proof \rangle$

**lemmas**  $hm\text{-append-op-refine}'[refine] = hm\text{-append-op-refine}[param\text{-fo}, THEN$   
 $nres\text{-rel}D]$

**lemma**  $hm\text{-append-op-}\alpha\text{-correct}: hm\text{-append-op } hm k v \leq_n SPEC (\lambda hm'. heapmap\text{-}\alpha \text{ } hm' = (heapmap\text{-}\alpha \text{ } hm)$   
 $(k \mapsto v))$   
 $\langle proof \rangle$

### 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**  $heapmap\text{-hmr-rel}I: (hm, h) \in heapmap\text{-rel} \implies (hm, hmr\text{-}\alpha \text{ } hm) \in hmr\text{-rel}$   
 $\langle proof \rangle$

**lemma**  $heapmap\text{-hmr-rel}I': heapmap\text{-invar } hm \implies (hm, hmr\text{-}\alpha \text{ } hm) \in hmr\text{-rel}$   
 $\langle proof \rangle$

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**  $heapmap\text{-nres-rel}I':$   
**assumes**  $hm \leq \Downarrow hmr\text{-rel } h'$   
**assumes**  $h' \leq SPEC (h.\text{heap-invar})$   
**assumes**  $hm \leq_n SPEC (\lambda hm'. RETURN (heapmap\text{-}\alpha \text{ } hm') \leq h)$   
**shows**  $hm \leq \Downarrow heapmap\text{-rel } h$   
 $\langle proof \rangle$



**lemma** *heapmap-nres-rel''*:  
**assumes**  $hm \leq \Downarrow hmr\text{-rel } h'$   
**assumes**  $h' \leq SPEC \Phi$   
**assumes**  $\bigwedge h'. \Phi h' \implies h.\text{heap-invar } h'$   
**assumes**  $hm \leq_n SPEC (\lambda hm'. RETURN (\text{heapmap-}\alpha \text{ } hm') \leq h)$   
**shows**  $hm \leq \Downarrow \text{heapmap-rel } h$   
 $\langle \text{proof} \rangle$

## Val-of

Indexing into the heap

**definition** *hm-val-of-op* ::  $('k, 'v) \text{ ahm} \Rightarrow \text{nat} \Rightarrow 'v \text{ nres}$  **where**  
 $hm\text{-val-of-op} \equiv \lambda hm \ i. \text{ do } \{$   
 $\quad k \leftarrow hm\text{-key-of-op } hm \ i;$   
 $\quad v \leftarrow hm\text{-the-lookup-op } hm \ k;$   
 $\quad RETURN \ v$   
 $\}$

**lemma** *hm-val-of-op-refine*:  $(hm\text{-val-of-op}, h.\text{val-of-op}) \in (hmr\text{-rel} \rightarrow \text{nat-rel} \rightarrow \langle Id \rangle \text{nres-rel})$   
 $\langle \text{proof} \rangle$

**lemmas**  $[refine] = hm\text{-val-of-op-refine}[param\text{-fo}, THEN \text{nres-rel}D]$

## Prio-of

Priority of key

**definition** *hm-prio-of-op*  $h \ i \equiv \text{do } \{ v \leftarrow hm\text{-val-of-op } h \ i; RETURN \ (prio \ v) \}$

**lemma** *hm-prio-of-op-refine*:  $(hm\text{-prio-of-op}, h.\text{prio-of-op}) \in (hmr\text{-rel} \rightarrow \text{nat-rel} \rightarrow \langle Id \rangle \text{nres-rel})$   
 $\langle \text{proof} \rangle$

**lemmas**  $hm\text{-prio-of-op-refine}'[refine] = hm\text{-prio-of-op-refine}[param\text{-fo}, THEN \text{nres-rel}D]$

## Swim

**definition** *hm-swim-op* ::  $('k, 'v) \text{ ahm} \Rightarrow \text{nat} \Rightarrow ('k, 'v) \text{ ahm nres}$  **where**  
 $hm\text{-swim-op } h \ i \equiv \text{do } \{$   
 $\quad RECT (\lambda swim \ (h, i). \text{ do } \{$   
 $\quad\quad ASSERT \ (hm\text{-valid } h \ i \wedge h.\text{swim-invar } (hmr\text{-}\alpha \text{ } h) \ i);$   
 $\quad\quad \text{if } hm\text{-valid } h \ (h.\text{parent } i) \text{ then do } \{$   
 $\quad\quad\quad ppi \leftarrow hm\text{-prio-of-op } h \ (h.\text{parent } i);$   
 $\quad\quad\quad pi \leftarrow hm\text{-prio-of-op } h \ i;$   
 $\quad\quad\quad \text{if } (\neg ppi \leq pi) \text{ then do } \{$   
 $\quad\quad\quad\quad h \leftarrow hm\text{-exch-op } h \ i \ (h.\text{parent } i);$   
 $\quad\quad\quad\}$   
 $\quad\quad\quad \}$   
 $\quad\quad \}$   
 $\}$

```

    swim (h, h.parent i)
  } else
    RETURN h
  } else
    RETURN h
  }) (h,i)
}

```

**lemma** *hm-swim-op-refine*:  $(hm-swim-op, h.swim-op) \in hmr-rel \rightarrow nat-rel \rightarrow \langle hmr-rel \rangle nres-rel$   
 $\langle proof \rangle$

**lemmas** *hm-swim-op-refine'*[*refine*] = *hm-swim-op-refine*[*param-fo*, *THEN nres-relD*]

**lemma** *hm-swim-op-nofail-imp-valid*:  
 $nofail (hm-swim-op hm i) \implies hm-valid hm i \wedge h.swim-invar (hmr-\alpha hm) i$   
 $\langle proof \rangle$

**lemma** *hm-swim-op- $\alpha$ -correct*:  $hm-swim-op hm i \leq_n SPEC (\lambda hm'. heapmap-\alpha hm' = heapmap-\alpha hm)$   
 $\langle proof \rangle$

## Sink

**definition** *hm-sink-op*

**where**

```

hm-sink-op h k  $\equiv$  RECT ( $\lambda D (h,k)$ . do {
  ASSERT ( $k > 0 \wedge k \leq hm-length h$ );
  let len = hm-length h;
  if ( $2*k \leq len$ ) then do {
    let j = 2*k;
    pj  $\leftarrow$  hm-prio-of-op h j;

    j  $\leftarrow$  (
      if  $j < len$  then do {
        psj  $\leftarrow$  hm-prio-of-op h (Suc j);
        if  $pj > psj$  then RETURN (j+1) else RETURN j
      } else RETURN j);

    pj  $\leftarrow$  hm-prio-of-op h j;
    pk  $\leftarrow$  hm-prio-of-op h k;
    if ( $pk > pj$ ) then do {
      h  $\leftarrow$  hm-exch-op h k j;
      D (h,j)
    } else
      RETURN h
  } else RETURN h
}) (h,k)

```

**lemma** *hm-sink-op-refine*:  $(hm\text{-sink-op}, h.\text{sink-op}) \in hmr\text{-rel} \rightarrow nat\text{-rel} \rightarrow \langle hmr\text{-rel} \rangle nres\text{-rel}$   
 $\langle proof \rangle$

**lemmas** *hm-sink-op-refine'*[*refine*] = *hm-sink-op-refine*[*param-fo*, *THEN nres-relD*]

**lemma** *hm-sink-op-nofail-imp-valid*:  $nofail (hm\text{-sink-op } hm \ i) \implies hm\text{-valid } hm \ i$   
 $\langle proof \rangle$

**lemma** *hm-sink-op- $\alpha$ -correct*:  $hm\text{-sink-op } hm \ i \leq_n SPEC (\lambda hm'. heapmap\text{-}\alpha \ hm' = heapmap\text{-}\alpha \ hm)$   
 $\langle proof \rangle$

## Repair

**definition** *hm-repair-op*  $hm \ i \equiv do \{$   
 $hm \leftarrow hm\text{-sink-op } hm \ i;$   
 $hm \leftarrow hm\text{-swim-op } hm \ i;$   
 $RETURN \ hm$   
 $\}$

**lemma** *hm-repair-op-refine*:  $(hm\text{-repair-op}, h.\text{repair-op}) \in hmr\text{-rel} \rightarrow nat\text{-rel} \rightarrow \langle hmr\text{-rel} \rangle nres\text{-rel}$   
 $\langle proof \rangle$

**lemmas** *hm-repair-op-refine'*[*refine*] = *hm-repair-op-refine*[*param-fo*, *THEN nres-relD*]

**lemma** *hm-repair-op- $\alpha$ -correct*:  $hm\text{-repair-op } hm \ i \leq_n SPEC (\lambda hm'. heapmap\text{-}\alpha \ hm' = heapmap\text{-}\alpha \ hm)$   
 $\langle proof \rangle$

### 3.10.4 Operations

In this section, we define the operations that implement the priority-map interface

#### Empty

**definition** *hm-empty-op* ::  $(k, v) \text{ ahm } nres$   
**where** *hm-empty-op*  $\equiv RETURN \ ([], Map.empty)$

**lemma** *hm-empty-aref*:  $(hm\text{-empty-op}, RETURN \ op\text{-map-empty}) \in \langle heapmap\text{-rel} \rangle nres\text{-rel}$   
 $\langle proof \rangle$

## Insert

**definition**  $hm\text{-insert-op} :: 'k \Rightarrow 'v \Rightarrow ('k, 'v) \text{ ahm} \Rightarrow ('k, 'v) \text{ ahm nres}$  **where**  
 $hm\text{-insert-op} \equiv \lambda k v h. \text{ do } \{$   
   $ASSERT (h.\text{heap-invar} (hmr\text{-}\alpha h));$   
   $h \leftarrow hm\text{-append-op } h k v;$   
   $\text{let } l = hm\text{-length } h;$   
   $h \leftarrow hm\text{-swim-op } h l;$   
   $RETURN h$   
 $\}$

**lemma**  $hm\text{-insert-op-refine[refine]:$   $\llbracket \text{heapmap-}\alpha \text{ hm } k = \text{None}; (hm, h) \in hmr\text{-rel} \rrbracket \implies$   
 $hm\text{-insert-op } k v hm \leq \Downarrow hmr\text{-rel } (h.\text{insert-op } v h)$   
 $\langle \text{proof} \rangle$

**lemma**  $hm\text{-insert-op-aref}:$   
 $(hm\text{-insert-op}, mop\text{-map-update-new}) \in Id \rightarrow Id \rightarrow \text{heapmap-rel} \rightarrow \langle \text{heapmap-rel} \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

## Is-Empty

**lemma**  $hmr\text{-}\alpha\text{-empty-iff[simp]:$   
 $hmr\text{-invar } hm \implies hmr\text{-}\alpha \text{ hm} = [] \iff \text{heapmap-}\alpha \text{ hm} = \text{Map.empty}$   
 $\langle \text{proof} \rangle$

**definition**  $hm\text{-is-empty-op} :: ('k, 'v) \text{ ahm} \Rightarrow \text{bool nres}$  **where**  
 $hm\text{-is-empty-op} \equiv \lambda hm. \text{ do } \{$   
   $ASSERT (hmr\text{-invar } hm);$   
   $\text{let } l = hm\text{-length } hm;$   
   $RETURN (l=0)$   
 $\}$

**lemma**  $hm\text{-is-empty-op-refine}: (hm\text{-is-empty-op}, h.\text{is-empty-op}) \in hmr\text{-rel} \rightarrow$   
 $\langle \text{bool-rel} \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**lemma**  $hm\text{-is-empty-op-aref}: (hm\text{-is-empty-op}, RETURN o \text{ op-map-is-empty})$   
 $\in \text{heapmap-rel} \rightarrow \langle \text{bool-rel} \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

## Lookup

**definition**  $hm\text{-lookup-op} :: 'k \Rightarrow ('k, 'v) \text{ ahm} \Rightarrow 'v \text{ option nres}$   
**where**  $hm\text{-lookup-op} \equiv \lambda k hm. ASSERT (\text{heapmap-invar } hm) \gg RETURN$   
 $(hm\text{-lookup } hm k)$

**lemma**  $hm\text{-lookup-op-aref}: (hm\text{-lookup-op}, RETURN oo \text{ op-map-lookup}) \in Id$   
 $\rightarrow \text{heapmap-rel} \rightarrow \langle \langle Id \rangle \text{option-rel} \rangle \text{nres-rel}$

⟨proof⟩

## Contains-Key

**definition**  $hm\text{-contains-key-op} \equiv \lambda k (pq, m). \text{ASSERT } (heapmap\text{-invar } (pq, m))$   
 $\gg \text{RETURN } (k \in dom\ m)$

**lemma**  $hm\text{-contains-key-op-aref}: (hm\text{-contains-key-op}, \text{RETURN} \text{ oo } op\text{-map-contains-key})$   
 $\in Id \rightarrow heapmap\text{-rel} \rightarrow \langle bool\text{-rel} \rangle nres\text{-rel}$   
⟨proof⟩

## Decrease-Key

**definition**  $hm\text{-decrease-key-op} \equiv \lambda k\ v\ hm. \text{do } \{$   
   $\text{ASSERT } (heapmap\text{-invar } hm);$   
   $\text{ASSERT } (heapmap\text{-}\alpha\ hm\ k \neq \text{None} \wedge prio\ v \leq prio\ (the\ (heapmap\text{-}\alpha\ hm\ k)));$   
   $i \leftarrow hm\text{-index-op } hm\ k;$   
   $hm \leftarrow hm\text{-update-op } hm\ i\ v;$   
   $hm\text{-swim-op } hm\ i$   
   $\}$

**definition** (in  $heapstruct$ )  $decrease\text{-key-op } i\ v\ h \equiv \text{do } \{$   
   $\text{ASSERT } (valid\ h\ i \wedge prio\ v \leq prio\text{-of } h\ i);$   
   $h \leftarrow update\text{-op } h\ i\ v;$   
   $swim\text{-op } h\ i$   
   $\}$

**lemma** (in  $heapstruct$ )  $decrease\text{-key-op-invar}$ :  
 $\llbracket heap\text{-invar } h; valid\ h\ i; prio\ v \leq prio\text{-of } h\ i \rrbracket \implies decrease\text{-key-op } i\ v\ h \leq$   
 $SPEC\ heap\text{-invar}$   
⟨proof⟩

**lemma**  $index\text{-op-inline-refine}$ :

**assumes**  $heapmap\text{-invar } hm$   
**assumes**  $heapmap\text{-}\alpha\ hm\ k \neq \text{None}$   
**assumes**  $f\ (hm\text{-index } hm\ k) \leq m$   
**shows**  $\text{do } \{i \leftarrow hm\text{-index-op } hm\ k; f\ i\} \leq m$   
⟨proof⟩

**lemma**  $hm\text{-decrease-key-op-refine}$ :

$\llbracket (hm, h) \in hmr\text{-rel}; (hm, m) \in heapmap\text{-rel}; m\ k = \text{Some } v \rrbracket$   
 $\implies hm\text{-decrease-key-op } k\ v\ hm \leq \Downarrow hmr\text{-rel } (h.\text{decrease-key-op } (hm\text{-index } hm\ k)\ v\ h)$   
⟨proof⟩

**lemma**  $hm\text{-index-op-inline-leaf}$ :

**assumes**  $f\ (hm\text{-index } hm\ k) \leq_n m$   
**shows**  $\text{do } \{i \leftarrow hm\text{-index-op } hm\ k; f\ i\} \leq_n m$   
⟨proof⟩

**lemma** *hm-decrease-key-op- $\alpha$ -correct*:  
 $heapmap\text{-invar } hm \implies hm\text{-decrease-key-op } k v hm \leq_n SPEC (\lambda hm'. heapmap\text{-}\alpha$   
 $hm' = (heapmap\text{-}\alpha hm)(k \mapsto v))$   
 $\langle proof \rangle$

**lemma** *hm-decrease-key-op-aref*:  
 $(hm\text{-decrease-key-op}, PR\text{-CONST } (mop\text{-}pm\text{-decrease-key } prio)) \in Id \rightarrow Id \rightarrow$   
 $heapmap\text{-rel} \rightarrow \langle heapmap\text{-rel} \rangle nres\text{-rel}$   
 $\langle proof \rangle$

## Increase-Key

**definition** *hm-increase-key-op*  $\equiv \lambda k v hm. do \{$   
 $ASSERT (heapmap\text{-invar } hm);$   
 $ASSERT (heapmap\text{-}\alpha hm k \neq None \wedge prio v \geq prio (the (heapmap\text{-}\alpha hm$   
 $k)))$ ;  
 $i \leftarrow hm\text{-index-op } hm k;$   
 $hm \leftarrow hm\text{-update-op } hm i v;$   
 $hm\text{-sink-op } hm i$   
 $\}$

**definition** (*in heapstruct*) *increase-key-op*  $i v h \equiv do \{$   
 $ASSERT (valid h i \wedge prio v \geq prio\text{-of } h i);$   
 $h \leftarrow update\text{-op } h i v;$   
 $sink\text{-op } h i$   
 $\}$

**lemma** (*in heapstruct*) *increase-key-op-invar*:  
 $\llbracket heap\text{-invar } h; valid h i; prio v \geq prio\text{-of } h i \rrbracket \implies increase\text{-key-op } i v h \leq$   
 $SPEC heap\text{-invar}$   
 $\langle proof \rangle$

**lemma** *hm-increase-key-op-refine*:  
 $\llbracket (hm, h) \in hmr\text{-rel}; (hm, m) \in heapmap\text{-rel}; m k = Some v \rrbracket$   
 $\implies hm\text{-increase-key-op } k v hm \leq \Downarrow hmr\text{-rel } (h.increase\text{-key-op } (hm\text{-index } hm$   
 $k) v h)$   
 $\langle proof \rangle$

**lemma** *hm-increase-key-op- $\alpha$ -correct*:  
 $heapmap\text{-invar } hm \implies hm\text{-increase-key-op } k v hm \leq_n SPEC (\lambda hm'. heapmap\text{-}\alpha$   
 $hm' = (heapmap\text{-}\alpha hm)(k \mapsto v))$   
 $\langle proof \rangle$

**lemma** *hm-increase-key-op-aref*:  
 $(hm\text{-increase-key-op}, PR\text{-CONST } (mop\text{-}pm\text{-increase-key } prio)) \in Id \rightarrow Id \rightarrow$   
 $heapmap\text{-rel} \rightarrow \langle heapmap\text{-rel} \rangle nres\text{-rel}$   
 $\langle proof \rangle$

## Change-Key

**definition**  $hm\text{-change-key-op} \equiv \lambda k v hm. \text{do} \{$   
 $ASSERT (heapmap\text{-invar } hm);$   
 $ASSERT (heapmap\text{-}\alpha \text{ } hm \text{ } k \neq None);$   
 $i \leftarrow hm\text{-index-op } hm \text{ } k;$   
 $hm \leftarrow hm\text{-update-op } hm \text{ } i \text{ } v;$   
 $hm\text{-repair-op } hm \text{ } i$   
 $\}$

**definition** (in  $heapstruct$ )  $change\text{-key-op } i \text{ } v \text{ } h \equiv \text{do} \{$   
 $ASSERT (valid \text{ } h \text{ } i);$   
 $h \leftarrow update\text{-op } h \text{ } i \text{ } v;$   
 $repair\text{-op } h \text{ } i$   
 $\}$

**lemma** (in  $heapstruct$ )  $change\text{-key-op-invar}$ :  
 $\llbracket heap\text{-invar } h; \text{ valid } h \text{ } i \rrbracket \implies change\text{-key-op } i \text{ } v \text{ } h \leq SPEC \text{ } heap\text{-invar}$   
 $\langle proof \rangle$

**lemma**  $hm\text{-change-key-op-refine}$ :  
 $\llbracket (hm, h) \in hmr\text{-rel}; (hm, m) \in heapmap\text{-rel}; m \text{ } k = Some \text{ } v \rrbracket$   
 $\implies hm\text{-change-key-op } k \text{ } v \text{ } hm \leq \Downarrow hmr\text{-rel } (h.change\text{-key-op } (hm\text{-index } hm \text{ } k))$   
 $v \text{ } h)$   
 $\langle proof \rangle$

**lemma**  $hm\text{-change-key-op-}\alpha\text{-correct}$ :  
 $heapmap\text{-invar } hm \implies hm\text{-change-key-op } k \text{ } v \text{ } hm \leq_n SPEC (\lambda hm'. heapmap\text{-}\alpha$   
 $hm' = (heapmap\text{-}\alpha \text{ } hm)(k \mapsto v))$   
 $\langle proof \rangle$

**lemma**  $hm\text{-change-key-op-aref}$ :  
 $(hm\text{-change-key-op}, mop\text{-map-update-ex}) \in Id \rightarrow Id \rightarrow heapmap\text{-rel} \rightarrow$   
 $\langle heapmap\text{-rel} \rangle nres\text{-rel}$   
 $\langle proof \rangle$

## Set

Realized as generic algorithm!

**lemma** (in  $-$ )  $op\text{-pm-set-gen-impl}$ :  $RETURN \text{ } ooo \text{ } op\text{-map-update} = (\lambda k v m.$   
 $\text{do} \{$   
 $c \leftarrow RETURN (op\text{-map-contains-key } k \text{ } m);$   
 $\text{if } c \text{ then}$   
 $mop\text{-map-update-ex } k \text{ } v \text{ } m$   
 $\text{else}$   
 $mop\text{-map-update-new } k \text{ } v \text{ } m$   
 $\}$   
 $\langle proof \rangle$

**definition**  $hm\text{-set-op } k \ v \ hm \equiv do \{$   
 $c \leftarrow hm\text{-contains-key-op } k \ hm;$   
 $if \ c \ then$   
 $hm\text{-change-key-op } k \ v \ hm$   
 $else$   
 $hm\text{-insert-op } k \ v \ hm$   
 $\}$

**lemma**  $hm\text{-set-op-aref}:$   
 $(hm\text{-set-op}, RETURN \ ooo \ op\text{-map-update}) \in Id \rightarrow Id \rightarrow heapmap\text{-rel} \rightarrow$   
 $\langle heapmap\text{-rel} \rangle nres\text{-rel}$   
 $\langle proof \rangle$

## Pop-Min

**definition**  $hm\text{-pop-min-op} :: ('k, 'v) \ ahm \Rightarrow (('k \times 'v) \times ('k, 'v) \ ahm) \ nres$  **where**  
 $hm\text{-pop-min-op } hm \equiv do \{$   
 $ASSERT \ (heapmap\text{-invar } hm);$   
 $ASSERT \ (hm\text{-valid } hm \ 1);$   
 $k \leftarrow hm\text{-key-of-op } hm \ 1;$   
 $v \leftarrow hm\text{-the-lookup-op } hm \ k;$   
 $let \ l = hm\text{-length } hm;$   
 $hm \leftarrow hm\text{-exch-op } hm \ 1 \ l;$   
 $hm \leftarrow hm\text{-butlast-op } hm;$   
 $if \ (l \neq 1) \ then \ do \ {$   
 $hm \leftarrow hm\text{-sink-op } hm \ 1;$   
 $RETURN \ ((k, v), hm)$   
 $\} \ else \ RETURN \ ((k, v), hm)$   
 $\}$

**lemma**  $hm\text{-pop-min-op-refine}:$   
 $(hm\text{-pop-min-op}, h.\text{pop-min-op}) \in hmr\text{-rel} \rightarrow \langle UNIV \times_r \ hmr\text{-rel} \rangle nres\text{-rel}$   
 $\langle proof \rangle$

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**  $heapmap\text{-min-prop}:$   
**assumes**  $INV: heapmap\text{-invar } hm$



**assumes**  $V'$ :  $\text{heapmap-}\alpha$   $hm$   $k = \text{Some } v'$   
**assumes**  $NE$ :  $hm\text{-valid } hm$  ( $\text{Suc } 0$ )  
**shows**  $\text{prio}$  (the ( $\text{heapmap-}\alpha$   $hm$  ( $hm\text{-key-of } hm$  ( $\text{Suc } 0$ ))))  $\leq \text{prio } v'$   
 $\langle \text{proof} \rangle$

With the above lemma, the correctness proof is straightforward

**lemma**  $hm\text{-pop-min-}\alpha\text{-correct}$ :  $hm\text{-pop-min-op } hm \leq_n \text{SPEC } (\lambda((k,v),hm').$   
 $\text{heapmap-}\alpha$   $hm$   $k = \text{Some } v$   
 $\wedge \text{heapmap-}\alpha$   $hm' = (\text{heapmap-}\alpha$   $hm)(k:=None)$   
 $\wedge (\forall k' v'. \text{heapmap-}\alpha$   $hm$   $k' = \text{Some } v' \longrightarrow \text{prio } v \leq \text{prio } v')$   
 $\langle \text{proof} \rangle$

**lemma**  $heapmap\text{-nres-rel-prodI}$ :  
**assumes**  $hmx \leq \Downarrow(\text{UNIV} \times_r \text{hmr-rel})$   $h'x$   
**assumes**  $h'x \leq \text{SPEC } (\lambda(-,h'). h.\text{heap-invar } h')$   
**assumes**  $hmx \leq_n \text{SPEC } (\lambda(r,hm'). \text{RETURN } (r,\text{heapmap-}\alpha$   $hm')) \leq \Downarrow(\text{R} \times_r \text{Id})$   
 $hx$   
**shows**  $hmx \leq \Downarrow(\text{R} \times_r \text{heapmap-rel})$   $hx$   
 $\langle \text{proof} \rangle$

**lemma**  $hm\text{-pop-min-op-aref}$ : ( $hm\text{-pop-min-op}$ ,  $\text{PR-CONST } (mop\text{-pm-pop-min}$   
 $\text{prio}) \in \text{heapmap-rel} \rightarrow \langle (\text{Id} \times_r \text{Id}) \times_r \text{heapmap-rel} \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**Second approach**      **definition**  $hm\text{-kv-of-op } hm$   $i \equiv \text{do } \{$   
 $\text{ASSERT } (hm\text{-valid } hm$   $i \wedge \text{hmr-invar } hm);$   
 $k \leftarrow hm\text{-key-of-op } hm$   $i;$   
 $v \leftarrow hm\text{-the-lookup-op } hm$   $k;$   
 $\text{RETURN } (k, v)$   
 $\}$

**definition**  $kvi\text{-rel } hm$   $i \equiv \{(k,v),v) \mid k v. hm\text{-key-of } hm$   $i = k\}$

**lemma**  $hm\text{-kv-op-refine}$ [ $\text{refine}$ ]:  
**assumes**  $(hm,h) \in \text{hmr-rel}$   
**shows**  $hm\text{-kv-of-op } hm$   $i \leq \Downarrow(kvi\text{-rel } hm$   $i)$  ( $h.\text{val-of-op } h$   $i$ )  
 $\langle \text{proof} \rangle$

**definition**  $hm\text{-pop-min-op}' :: ('k,'v)$   $ahm \Rightarrow ((k \times v) \times (k,'v)$   $ahm)$   $\text{nres}$  **where**  
 $hm\text{-pop-min-op}'$   $hm \equiv \text{do } \{$   
 $\text{ASSERT } (\text{heapmap-invar } hm);$   
 $\text{ASSERT } (hm\text{-valid } hm$   $1);$   
 $kv \leftarrow hm\text{-kv-of-op } hm$   $1;$   
 $\text{let } l = hm\text{-length } hm;$   
 $hm \leftarrow hm\text{-exch-op } hm$   $1$   $l;$   
 $hm \leftarrow hm\text{-butlast-op } hm;$

```

if (l≠1) then do {
  hm ← hm-sink-op hm 1;
  RETURN (kv,hm)
} else RETURN (kv,hm)
}

```

**lemma** *hm-pop-min-op-refine'*:

$\llbracket (hm,h) \in hmr\text{-rel} \rrbracket \implies hm\text{-pop-min-op}' hm \leq \Downarrow (kvi\text{-rel } hm \ 1 \times_r hmr\text{-rel})$   
*(h.pop-min-op h)*  
*<proof>*

**lemma** *heapmap-nres-rel-prodI'*:

**assumes**  $hmx \leq \Downarrow (S \times_r hmr\text{-rel}) h'x$   
**assumes**  $h'x \leq SPEC \Phi$   
**assumes**  $\bigwedge h' r. \Phi (r,h') \implies h.\text{heap-invar } h'$   
**assumes**  $hmx \leq_n SPEC (\lambda(r,hm'). (\exists r'. (r,r') \in S \wedge \Phi (r',hmr\text{-}\alpha hm')) \wedge hmr\text{-invar } hm' \longrightarrow RETURN (r,heapmap\text{-}\alpha hm') \leq \Downarrow (R \times_r Id) hx)$   
**shows**  $hmx \leq \Downarrow (R \times_r heapmap\text{-rel}) hx$   
*<proof>*

**lemma** *ex-in-kvi-rel-conv*:

$(\exists r'. (r,r') \in kvi\text{-rel } hm \ i \wedge \Phi r') \longleftrightarrow (fst\ r = hm\text{-key-of } hm \ i \wedge \Phi (snd\ r))$   
*<proof>*

**lemma** *hm-pop-min-aref'*:  $(hm\text{-pop-min-op}', mop\text{-pm-pop-min } prio) \in heapmap\text{-rel}$

$\rightarrow \langle (Id \times_r Id) \times_r heapmap\text{-rel} \rangle nres\text{-rel}$   
*<proof>*

## Remove

**definition** *hm-remove-op*  $k\ hm \equiv do \{$

```

  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*  $i\ h \equiv do \{$

```

  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]:  $swap\ l\ i\ j = [] \longleftrightarrow l=[]$   
 ⟨proof⟩

**lemma** (in *heapstruct*)  
*butlast-exch-last*:  $butlast\ (exch\ h\ i\ (length\ h)) = update\ (butlast\ h)\ i\ (last\ h)$   
 ⟨proof⟩

**lemma** (in *heapstruct*) *remove-op-invar*:  
 $\llbracket heap-invar\ h; valid\ h\ i \rrbracket \implies remove-op\ i\ h \leq SPEC\ heap-invar$   
 ⟨proof⟩

**lemma** *hm-remove-op-refine*[refine]:  
 $\llbracket (hm,m) \in heapmap-rel; (hm,h) \in hmr-rel; heapmap-\alpha\ hm\ k \neq None \rrbracket \implies$   
 $hm-remove-op\ k\ hm \leq \Downarrow hmr-rel\ (h.remove-op\ (hm-index\ hm\ k)\ h)$   
 ⟨proof⟩

**lemma** *hm-remove-op- $\alpha$ -correct*:  
 $hm-remove-op\ k\ hm \leq_n SPEC\ (\lambda hm'. heapmap-\alpha\ hm' = (heapmap-\alpha\ hm)(k:=None))$   
 ⟨proof⟩

**lemma** *hm-remove-op-aref*:  
 $(hm-remove-op, mop-map-delete-ex) \in Id \rightarrow heapmap-rel \rightarrow \langle heapmap-rel \rangle nres-rel$   
 ⟨proof⟩

## Peek-Min

**definition** *hm-peek-min-op* ::  $(k,v) ahm \Rightarrow (k \times v) nres$  **where**  
 $hm-peek-min-op\ hm \equiv hm-kv-of-op\ hm\ 1$

**lemma** *hm-peek-min-op-aref*:  
 $(hm-peek-min-op, PR-CONST\ (mop-pm-peek-min\ prio)) \in heapmap-rel \rightarrow$   
 $\langle Id \times_r Id \rangle nres-rel$   
 ⟨proof⟩

**end**

**end**

### 3.11 Plain Arrays Implementing List Interface

```

theory IICF-Array
imports ../Intf/IICF-List
begin

```

Lists of fixed length are directly implemented with arrays.

```

definition is-array  $l\ p \equiv p \rightarrow_a l$ 

```

```

lemma is-array-precise[safe-constraint-rules]: precise is-array
   $\langle proof \rangle$ 

```

```

definition array-assn where array-assn  $A \equiv hr\text{-comp}\ is\text{-array}\ (\langle the\text{-pure}\ A \rangle list\text{-rel})$ 
lemmas [safe-constraint-rules] = CN-FALSEI[of is-pure array-assn A for A]

```

```

definition [simp,code-unfold]: heap-array-empty  $\equiv Array.of\text{-list}\ []$ 

```

```

definition [simp,code-unfold]: heap-array-set  $p\ i\ v \equiv Array.upd\ i\ v\ p$ 

```

**context**

```

notes [fcomp-norm-unfold] = array-assn-def[symmetric]

```

```

notes [intro!] = hfrefI 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))  $\in unit\text{-assn}^k \rightarrow_a is\text{-array}$ 
   $\langle proof \rangle$ 

```

```

sepref-decl-impl (no-register) array-empty: array-empty-hnr-aux  $\langle proof \rangle$ 

```

```

lemma array-replicate-hnr-aux:

```

```

  (uncurry Array.new, uncurry (RETURN oo op-list-replicate))

```

```

   $\in nat\text{-assn}^k *_a id\text{-assn}^k \rightarrow_a is\text{-array}$ 

```

```

   $\langle proof \rangle$ 

```

```

sepref-decl-impl (no-register) array-replicate: array-replicate-hnr-aux  $\langle proof \rangle$ 

```

```

definition [simp]: op-array-replicate  $\equiv op\text{-list-replicate}$ 

```

```

sepref-register op-array-replicate

```

```

lemma array-fold-custom-replicate:

```

```

  replicate = op-array-replicate

```

```

  op-list-replicate = op-array-replicate

```

```

  mop-list-replicate = RETURN oo op-array-replicate

```

```

   $\langle proof \rangle$ 

```

```

lemmas array-replicate-custom-hnr[sepref-fr-rules] = array-replicate-hnr[unfolded
array-fold-custom-replicate]

```

```

lemma array-of-list-hnr-aux: (Array.of-list,RETURN o op-list-copy)  $\in (list\text{-assn}$ 
id-assn) $k$   $\rightarrow_a is\text{-array}$ 

```

$\langle proof \rangle$   
**sepref-decl-impl** (*no-register*) *array-of-list*: *array-of-list-hnr-aux*  $\langle proof \rangle$

**definition** [*simp*]: *op-array-of-list*  $\equiv$  *op-list-copy*  
**sepref-register** *op-array-of-list*  
**lemma** *array-fold-custom-of-list*:  
 $l = op\text{-}array\text{-}of\text{-}list\ l$   
 $op\text{-}list\text{-}copy = op\text{-}array\text{-}of\text{-}list$   
 $mop\text{-}list\text{-}copy = RETURN\ o\ op\text{-}array\text{-}of\text{-}list$   
 $\langle proof \rangle$

**lemmas** *array-of-list-custom-hnr*[*sepref-fr-rules*] = *array-of-list-hnr*[*folded op-array-of-list-def*]

**lemma** *array-copy-hnr-aux*: (*array-copy*, *RETURN o op-list-copy*)  $\in is\text{-}array^k$   
 $\rightarrow_a is\text{-}array$   
 $\langle proof \rangle$   
**sepref-decl-impl** *array-copy*: *array-copy-hnr-aux*  $\langle proof \rangle$

**lemma** *array-get-hnr-aux*: (*uncurry Array.nth*, *uncurry (RETURN oo op-list-get)*)  
 $\in [\lambda(l,i). i < length\ l]_a is\text{-}array^k *_a nat\text{-}assn^k \rightarrow id\text{-}assn$   
 $\langle proof \rangle$   
**sepref-decl-impl** *array-get*: *array-get-hnr-aux*  $\langle proof \rangle$

**lemma** *array-set-hnr-aux*: (*uncurry2 heap-array-set*, *uncurry2 (RETURN ooo op-list-set)*)  $\in [\lambda((l,i),-). i < length\ l]_a is\text{-}array^d *_a nat\text{-}assn^k *_a id\text{-}assn^k \rightarrow is\text{-}array$   
 $\langle proof \rangle$   
**sepref-decl-impl** *array-set*: *array-set-hnr-aux*  $\langle proof \rangle$

**lemma** *array-length-hnr-aux*: (*Array.len*, *RETURN o op-list-length*)  $\in is\text{-}array^k$   
 $\rightarrow_a nat\text{-}assn$   
 $\langle proof \rangle$   
**sepref-decl-impl** *array-length*: *array-length-hnr-aux*  $\langle proof \rangle$

**end**

**definition** [*simp*]: *op-array-empty*  $\equiv$  *op-list-empty*  
**interpretation** *array*: *list-custom-empty array-assn A heap-array-empty op-array-empty*  
 $\langle proof \rangle$

**end**  
**theory** *IICF-MS-Array-List*  
**imports**  
 $\dots / Intf / IICF\text{-}List$   
 $Separation\text{-}Logic\text{-}Imperative\text{-}HOL.Array\text{-}Blit$   
 $Separation\text{-}Logic\text{-}Imperative\text{-}HOL.Default\text{-}Insts$   
**begin**

**type-synonym**  $'a$  *ms-array-list* =  $'a$  *Heap.array*  $\times$  *nat*

**definition** *is-ms-array-list* *ms l*  $\equiv \lambda(a,n). \exists A l'. a \mapsto_a l' * \uparrow(n \leq \text{length } l' \wedge l = \text{take } n \ l' \wedge \text{ms=length } l')$

**lemma** *is-ms-array-list-prec*[*safe-constraint-rules*]: *precise* (*is-ms-array-list* *ms*)  
<*proof*>

**definition** *marl-empty-sz* *maxsize*  $\equiv$  *do* {  
   $a \leftarrow \text{Array.new } \text{maxsize } \text{default};$   
   $\text{return } (a,0)$   
}

**definition** *marl-append*  $\equiv \lambda(a,n) x. \text{do}$  {  
   $a \leftarrow \text{Array.upd } n \ x \ a;$   
   $\text{return } (a,n+1)$   
}

**definition** *marl-length* ::  $'a::\text{heap } \text{ms-array-list} \Rightarrow \text{nat } \text{Heap}$  **where**  
*marl-length*  $\equiv \lambda(a,n). \text{return } (n)$

**definition** *marl-is-empty* ::  $'a::\text{heap } \text{ms-array-list} \Rightarrow \text{bool } \text{Heap}$  **where**  
*marl-is-empty*  $\equiv \lambda(a,n). \text{return } (n=0)$

**definition** *marl-last* ::  $'a::\text{heap } \text{ms-array-list} \Rightarrow 'a \text{ Heap}$  **where**  
*marl-last*  $\equiv \lambda(a,n). \text{do}$  {  
   $\text{Array.nth } a \ (n - 1)$   
}

**definition** *marl-butlast* ::  $'a::\text{heap } \text{ms-array-list} \Rightarrow 'a \ \text{ms-array-list } \text{Heap}$  **where**  
*marl-butlast*  $\equiv \lambda(a,n). \text{do}$  {  
   $\text{return } (a,n - 1)$   
}

**definition** *marl-get* ::  $'a::\text{heap } \text{ms-array-list} \Rightarrow \text{nat} \Rightarrow 'a \ \text{Heap}$  **where**  
*marl-get*  $\equiv \lambda(a,n) i. \text{Array.nth } a \ i$

**definition** *marl-set* ::  $'a::\text{heap } \text{ms-array-list} \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow 'a \ \text{ms-array-list } \text{Heap}$   
**where**  
*marl-set*  $\equiv \lambda(a,n) i x. \text{do}$  {  $a \leftarrow \text{Array.upd } i \ x \ a;$   $\text{return } (a,n)$  }

**lemma** *marl-empty-sz-rule*[*sep-heap-rules*]:  $\langle \text{emp} \rangle \text{marl-empty-sz } N \langle \text{is-ms-array-list } N \ \square \rangle$   
<*proof*>

**lemma** *marl-append-rule*[*sep-heap-rules*]:  $\text{length } l < N \implies$   
 $\langle \text{is-ms-array-list } N \ l \ a \ \rangle$   
*marl-append*  $a \ x$

$\langle \lambda a. \text{is-ms-array-list } N (l@[x]) a \rangle_t$   
 $\langle \text{proof} \rangle$

**lemma** *marl-length-rule*[sep-heap-rules]:

$\langle \text{is-ms-array-list } N l a \rangle$   
*marl-length*  $a$   
 $\langle \lambda r. \text{is-ms-array-list } N l a * \uparrow(r=\text{length } l) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *marl-is-empty-rule*[sep-heap-rules]:

$\langle \text{is-ms-array-list } N l a \rangle$   
*marl-is-empty*  $a$   
 $\langle \lambda r. \text{is-ms-array-list } N l a * \uparrow(r \longleftrightarrow (l=[])) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *marl-last-rule*[sep-heap-rules]:

$l \neq [] \implies$   
 $\langle \text{is-ms-array-list } N l a \rangle$   
*marl-last*  $a$   
 $\langle \lambda r. \text{is-ms-array-list } N l a * \uparrow(r=\text{last } l) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *marl-butlast-rule*[sep-heap-rules]:

$l \neq [] \implies$   
 $\langle \text{is-ms-array-list } N l a \rangle$   
*marl-butlast*  $a$   
 $\langle \text{is-ms-array-list } N (\text{butlast } l) \rangle_t$   
 $\langle \text{proof} \rangle$

**lemma** *marl-get-rule*[sep-heap-rules]:

$i < \text{length } l \implies$   
 $\langle \text{is-ms-array-list } N l a \rangle$   
*marl-get*  $a i$   
 $\langle \lambda r. \text{is-ms-array-list } N l a * \uparrow(r=!!i) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *marl-set-rule*[sep-heap-rules]:

$i < \text{length } l \implies$   
 $\langle \text{is-ms-array-list } N l a \rangle$   
*marl-set*  $a i x$   
 $\langle \text{is-ms-array-list } N (l[i:=x]) \rangle$   
 $\langle \text{proof} \rangle$

**definition** *marl-assn*  $N A \equiv \text{hr-comp } (\text{is-ms-array-list } N) (\langle \text{the-pure } A \rangle \text{list-rel})$

**lemmas** [*safe-constraint-rules*] = *CN-FALSEI*[of *is-pure marl-assn*  $N A$  for  $N A$ ]

**context**

**notes** [*fcomp-norm-unfold*] = *marl-assn-def*[*symmetric*]

**notes** [intro!] = hfrefI 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**  
**sepref-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)  $\langle$ proof $\rangle$

**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  
 $\langle$ proof $\rangle$

**lemma** marl-empty-hnr-aux: (uncurry0 (marl-empty-sz N), uncurry0 (RETURN op-list-empty))  $\in$  unit-assn<sup>k</sup>  $\rightarrow_a$  is-ms-array-list N  
 $\langle$ proof $\rangle$

**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 mk-mop-rl0-np[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<sup>k</sup>  $\rightarrow_a$  marl-assn N A  
 $\langle$ proof $\rangle$

**lemma** marl-append-hnr-aux: (uncurry marl-append, uncurry (RETURN oo op-list-append))  $\in$  [ $\lambda(l,-). \text{length } l < N$ ]<sub>a</sub> ((is-ms-array-list N)<sup>d</sup> \*<sub>a</sub> id-assn<sup>k</sup>)  $\rightarrow$  is-ms-array-list N  
 $\langle$ proof $\rangle$

**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)  $\in$  (is-ms-array-list N)<sup>k</sup>  $\rightarrow_a$  nat-assn  
 $\langle$ proof $\rangle$

**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)  $\in$  (is-ms-array-list N)<sup>k</sup>  $\rightarrow_a$  bool-assn  
 $\langle$ proof $\rangle$

**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*) ∈ [ $\lambda x. x \neq []$ ]<sub>a</sub> (*is-ms-array-list*  
*N*)<sup>k</sup> → *id-assn*

*<proof>*

**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*) ∈ [ $\lambda x.$   
 $x \neq []$ ]<sub>a</sub> (*is-ms-array-list N*)<sup>d</sup> → (*is-ms-array-list N*)

*<proof>*

**lemmas** *marl-butlast-hnr*[*sepref-fr-rules*] = *marl-butlast-hnr-aux*[*FCOMP op-list-butlast.fref*]  
*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)*)  
∈ [ $\lambda(l,i). i < \text{length } l$ ]<sub>a</sub> ((*is-ms-array-list N*)<sup>k</sup> \*<sub>a</sub> *nat-assn*<sup>k</sup>) → *id-assn*

*<proof>*

**lemmas** *marl-get-hnr*[*sepref-fr-rules*] = *marl-get-hnr-aux*[*FCOMP op-list-get.fref*]

**lemmas** *marl-get-hnr-mop*[*sepref-fr-rules*] = *marl-get-hnr*[*FCOMP mk-mop-rl2* [*OF*  
*mop-list-get-alt*]]

**lemma** *marl-set-hnr-aux*: (*uncurry2 marl-set,uncurry2 (RETURN ooo op-list-set)*)  
∈ [ $\lambda((l,i),-). i < \text{length } l$ ]<sub>a</sub> ((*is-ms-array-list N*)<sup>d</sup> \*<sub>a</sub> *nat-assn*<sup>k</sup> \*<sub>a</sub> *id-assn*<sup>k</sup>) → (*is-ms-array-list*  
*N*)

*<proof>*

**lemmas** *marl-set-hnr*[*sepref-fr-rules*] = *marl-set-hnr-aux*[*FCOMP op-list-set.fref*]

**lemmas** *marl-set-hnr-mop*[*sepref-fr-rules*] = *marl-set-hnr*[*FCOMP mk-mop-rl3* [*OF*  
*mop-list-set-alt*]]

**end**

**context**

**fixes** *N* :: *nat*

**assumes** *N-sz*: *N > 10*

**begin**

**schematic-goal** *hn-refine (emp) (?c::?'c Heap) ?Γ' ?R (do {*

*let x = op-marl-empty-sz N;*

*RETURN (x@[1::nat])*

*})*

*<proof>*

**end**

```

schematic-goal hn-refine (emp) (?c::?'c Heap) ? $\Gamma$ ' ?R (do {
  let x = op-list-empty;
  RETURN (x@[1::nat])
})
  <proof>

```

```

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.

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 aial = nat list  $\times$  nat list

locale ial-invar = fixes
  maxsize :: nat
and l :: nat list
and qp :: nat list
assumes maxsize-eq[simp]: maxsize = length qp
assumes l-distinct[simp]: distinct l
assumes l-set: set l  $\subseteq$   $\{0..<length\ qp\}$ 
assumes qp-def:  $\forall k < length\ qp. qp!k = (if\ k \in set\ l\ then\ List-Index.index\ l\ k\ else\ length\ qp)$ 
begin
  lemma l-len: length l  $\leq$  length qp
    <proof>

  lemma idx-len[simp]: i < length l  $\implies$  l!i < length qp
    <proof>

  lemma l-set-simp[simp]: k  $\in$  set l  $\implies$  k < length qp
    <proof>

  lemma qpk-idx: k < length qp  $\implies$  qp ! k < length l  $\iff$  k  $\in$  set l
    <proof>

  lemma lqpk[simp]: k  $\in$  set l  $\implies$  l ! (qp ! k) = k

```

⟨proof⟩

**lemma**  $[[i < \text{length } l; j < \text{length } l; !i = !j]] \implies i = j$   
⟨proof⟩

**lemmas**  $\text{index-swap}[\text{simp}] = \text{index-swap-if-distinct}[\text{folded swap-def}, \text{OF } l\text{-distinct}]$

**lemma**  $\text{swap-invar}$ :

**assumes**  $i < \text{length } l \ j < \text{length } l$

**shows**  $\text{ial-invar } (\text{length } qp) (\text{swap } l \ i \ j) (qp[l \ ! \ j := i, l \ ! \ i := j])$

⟨proof⟩

**end**

**definition**  $\text{ial-rel1 } \text{maxsize} \equiv \text{br fst } (\text{uncurry } (\text{ial-invar } \text{maxsize}))$

**definition**  $\text{ial-assn2} :: \text{nat} \Rightarrow \text{nat list} * \text{nat list} \Rightarrow -$  **where**

$\text{ial-assn2 } \text{maxsize} \equiv \text{prod-assn } (\text{marl-assn } \text{maxsize } \text{nat-assn}) (\text{array-assn } \text{nat-assn})$

**definition**  $\text{ial-assn } \text{maxsize } A \equiv \text{hr-comp } (\text{hr-comp } (\text{ial-assn2 } \text{maxsize}) (\text{ial-rel1 } \text{maxsize})) (\langle \text{the-pure } A \rangle \text{list-rel})$

**lemmas**  $[\text{safe-constraint-rules}] = \text{CN-FALSEI}[\text{of is-pure } \text{ial-assn } \text{maxsize } A \text{ for } \text{maxsize } A]$

### 3.11.1 Empty

**definition**  $\text{op-ial-empty-sz} :: \text{nat} \Rightarrow 'a \text{ list}$

**where**  $[\text{simp}]: \text{op-ial-empty-sz } ms \equiv \text{op-list-empty}$

**lemma**  $[\text{def-pat-rules}]: \text{op-ial-empty-sz } \$ \text{maxsize} \equiv \text{UNPROTECT } (\text{op-ial-empty-sz } \text{maxsize})$

⟨proof⟩

**context** **fixes**  $\text{maxsize} :: \text{nat}$  **begin**

**sepref-register**  $\text{PR-CONST } (\text{op-ial-empty-sz } \text{maxsize})$

**end**

**context**

**fixes**  $\text{maxsize} :: \text{nat}$

**notes**  $[\text{fcomp-norm-unfold}] = \text{ial-assn-def}[\text{symmetric}]$

**notes**  $[\text{simp}] = \text{hn-ctxt-def pure-def}$

**begin**

**definition**  $\text{aial-empty} \equiv \text{do } \{$

$\text{let } l = \text{op-marl-empty-sz } \text{maxsize};$

$\text{let } qp = \text{op-array-replicate } \text{maxsize } \text{maxsize};$

```

  RETURN (l,qp)
}

```

**lemma** *aial-empty-impl*:  $(aial\text{-}empty, RETURN\ op\text{-}list\text{-}empty) \in \langle ial\text{-}rel1\ maxsize \rangle nres\text{-}rel$   
 $\langle proof \rangle$

**context**

**notes** [*ial-rules*] = *itypeI*[*Pure.of maxsize TYPE(nat)*]

**notes** [*sepref-import-param*] = *IdI*[*of maxsize*]

**begin**

**sepref-definition** *ial-empty is uncurry0 aial-empty :: unit-assn<sup>k</sup> →<sub>a</sub> ial-assn2 maxsize*

$\langle proof \rangle$

**end**

**sepref-decl-impl** (*no-register*) *ial-empty: ial-empty.refine[FCOMP aial-empty-impl]*  
 $\langle proof \rangle$

**lemma** *ial-empty-sz-hnr*[*sepref-fr-rules*]:

$(uncurry0\ local.\ ial\text{-}empty, uncurry0\ (RETURN\ (PR\text{-}CONST\ (op\text{-}ial\text{-}empty\text{-}sz\ maxsize)))) \in unit\text{-}assn^k \rightarrow_a ial\text{-}assn\ maxsize\ A$

$\langle proof \rangle$

### 3.11.2 Swap

**definition** *aial-swap*  $\equiv \lambda(l,qp)\ i\ j.\ do\ \{$

$vi \leftarrow mop\text{-}list\text{-}get\ l\ i;$

$vj \leftarrow mop\text{-}list\text{-}get\ l\ j;$

$l \leftarrow mop\text{-}list\text{-}set\ l\ i\ vj;$

$l \leftarrow mop\text{-}list\text{-}set\ l\ j\ vi;$

$qp \leftarrow mop\text{-}list\text{-}set\ qp\ vj\ i;$

$qp \leftarrow mop\text{-}list\text{-}set\ qp\ vi\ j;$

$RETURN\ (l,qp)$

$\}$

**lemma** *in-ial-rel1-conv*:

$((pq, qp), l) \in ial\text{-}rel1\ ms \longleftrightarrow pq=l \wedge ial\text{-}invar\ ms\ l\ qp$

$\langle proof \rangle$

**lemma** *aial-swap-impl*:

$(aial\text{-}swap, mop\text{-}list\text{-}swap) \in ial\text{-}rel1\ maxsize \rightarrow nat\text{-}rel \rightarrow nat\text{-}rel \rightarrow \langle ial\text{-}rel1\ maxsize \rangle nres\text{-}rel$

$\langle proof \rangle$

**sepref-definition** *ial-swap is*

$uncurry2\ aial\text{-}swap :: (ial\text{-}assn2\ maxsize)^d *_a nat\text{-}assn^k *_a nat\text{-}assn^k \rightarrow_a ial\text{-}assn2\ maxsize$

$\langle proof \rangle$

**sempref-decl-impl** (*ismop*) *test*: *ial-swap.refine*[*FCOMP ial-swap-impl*]  
**uses** *mop-list-swap.fref*  $\langle$ *proof* $\rangle$

### 3.11.3 Length

**definition** *aial-length* :: *aial*  $\Rightarrow$  *nat nres*  
**where** *aial-length*  $\equiv \lambda(l,-).$  *RETURN* (*op-list-length* *l*)

**lemma** *aial-length-impl*: (*aial-length*, *mop-list-length*)  $\in$  *ial-rel1 maxsize*  $\rightarrow$   
 $\langle$ *nat-rel* $\rangle$ *nres-rel*  
 $\langle$ *proof* $\rangle$

**sempref-definition** *ial-length* **is** *aial-length* :: (*ial-assn2 maxsize*)<sup>*k*</sup>  $\rightarrow_a$  *nat-assn*  
 $\langle$ *proof* $\rangle$

**sempref-decl-impl** (*ismop*) *ial-length*: *ial-length.refine*[*FCOMP aial-length-impl*]  
 $\langle$ *proof* $\rangle$

### 3.11.4 Index

**definition** *aial-index* :: *aial*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat nres* **where**  
*aial-index*  $\equiv \lambda(l,qp).$  *do* {  
  *ASSERT* (*k*  $\in$  *set* *l*);  
  *i*  $\leftarrow$  *mop-list-get* *qp* *k*;  
  *RETURN* *i*  
}

**lemma** *aial-index-impl*:  
(*uncurry aial-index*, *uncurry mop-list-index*)  $\in$   
 $[\lambda(l,k).$  *k*  $\in$  *set* *l*]<sub>*f*</sub> *ial-rel1 maxsize*  $\times_r$  *nat-rel*  $\rightarrow$   $\langle$ *nat-rel* $\rangle$ *nres-rel*  
 $\langle$ *proof* $\rangle$

**sempref-definition** *ial-index* **is** *uncurry aial-index* :: (*ial-assn2 maxsize*)<sup>*k*</sup>  $\ast_a$   
*nat-assn*<sup>*k*</sup>  $\rightarrow_a$  *nat-assn*  
 $\langle$ *proof* $\rangle$

**sempref-decl-impl** (*ismop*) *ial-index*: *ial-index.refine*[*FCOMP aial-index-impl*]  
 $\langle$ *proof* $\rangle$

### 3.11.5 Butlast

**definition** *aial-butlast* :: *aial*  $\Rightarrow$  *aial nres* **where**  
*aial-butlast*  $\equiv \lambda(l,qp).$  *do* {  
  *ASSERT* (*l*  $\neq$  []);  
  *len*  $\leftarrow$  *mop-list-length* *l*;  
  *k*  $\leftarrow$  *mop-list-get* *l* (*len* - 1);  
  *l*  $\leftarrow$  *mop-list-butlast* *l*;  
  *qp*  $\leftarrow$  *mop-list-set* *qp* *k* (*length qp*);  
  *RETURN* (*l,qp*)  
}

}

**lemma** *aial-butlast-refine*:  $(aial\text{-}butlast, mop\text{-}list\text{-}butlast) \in ial\text{-}rel1\ maxsize \rightarrow \langle ial\text{-}rel1\ maxsize \rangle nres\text{-}rel$   
 ⟨proof⟩

**sepref-definition** *aial-butlast is aial-butlast* ::  $(ial\text{-}assn2\ maxsize)^d \rightarrow_a ial\text{-}assn2\ maxsize$   
 ⟨proof⟩

**sepref-decl-impl** (*ismop*) *aial-butlast*: *aial-butlast.refine*[*FCOMP aial-butlast-refine*]  
 ⟨proof⟩

### 3.11.6 Append

**definition** *aial-append* :: *aial*  $\Rightarrow$  *nat*  $\Rightarrow$  *aial nres* **where**

*aial-append*  $\equiv \lambda(l, qp)\ k.\ do \{$   
 ASSERT ( $k < length\ qp \wedge k \notin set\ l \wedge length\ l < length\ qp$ );  
 len  $\leftarrow mop\text{-}list\text{-}length\ l$ ;  
 l  $\leftarrow mop\text{-}list\text{-}append\ l\ k$ ;  
 qp  $\leftarrow mop\text{-}list\text{-}set\ qp\ k\ len$ ;  
 RETURN ( $l, qp$ )  
 }

**lemma** *aial-append-refine*:

$(uncurry\ aial\text{-}append, uncurry\ mop\text{-}list\text{-}append) \in$   
 $[\lambda(l, k). k < maxsize \wedge k \notin set\ l]_f\ ial\text{-}rel1\ maxsize \times_r\ nat\text{-}rel \rightarrow \langle ial\text{-}rel1\ maxsize \rangle nres\text{-}rel$   
 ⟨proof⟩ **lemma** *aial-append-impl-aux*:  $((l, qp), l') \in ial\text{-}rel1\ maxsize \implies l' = l$   
 $\wedge maxsize = length\ qp$   
 ⟨proof⟩

**context**

**notes** [*dest!*] = *aial-append-impl-aux*

**begin**

**sepref-definition** *aial-append is*

$uncurry\ aial\text{-}append :: [\lambda(lqp, -). lqp \in Domain\ (ial\text{-}rel1\ maxsize)]_a\ (ial\text{-}assn2\ maxsize)^d *_{\alpha}\ nat\text{-}assn^k \rightarrow ial\text{-}assn2\ maxsize$   
 ⟨proof⟩

**end**

**lemma**  $(\lambda b. b < maxsize, X) \in A \rightarrow bool\text{-}rel$

⟨proof⟩

**context begin**

**private lemma** *append-fref'*:  $[[IS\text{-}BELOW\text{-}ID\ R]]$

$\implies (\text{uncurry mop-list-append}, \text{uncurry mop-list-append}) \in \langle R \rangle \text{list-rel} \times_r R$   
 $\rightarrow_f \langle \langle R \rangle \text{list-rel} \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**sepref-decl-impl** (*ismop*) *ial-append*: *ial-append.refine*[FCOMP *aial-append.refine*]  
**uses** *append-fref'*  
 $\langle \text{proof} \rangle$   
**end**

### 3.11.7 Get

**definition** *aial-get* :: *aial*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat nres* **where**  
*aial-get*  $\equiv \lambda(l, qp) i. \text{mop-list-get } l \ i$

**lemma** *aial-get.refine*: (*aial-get*, *mop-list-get*)  $\in$  *ial-rel1 maxsize*  $\rightarrow$  *nat-rel*  $\rightarrow$   
 $\langle \text{nat-rel} \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**sepref-definition** *ial-get* **is** *uncurry aial-get* :: (*ial-assn2 maxsize*)<sup>k</sup> \*<sub>a</sub> *nat-assn*<sup>k</sup>  
 $\rightarrow_a$  *nat-assn*  
 $\langle \text{proof} \rangle$

**sepref-decl-impl** (*ismop*) *ial-get*: *ial-get.refine*[FCOMP *aial-get.refine*]  $\langle \text{proof} \rangle$

### 3.11.8 Contains

**definition** *aial-contains* :: *nat*  $\Rightarrow$  *aial*  $\Rightarrow$  *bool nres* **where**  
*aial-contains*  $\equiv \lambda k (l, qp). \text{do } \{$   
 $\quad \text{if } k < \text{maxsize} \text{ then do } \{$   
 $\quad \quad i \leftarrow \text{mop-list-get } qp \ k;$   
 $\quad \quad \text{RETURN } (i < \text{maxsize})$   
 $\quad \quad \} \text{ else RETURN False}$   
 $\quad \}$

**lemma** *aial-contains.refine*: (*uncurry aial-contains*, *uncurry mop-list-contains*)  
 $\in$  (*nat-rel*  $\times_r$  *ial-rel1 maxsize*)  $\rightarrow_f$   $\langle \text{bool-rel} \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**context**

**notes** [*id-rules*] = *itypeI*[*Pure.of maxsize TYPE(nat)*]

**notes** [*sepref-import-param*] = *IdI*[*of maxsize*]

**begin**

**sepref-definition** *ial-contains* **is** *uncurry aial-contains* :: *nat-assn*<sup>k</sup> \*<sub>a</sub> (*ial-assn2*  
*maxsize*)<sup>k</sup>  $\rightarrow_a$  *bool-assn*  
 $\langle \text{proof} \rangle$

**end**

**sepref-decl-impl** (*ismop*) *ial-contains*: *ial-contains.refine*[FCOMP *aial-contains.refine*]  
 $\langle \text{proof} \rangle$   
**end**

```

interpretation ial-sz: list-custom-empty ial-assn N A ial-empty N PR-CONST
(op-ial-empty-sz N)
  <proof>

```

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
  <proof>

```

```

type-synonym 'v hma = nat list × ('v list)
sepref-decl-intf 'v i-hma is nat list × (nat → 'v)

```

```

locale hmstruct-impl = hmstruct prio for prio :: 'v::heap ⇒ 'p::linorder
begin
  lemma param-prio: (prio,prio) ∈ Id → Id <proof>
  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 <proof>
interpretation hmstruct-impl <proof>

```

```

definition hm-impl1-α ≡ λ(pq,ml).
  (pq,λk. if k∈set pq then Some (ml!k) else None)

```



**definition**  $hm-impl1-invar \equiv \lambda(pq,ml).$   
 $hmr-invar (hm-impl1-\alpha (pq,ml))$   
 $\wedge set\ pq \subseteq \{0..<maxsize\}$   
 $\wedge ((pq=[] \wedge ml=[])) \vee (length\ ml = maxsize))$

**definition**  $hm-impl1-weak-invar \equiv \lambda(pq,ml).$   
 $set\ pq \subseteq \{0..<maxsize\}$   
 $\wedge ((pq=[] \wedge ml=[])) \vee (length\ ml = maxsize))$

**definition**  $hm-impl1-rel \equiv br\ hm-impl1-\alpha\ hm-impl1-invar$

**definition**  $hm-weak-impl'-rel \equiv br\ hm-impl1-\alpha\ hm-impl1-weak-invar$

**lemmas**  $hm-impl1-rel-defs =$   
 $hm-impl1-rel-def\ hm-weak-impl'-rel-def\ hm-impl1-weak-invar-def\ hm-impl1-invar-def$   
 $hm-impl1-\alpha-def\ in-br-conv$

**lemma**  $hm-impl-\alpha-fst-eq:$   
 $(x1, x2) = hm-impl1-\alpha (x1a, x2a) \implies x1 = x1a$   
 $\langle proof \rangle$

**term**  $hm-empty-op$

**definition**  $hm-empty-op' :: 'v\ hma\ nres$   
**where**  $hm-empty-op' \equiv do \{$   
 $let\ pq = op-ial-empty-sz\ maxsize;$   
 $let\ ml = op-list-empty;$   
 $RETURN (pq,ml)$   
 $\}$

**lemma**  $hm-empty-op'-refine: (hm-empty-op', hm-empty-op) \in (hm-impl1-rel)\ nres-rel$

$\langle proof \rangle$

**definition**  $hm-length' :: 'v\ hma \Rightarrow nat$  **where**  $hm-length' \equiv \lambda(pq,ml). length$   
 $pq$

**lemma**  $hm-length'-refine: (RETURN\ o\ hm-length', RETURN\ o\ hm-length) \in$   
 $hm-impl1-rel \rightarrow (nat-rel)\ nres-rel$   
 $\langle proof \rangle$

**term**  $hm-key-of-op$

**definition**  $hm-key-of-op' \equiv \lambda(pq,ml)\ i. ASSERT\ (i > 0) \gg mop-list-get\ pq\ (i$   
 $- 1)$

**lemma**  $hm-key-of-op'-refine: (hm-key-of-op', hm-key-of-op) \in hm-impl1-rel \rightarrow$   
 $nat-rel \rightarrow (nat-rel)\ nres-rel$

*<proof>*

**term** *hm-lookup*

**definition** *hm-lookup-op'*  $\equiv \lambda(pq,ml) k. do \{$   
  *if* (*k* < *maxsize*) *then do* { — TODO: This check can be eliminated, but this  
  will complicate refinement of keys in basic ops  
    *let* *c* = *op-list-contains k pq*;  
    *if* *c* *then do* {  
      *v*  $\leftarrow$  *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)*)

$\in (hm-impl1-rel \times_r nat-rel) \rightarrow_f \langle \langle Id \rangle option-rel \rangle nres-rel$   
*<proof>*

**term** *hm-contains-key-op*

**definition** *hm-contains-key-op'*  $\equiv \lambda k (pq,ml). do \{$   
  *if* (*k* < *maxsize*) *then do* { — TODO: This check can be eliminated, but this  
  will complicate refinement of keys in basic ops  
    *RETURN (op-list-contains k pq)*  
  } *else RETURN False*  
}

**lemma** *hm-contains-key-op'-refine*: (*uncurry hm-contains-key-op'*, *uncurry hm-contains-key-op*)

$\in (nat-rel \times_r hm-impl1-rel) \rightarrow_f \langle bool-rel \rangle nres-rel$   
*<proof>*

**term** *hm-valid*

**definition** *hm-exch-op'*  $\equiv \lambda(pq,ml) i j. do \{$   
  *ASSERT (hm-valid (hm-impl1- $\alpha$  (pq,ml)) i)*;  
  *ASSERT (hm-valid (hm-impl1- $\alpha$  (pq,ml)) j)*;  
  *pq*  $\leftarrow$  *mop-list-swap pq (i - 1) (j - 1)*;  
  *RETURN (pq,ml)*  
}

**lemma** *hm-impl1-relI*:

**assumes** *hmr-invar b*  
**assumes** (*a,b*)  $\in$  *hm-weak-impl'-rel*  
**shows** (*a,b*)  $\in$  *hm-impl1-rel*  
*<proof>*

**lemma** *hm-impl1-nres-relI*:

**assumes**  $b \leq_n \text{SPEC } \text{hmr-invar}$   
**assumes**  $(a,b) \in \langle \text{hm-weak-impl}'\text{-rel} \rangle \text{nres-rel}$   
**shows**  $(a,b) \in \langle \text{hm-impl1-rel} \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{hm-exch-op}'\text{-refine}$ :  $(\text{hm-exch-op}', \text{hm-exch-op}) \in \text{hm-impl1-rel} \rightarrow \text{nat-rel}$   
 $\rightarrow \text{nat-rel} \rightarrow \langle \text{hm-impl1-rel} \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**term**  $\text{hm-index-op}$

**definition**  $\text{hm-index-op}' \equiv \lambda(pq,ml) k.$

$\text{do } \{$   
 $\text{ASSERT } (\text{hm-impl1-invar } (pq,ml) \wedge \text{heapmap-}\alpha (\text{hm-impl1-}\alpha (pq,ml))) k \neq$   
 $\text{None} \wedge k \in \text{set } pq);$   
 $i \leftarrow \text{mop-list-index } pq \ k;$   
 $\text{RETURN } (i+1)$   
 $\}$

**lemma**  $\text{hm-index-op}'\text{-refine}$ :  $(\text{hm-index-op}', \text{hm-index-op})$   
 $\in \text{hm-impl1-rel} \rightarrow \text{nat-rel} \rightarrow \langle \text{nat-rel} \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**definition**  $\text{hm-update-op}'$  **where**

$\text{hm-update-op}' \equiv \lambda(pq,ml) i v. \text{do } \{$   
 $\text{ASSERT } (\text{hm-valid } (\text{hm-impl1-}\alpha (pq,ml)) i \wedge \text{hm-impl1-invar } (pq,ml));$   
 $k \leftarrow \text{mop-list-get } pq \ (i - 1);$   
 $ml \leftarrow \text{mop-list-set } ml \ k \ v;$   
 $\text{RETURN } (pq, ml)$   
 $\}$

**lemma**  $\text{hm-update-op}'\text{-refine}$ :  $(\text{hm-update-op}', \text{hm-update-op}) \in \text{hm-impl1-rel} \rightarrow$   
 $\text{nat-rel} \rightarrow \text{Id} \rightarrow \langle \text{hm-impl1-rel} \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**term**  $\text{hm-butlast-op}$

**lemma**  $\text{hm-butlast-op-invar}$ :  $\text{hm-butlast-op } \text{hm} \leq_n \text{SPEC } \text{hmr-invar}$   
 $\langle \text{proof} \rangle$

**definition**  $\text{hm-butlast-op}'$  **where**

$\text{hm-butlast-op}' \equiv \lambda(pq,ml). \text{do } \{$   
 $\text{ASSERT } (\text{hmr-invar } (\text{hm-impl1-}\alpha (pq,ml)));$   
 $pq \leftarrow \text{mop-list-butlast } pq;$   
 $\text{RETURN } (pq,ml)$   
 $\}$

**lemma** *set-butlast-distinct-conv*:

$\llbracket \text{distinct } l \rrbracket \implies \text{set } (\text{butlast } l) = \text{set } l - \{\text{last } l\}$   
 $\langle \text{proof} \rangle$

**lemma** *hm-butlast-op'-refine*:  $(\text{hm-butlast-op}', \text{hm-butlast-op}) \in \text{hm-impl1-rel}$   
 $\rightarrow \langle \text{hm-impl1-rel} \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**definition** *hm-append-op'*

**where**  $\text{hm-append-op}' \equiv \lambda(pq, ml) k v. \text{do } \{$   
 $\text{ASSERT } (k \notin \text{set } pq \wedge k < \text{maxsize});$   
 $\text{ASSERT } (\text{hm-impl1-invar } (pq, ml));$   
 $pq \leftarrow \text{mop-list-append } pq k;$   
 $ml \leftarrow (\text{if length } ml = 0 \text{ then mop-list-rotate maxsize } v \text{ else RETURN } ml);$   
 $ml \leftarrow \text{mop-list-set } ml k v;$   
 $\text{RETURN } (pq, ml)$   
 $\}$

**lemma** *hm-append-op'-refine*:  $(\text{uncurry2 } \text{hm-append-op}', \text{uncurry2 } \text{hm-append-op})$

$\in [\lambda((hm, k), v). k < \text{maxsize}]_f (\text{hm-impl1-rel} \times_r \text{nat-rel}) \times_r \text{Id} \rightarrow \langle \text{hm-impl1-rel} \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**definition** *hm-impl2-rel*  $\equiv \text{prod-assn } (\text{ial-assn maxsize id-assn}) (\text{array-assn id-assn})$

**definition** *hm-impl-rel*  $\equiv \text{hr-comp } \text{hm-impl2-rel } \text{hm-impl1-rel}$

**lemmas**  $[\text{fcomp-norm-unfold}] = \text{hm-impl-rel-def}[\text{symmetric}]$

### 3.12.1 Implement Basic Operations

**lemma** *param-parent*:  $(\text{efficient-nat-div2}, h.\text{parent}) \in \text{Id} \rightarrow \text{Id}$   
 $\langle \text{proof} \rangle$

**lemmas**  $[\text{sepref-import-param}] = \text{param-parent}$   
**sepref-register**  $h.\text{parent}$

**lemma** *param-left*:  $(h.\text{left}, h.\text{left}) \in \text{Id} \rightarrow \text{Id} \langle \text{proof} \rangle$

**lemmas**  $[\text{sepref-import-param}] = \text{param-left}$   
**sepref-register**  $h.\text{left}$

**lemma** *param-right*:  $(h.\text{right}, h.\text{right}) \in \text{Id} \rightarrow \text{Id} \langle \text{proof} \rangle$

**lemmas**  $[\text{sepref-import-param}] = \text{param-right}$   
**sepref-register**  $h.\text{right}$

**abbreviation**  $(\text{input}) \text{prio-rel} \equiv (\text{Id}::('p \times 'p) \text{set})$

**lemma** *param-prio-le*:  $((\leq), (\leq)) \in \text{prio-rel} \rightarrow \text{prio-rel} \rightarrow \text{bool-rel} \langle \text{proof} \rangle$

**lemmas**  $[\text{sepref-import-param}] = \text{param-prio-le}$

**lemma** *param-prio-lt*:  $((<), (<)) \in \text{prio-rel} \rightarrow \text{prio-rel} \rightarrow \text{bool-rel} \langle \text{proof} \rangle$

**lemmas** [*sepref-import-param*] = *param-prio-lt*

**abbreviation** *I-HM-UNF*  $\equiv \text{TYPE}(\text{nat list} \times 'v \text{ list})$

**sepref-definition** *hm-length-impl* **is** *RETURN* *o hm-length'* ::  $\text{hm-impl2-rel}^k \rightarrow_a \text{nat-assn}$   
 $\langle \text{proof} \rangle$

**lemmas** [*sepref-fr-rules*] = *hm-length-impl.refine[FCOMP hm-length'-refine]*

**sepref-register** *hm-length*::(*nat, 'v*) *ahm*  $\Rightarrow$  -

**sepref-definition** *hm-key-of-op-impl* **is** *uncurry hm-key-of-op'* ::  $\text{hm-impl2-rel}^k *_a \text{nat-assn}^k$   
 $\rightarrow_a \text{nat-assn}$   
 $\langle \text{proof} \rangle$

**lemmas** [*sepref-fr-rules*] = *hm-key-of-op-impl.refine[FCOMP hm-key-of-op'-refine]*

**sepref-register** *hm-key-of-op*::(*nat, 'v*) *ahm*  $\Rightarrow$  -

**context**

**notes** [*id-rules*] = *itypeI[Pure.of maxsize TYPE(nat)]*

**notes** [*sepref-import-param*] = *IdI[of maxsize]*

**begin**

**sepref-definition** *hm-lookup-impl* **is** *uncurry hm-lookup-op'* ::  $(\text{hm-impl2-rel}^k *_a \text{nat-assn}^k)$   
 $\rightarrow_a \text{option-assn id-assn}$   
 $\langle \text{proof} \rangle$

**lemmas** [*sepref-fr-rules*] =

*hm-lookup-impl.refine[FCOMP hm-lookup-op'-refine]*

**sepref-register** *hm-lookup*::(*nat, 'v*) *ahm*  $\Rightarrow$  -

**sepref-definition** *hm-exch-op-impl* **is** *uncurry2 hm-exch-op'* ::  $\text{hm-impl2-rel}^d *_a \text{nat-assn}^k *_a \text{nat-assn}^k$   
 $\rightarrow_a \text{hm-impl2-rel}$   
 $\langle \text{proof} \rangle$

**lemmas** [*sepref-fr-rules*] = *hm-exch-op-impl.refine[FCOMP hm-exch-op'-refine]*

**sepref-register** *hm-exch-op*::(*nat, 'v*) *ahm*  $\Rightarrow$  -

**sepref-definition** *hm-index-op-impl* **is** *uncurry hm-index-op'* ::  $\text{hm-impl2-rel}^k *_a \text{id-assn}^k$   
 $\rightarrow_a \text{id-assn}$   
 $\langle \text{proof} \rangle$

**lemmas** [*sepref-fr-rules*] = *hm-index-op-impl.refine[FCOMP hm-index-op'-refine]*

**sepref-register** *hm-index-op*::(*nat, 'v*) *ahm*  $\Rightarrow$  -

**sepref-definition** *hm-update-op-impl* **is** *uncurry2 hm-update-op'* ::  $\text{hm-impl2-rel}^d *_a \text{id-assn}^k *_a \text{id-assn}^k$   
 $\rightarrow_a \text{hm-impl2-rel}$   
 $\langle \text{proof} \rangle$

**lemmas** [*sepref-fr-rules*] = *hm-update-op-impl.refine[FCOMP hm-update-op'-refine]*

**sepref-register** *hm-update-op*::(*nat, 'v*) *ahm*  $\Rightarrow$  -

**sepref-definition** *hm-butlast-op-impl* is *hm-butlast-op'* :: *hm-impl2-rel*<sup>d</sup> →<sub>a</sub> *hm-impl2-rel*  
 ⟨proof⟩  
**lemmas** [*sepref-fr-rules*] = *hm-butlast-op-impl.refine*[*FCOMP hm-butlast-op'-refine*]  
**sepref-register** *hm-butlast-op*::(*nat, 'v*) *ahm* ⇒ -

**sepref-definition** *hm-append-op-impl* is *uncurry2 hm-append-op'* :: *hm-impl2-rel*<sup>d</sup>  
 \*<sub>a</sub> *id-assn*<sup>k</sup> \*<sub>a</sub> *id-assn*<sup>k</sup> →<sub>a</sub> *hm-impl2-rel*  
 ⟨proof⟩  
**lemmas** [*sepref-fr-rules*] = *hm-append-op-impl.refine*[*FCOMP hm-append-op'-refine*]  
**sepref-register** *hm-append-op*::(*nat, 'v*) *ahm* ⇒ -

### 3.12.2 Auxiliary Operations

**lemmas** [*intf-of-assn*] = *intf-of-assnI*[**where** *R=hm-impl-rel* :: (*nat, 'v*) *ahm* ⇒  
 - **and** *'a='v i-hma*]

**sepref-definition** *hm-valid-impl* is *uncurry (RETURN oo hm-valid)* :: *hm-impl-rel*<sup>k</sup> \*<sub>a</sub> *nat-assn*<sup>k</sup>  
 →<sub>a</sub> *bool-assn*  
 ⟨proof⟩  
**lemmas** [*sepref-fr-rules*] = *hm-valid-impl.refine*  
**sepref-register** *hm-valid*::(*nat, 'v*) *ahm* ⇒ -

**definition** *hm-the-lookup-op'* *hm k* ≡ *do* {  
*let* (*pq, ml*) = *hm*;  
*ASSERT* (*heapmap-α (hm-impl1-α hm) k ≠ None* ∧ *hm-impl1-invar hm*);  
*v* ← *mop-list-get ml k*;  
*RETURN v*  
 }

**lemma** *hm-the-lookup-op'-refine*:  
 (*hm-the-lookup-op', hm-the-lookup-op*) ∈ *hm-impl1-rel* → *nat-rel* → ⟨*Id*⟩ *nres-rel*  
 ⟨proof⟩

**sepref-definition** *hm-the-lookup-op-impl* is *uncurry hm-the-lookup-op'* :: *hm-impl2-rel*<sup>k</sup> \*<sub>a</sub> *id-assn*<sup>k</sup>  
 →<sub>a</sub> *id-assn*  
 ⟨proof⟩  
**lemmas** *hm-the-lookup-op-impl*[*sepref-fr-rules*] = *hm-the-lookup-op-impl.refine*[*FCOMP*  
*hm-the-lookup-op'-refine*]  
**sepref-register** *hm-the-lookup-op*::(*nat, 'v*) *ahm* ⇒ -

**sepref-definition** *hm-val-of-op-impl* is *uncurry hm-val-of-op* :: *hm-impl-rel*<sup>k</sup> \*<sub>a</sub> *id-assn*<sup>k</sup>  
 →<sub>a</sub> *id-assn*  
 ⟨proof⟩  
**lemmas** [*sepref-fr-rules*] = *hm-val-of-op-impl.refine*  
**sepref-register** *hm-val-of-op*::(*nat, 'v*) *ahm* ⇒ -

**sepref-definition** *hm-prio-of-op-impl* is *uncurry (PR-CONST hm-prio-of-op)*

$:: hm\text{-impl-rel}^k *_a id\text{-assn}^k \rightarrow_a id\text{-assn}$   
 $\langle proof \rangle$   
**lemmas** [sepref-fr-rules] =  $hm\text{-prio-of-op-impl.refine}$   
**sepref-register**  $PR\text{-CONST } hm\text{-prio-of-op}::(nat,'v) ahm \Rightarrow -$   
**lemma** [def-pat-rules]:  $hmstruct.hm\text{-prio-of-op}\$prio \equiv PR\text{-CONST } hm\text{-prio-of-op}$   
 $\langle proof \rangle$

No code theorem preparation, as we define optimized version later

**sepref-definition** (*no-prep-code*)  $hm\text{-swim-op-impl}$  **is** *uncurry* ( $PR\text{-CONST } hm\text{-swim-op}$ )  $:: hm\text{-impl-rel}^d *_a nat\text{-assn}^k \rightarrow_a hm\text{-impl-rel}$   
 $\langle proof \rangle$   
**lemmas** [sepref-fr-rules] =  $hm\text{-swim-op-impl.refine}$   
**sepref-register**  $PR\text{-CONST } hm\text{-swim-op}::(nat,'v) ahm \Rightarrow -$   
**lemma** [def-pat-rules]:  $hmstruct.hm\text{-swim-op}\$prio \equiv PR\text{-CONST } hm\text{-swim-op}$   
 $\langle proof \rangle$

No code theorem preparation, as we define optimized version later

**sepref-definition** (*no-prep-code*)  $hm\text{-sink-op-impl}$  **is** *uncurry* ( $PR\text{-CONST } hm\text{-sink-op}$ )  $:: hm\text{-impl-rel}^d *_a nat\text{-assn}^k \rightarrow_a hm\text{-impl-rel}$   
 $\langle proof \rangle$   
**lemmas** [sepref-fr-rules] =  $hm\text{-sink-op-impl.refine}$   
**sepref-register**  $PR\text{-CONST } hm\text{-sink-op}::(nat,'v) ahm \Rightarrow -$   
**lemma** [def-pat-rules]:  $hmstruct.hm\text{-sink-op}\$prio \equiv PR\text{-CONST } hm\text{-sink-op}$   
 $\langle proof \rangle$

**sepref-definition**  $hm\text{-repair-op-impl}$  **is** *uncurry* ( $PR\text{-CONST } hm\text{-repair-op}$ )  $:: hm\text{-impl-rel}^d *_a nat\text{-assn}^k \rightarrow_a hm\text{-impl-rel}$   
 $\langle proof \rangle$   
**lemmas** [sepref-fr-rules] =  $hm\text{-repair-op-impl.refine}$   
**sepref-register**  $PR\text{-CONST } hm\text{-repair-op}::(nat,'v) ahm \Rightarrow -$   
**lemma** [def-pat-rules]:  $hmstruct.hm\text{-repair-op}\$prio \equiv PR\text{-CONST } hm\text{-repair-op}$   
 $\langle proof \rangle$

### 3.12.3 Interface Operations

**definition**  $hm\text{-rel-np}$  **where**

$hm\text{-rel-np} \equiv hr\text{-comp } hm\text{-impl-rel } heapmap\text{-rel}$

**lemmas** [fcomp-norm-unfold] =  $hm\text{-rel-np-def}[symmetric]$

**definition**  $hm\text{-rel}$  **where**

$hm\text{-rel } K V \equiv hr\text{-comp } hm\text{-rel-np } (\langle the\text{-pure } K, the\text{-pure } V \rangle map\text{-rel})$

**lemmas** [fcomp-norm-unfold] =  $hm\text{-rel-def}[symmetric]$

**lemmas** [intf-of-assn] =  $intf\text{-of-assnI}[\mathbf{where } R=hm\text{-rel } K V \mathbf{ and } 'a=(kk,'vv) i\text{-map for } K V]$

**lemma**  $hm\text{-rel-id-conv}$ :  $hm\text{-rel } id\text{-assn } id\text{-assn} = hm\text{-rel-np}$

— Used for generic algorithms: Unfold with this, then let decl-impl compose with  $map\text{-rel}$  again.

$\langle proof \rangle$

## Synthesis

**definition**  $op-hm-empty-sz :: nat \Rightarrow 'kk \rightarrow 'vv$

**where**  $[simp]: op-hm-empty-sz sz \equiv op-map-empty$

**sepredef-register**  $PR-CONST (op-hm-empty-sz maxsize) :: ('k, 'v) i-map$

**lemma**  $[def-pat-rules]: op-hm-empty-sz \$maxsize \equiv UNPROTECT (op-hm-empty-sz maxsize) \langle proof \rangle$

**lemma**  $hm-fold-custom-empty-sz:$

$op-map-empty = op-hm-empty-sz sz$

$Map.empty = op-hm-empty-sz sz$

$\langle proof \rangle$

**sepredef-definition**  $hm-empty-op-impl$  **is**  $uncurry0 hm-empty-op' :: unit-assn^k \rightarrow_a hm-impl2-rel$

$\langle proof \rangle$

**sepredef-definition**  $hm-insert-op-impl$  **is**  $uncurry2 hm-insert-op :: [\lambda((k, -), -). k < maxsize]_a id-assn^k *_a id-assn^k *_a hm-impl-rel^d \rightarrow_a hm-impl-rel$

$\langle proof \rangle$

**sepredef-definition**  $hm-is-empty-op-impl$  **is**  $hm-is-empty-op :: hm-impl-rel^k \rightarrow_a bool-assn$

$\langle proof \rangle$

**sepredef-definition**  $hm-lookup-op-impl$  **is**  $uncurry hm-lookup-op :: id-assn^k *_a hm-impl-rel^k \rightarrow_a option-assn id-assn$

$\langle proof \rangle$

**sepredef-definition**  $hm-contains-key-impl$  **is**  $uncurry hm-contains-key-op' :: id-assn^k *_a hm-impl2-rel^k \rightarrow_a bool-assn$

$\langle proof \rangle$

**sepredef-definition**  $hm-decrease-key-op-impl$  **is**  $uncurry2 hm-decrease-key-op :: id-assn^k *_a id-assn^k *_a hm-impl-rel^d \rightarrow_a hm-impl-rel$

$\langle proof \rangle$

**sepredef-definition**  $hm-increase-key-op-impl$  **is**  $uncurry2 hm-increase-key-op :: id-assn^k *_a id-assn^k *_a hm-impl-rel^d \rightarrow_a hm-impl-rel$

$\langle proof \rangle$

**sepredef-definition**  $hm-change-key-op-impl$  **is**  $uncurry2 hm-change-key-op :: id-assn^k *_a id-assn^k *_a hm-impl-rel^d \rightarrow_a hm-impl-rel$

$\langle proof \rangle$

**sepredef-definition**  $hm-pop-min-op-impl$  **is**  $hm-pop-min-op :: hm-impl-rel^d \rightarrow_a prod-assn (prod-assn nat-assn id-assn) hm-impl-rel$



*<proof>*

**sepref-definition** *hm-remove-op-impl* is *uncurry hm-remove-op :: id-assn<sup>k</sup> \*<sub>a</sub> hm-impl-rel<sup>d</sup> →<sub>a</sub> hm-impl-rel*  
*<proof>*

**sepref-definition** *hm-peek-min-op-impl* is *hm-peek-min-op :: hm-impl-rel<sup>k</sup> →<sub>a</sub> prod-assn nat-assn id-assn*  
*<proof>*

## Setup of Refinements

**sepref-decl-impl** (*no-register*) *hm-empty*:  
*hm-empty-op-impl.refine[FCOMP hm-empty-op'-refine, FCOMP hm-empty-aref]*  
*<proof>*

**context** fixes *K* assumes *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** (*ismop*) *hm-insert*: *hm-insert-op-impl.refine[FCOMP hm-insert-op-aref]*  
  **uses** *mop-map-update-new-fref'*  
  *<proof>*

**sepref-decl-impl** *hm-is-empty*: *hm-is-empty-op-impl.refine[FCOMP hm-is-empty-op-aref]*  
*<proof>*  
**sepref-decl-impl** *hm-lookup*: *hm-lookup-op-impl.refine[FCOMP hm-lookup-op-aref]*  
*<proof>*

**sepref-decl-impl** *hm-contains-key*:  
*hm-contains-key-impl.refine[FCOMP hm-contains-key-op'-refine, FCOMP hm-contains-key-op-aref]*  
*<proof>*

**sepref-decl-impl** (*ismop*) *hm-decrease-key*: *hm-decrease-key-op-impl.refine[FCOMP hm-decrease-key-op-aref]* *<proof>*  
**sepref-decl-impl** (*ismop*) *hm-increase-key*: *hm-increase-key-op-impl.refine[FCOMP hm-increase-key-op-aref]* *<proof>*  
**sepref-decl-impl** (*ismop*) *hm-change-key*: *hm-change-key-op-impl.refine[FCOMP hm-change-key-op-aref]* *<proof>*

**sepref-decl-impl** (*ismop*) *hm-remove*: *hm-remove-op-impl.refine[FCOMP hm-remove-op-aref]*  
*<proof>*

**sepref-decl-impl** (*ismop*) *hm-pop-min*: *hm-pop-min-op-impl.refine[FCOMP hm-pop-min-op-aref]*  
*<proof>*  
**sepref-decl-impl** (*ismop*) *hm-peek-min*: *hm-peek-min-op-impl.refine[FCOMP hm-peek-min-op-aref]*  
*<proof>*  
**sepref-definition** *hm-upd-op-impl* is *uncurry2 (RETURN ooo op-map-update) ::*

$[\lambda((k,-),-). k < \text{maxsize}]_a \text{id-assn}^k *_a \text{id-assn}^k *_a (\text{hm-rel id-assn id-assn})^d \rightarrow \text{hm-rel id-assn id-assn}$

$\langle \text{proof} \rangle$

**sepref-decl-impl**  $\text{hm-upd-op-impl.refine}[\text{unfolded hm-rel-id-conv}]$  **uses**  $\text{op-map-update-fref}'$   
 $\langle \text{proof} \rangle$

**end**  
**end**

**interpretation**  $\text{hm}$ :  $\text{map-custom-empty PR-CONST (op-hm-empty-sz maxsize)}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{op-hm-empty-sz-hnr}[\text{sepref-fr-rules}]$ :  
 $(\text{uncurry0 (hm-empty-op-impl maxsize), uncurry0 (RETURN (PR-CONST (op-hm-empty-sz maxsize)))) \in \text{unit-assn}^k \rightarrow_a \text{hm-rel maxsize prio } K \ V$   
 $\langle \text{proof} \rangle$

### 3.12.4 Manual fine-tuning of code-lemmas

**context**

**notes**  $[\text{simp del}] = \text{CNV-def efficient-nat-div2}$

**begin**

**lemma**  $\text{nested-case-bind}$ :

$(\text{case } p \text{ of } (a,b) \Rightarrow \text{bind (case } a \text{ of } (a1,a2) \Rightarrow m \ a \ b \ a1 \ a2) (f \ a \ b))$   
 $= (\text{case } p \text{ of } ((a1,a2),b) \Rightarrow \text{bind (m (a1,a2) b a1 a2) (f (a1,a2) b)})$   
 $(\text{case } p \text{ of } (a,b) \Rightarrow \text{bind (case } b \text{ of } (b1,b2) \Rightarrow m \ a \ b \ b1 \ b2) (f \ a \ b))$   
 $= (\text{case } p \text{ of } (a,b1,b2) \Rightarrow \text{bind (m a (b1,b2) b1 b2) (f a (b1,b2))})$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{it-case}$ :  $(\text{case } p \text{ of } (a,b) \Rightarrow f \ p \ a \ b) = (\text{case } p \text{ of } (a,b) \Rightarrow f \ (a,b) \ a \ b)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{c2l}$ :  $(\text{case } p \text{ of } (a,b) \Rightarrow \text{bind (m a b) (f a b)}) =$   
 $\text{do } \{ \text{let } (a,b) = p; \text{bind (m a b) (f a b)} \}$   $\langle \text{proof} \rangle$

**lemma**  $\text{bind-Let}$ :  $\text{do } \{ x \leftarrow \text{do } \{ \text{let } y = v; (f \ y :: 'a \ \text{Heap}) \}; g \ x \} = \text{do } \{ \text{let } y=v;$   
 $x \leftarrow f \ y; g \ x \}$   $\langle \text{proof} \rangle$

**lemma**  $\text{bind-case}$ :  $\text{do } \{ x \leftarrow (\text{case } y \text{ of } (a,b) \Rightarrow f \ a \ b); (g \ x :: 'a \ \text{Heap}) \} = \text{do } \{$   
 $\text{let } (a,b) = y; x \leftarrow f \ a \ b; g \ x \}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{bind-case-mvup}$ :  $\text{do } \{ x \leftarrow f; \text{case } y \text{ of } (a,b) \Rightarrow g \ a \ b \ x \}$   
 $= \text{do } \{ \text{let } (a,b) = y; x \leftarrow f; (g \ a \ b \ x :: 'a \ \text{Heap}) \}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{if-case-mvup}$ :  $(\text{if } b \text{ then case } p \text{ of } (x1,x2) \Rightarrow f \ x1 \ x2 \ \text{else } e)$   
 $= (\text{case } p \text{ of } (x1,x2) \Rightarrow \text{if } b \text{ then } f \ x1 \ x2 \ \text{else } e)$   $\langle \text{proof} \rangle$

**lemma** *nested-case*:  $(\text{case } p \text{ of } (a,b) \Rightarrow (\text{case } p \text{ of } (c,d) \Rightarrow f a b c d)) =$   
 $(\text{case } p \text{ of } (a,b) \Rightarrow f a b a b)$   
 $\langle \text{proof} \rangle$

**lemma** *split-prod-bound*:  $(\lambda p. f p) = (\lambda(a,b). f (a,b)) \langle \text{proof} \rangle$

**lemma** *bpc-conv*:  $\text{do } \{ (a,b) \leftarrow (m::(-*) \text{Heap}); f a b \} = \text{do } \{$   
 $ab \leftarrow (m);$   
 $f (\text{fst } ab) (\text{snd } ab)$   
 $\}$   
 $\langle \text{proof} \rangle$

**lemma** *it-case-pp*:  $(\text{case } p \text{ of } ((p1,p2)) \Rightarrow \text{case } p \text{ of } ((p1',p2')) \Rightarrow f p1 p2 p1' p2')$   
 $= (\text{case } p \text{ of } ((p1,p2)) \Rightarrow f p1 p2 p1 p2)$   
 $\langle \text{proof} \rangle$

**lemma** *it-case-ppp*:  $(\text{case } p \text{ of } ((p1,p2),p3) \Rightarrow \text{case } p \text{ of } ((p1',p2'),p3') \Rightarrow f p1$   
 $p2 p3 p1' p2' p3')$   
 $= (\text{case } p \text{ of } ((p1,p2),p3) \Rightarrow f p1 p2 p3 p1 p2 p3)$   
 $\langle \text{proof} \rangle$

**lemma** *it-case-pppp*:  $(\text{case } a1 \text{ of}$   
 $((a, b), c), d) \Rightarrow$   
 $\text{case } a1 \text{ of}$   
 $((a', b'), c'), d') \Rightarrow f a b c d a' b' c' d') =$   
 $(\text{case } a1 \text{ of}$   
 $((a, b), c), d) \Rightarrow f a b c d a b c d)$   
 $\langle \text{proof} \rangle$  **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-of-op-impl-def hm-key-of-op-impl-def*  
*ial-get-def hm-the-lookup-op-impl-def heap-array-set-def marl-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*  
 $\langle \text{proof} \rangle$

**schematic-goal** *hm-swim-op-impl prio hm i ≡ ?f*  
 $\langle \text{proof} \rangle$

**lemma** *hm-swim-op-impl-code*[code]: *hm-swim-op-impl prio hm i ≡ ccpo.fixp (fun-lub*  
*Heap-lub) (fun-ord Heap-ord)*  
 $(\lambda cf (a1, a2).$   
 $\text{case } a1 \text{ of}$

```

((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
  })
(hm, i)
⟨proof⟩

```

**prepare-code-thms** *hm-swim-op-impl-code*

**schematic-goal** *hm-sink-opt-impl-code*[code]: *hm-sink-op-impl prio hm i ≡ ?f*  
 ⟨proof⟩

**prepare-code-thms** *hm-sink-opt-impl-code*

**export-code** *hm-swim-op-impl* in *SML-imp* **module-name** *Test*

**schematic-goal** *hm-change-key-opt-impl-code*[code]:  
*hm-change-key-op-impl prio k v hm ≡ ?f*  
 ⟨proof⟩

**schematic-goal** *hm-change-key-opt-impl-code*[code]:  
*hm-change-key-op-impl prio k v hm ≡ case hm of (((a, b), ba), x2) ⇒*  
 (do {  
 x ← Array.nth ba k;  
 xa ← Array.nth a x;  
 xa ← Array.upd xa v x2;  
 hm-repair-op-impl prio (((a, b), ba), xa) (Suc x)  
 })  
 ⟨proof⟩

**schematic-goal** *hm-set-opt-impl-code*[code]: *hm-upd-op-impl maxsize prio hm k v*  
 ≡ ?f

```

    <proof>

schematic-goal hm-pop-min-opt-impl-code[code]: hm-pop-min-op-impl prio hm ≡
    ?f
    <proof>

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 HICF-Matrix
imports ../Sepref
begin

```

#### 3.13.1 Relator and Interface

**definition** [to-relAPP]:  $mtx-rel\ A \equiv nat-rel \times_r nat-rel \rightarrow A$

**lemma**  $mtx-rel-id[simp]$ :  $\langle Id \rangle mtx-rel = Id$  <proof>

**type-synonym** 'a  $mtx = nat \times nat \Rightarrow 'a$   
**sepref-decl-intf** 'a  $i-mtx$  **is**  $nat \times nat \Rightarrow 'a$

**lemma** [synth-rules]:  $INTF-OF-REL\ A\ TYPE('a) \Longrightarrow INTF-OF-REL\ (\langle A \rangle mtx-rel)$   
 $TYPE('a\ i-mtx)$   
 <proof>

#### 3.13.2 Operations

**definition**  $op-mtx-new :: 'a\ mtx \Rightarrow 'a\ mtx$  **where** [simp]:  $op-mtx-new\ c \equiv c$

**sepref-decl-op** (*no-def*) *mtx-new*: *op-mtx-new* :: (*nat-rel* ×<sub>*r*</sub> *nat-rel* → *A*) →  $\langle A \rangle$  *mtx-rel*  $\langle$  *proof*  $\rangle$

**lemma** *mtx-init-adhoc-frame-match-rule*[*sepref-frame-match-rules*]:  
 $hn\text{-}val\ (nat\text{-}rel \times_r\ nat\text{-}rel \rightarrow A)\ x\ y \implies_t\ hn\text{-}val\ (nat\text{-}rel \times_r\ nat\text{-}rel \rightarrow the\text{-}pure\ (pure\ A))\ x\ y$   
 $\langle$  *proof*  $\rangle$

**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* ::  $\langle A \rangle$  *mtx-rel* →  $\langle A \rangle$  *mtx-rel*  $\langle$  *proof*  $\rangle$

**sepref-decl-op** *mtx-get*:  $\lambda(c :: 'a\ mtx)\ ij.\ c\ ij$  ::  $\langle A \rangle$  *mtx-rel* → (*nat-rel* ×<sub>*r*</sub> *nat-rel*) → *A*  $\langle$  *proof*  $\rangle$

**sepref-decl-op** *mtx-set*: *fun-upd*::'*a* *mtx* ⇒ - ::  $\langle A \rangle$  *mtx-rel* → (*nat-rel* ×<sub>*r*</sub> *nat-rel*) → *A* →  $\langle A \rangle$  *mtx-rel*  $\langle$  *proof*  $\rangle$

**definition** *mtx-nonzero* :: - *mtx* ⇒ (*nat* × *nat*) *set* **where** *mtx-nonzero* *m* ≡ {(*i,j*), *m* (*i,j*) ≠ 0}

**sepref-decl-op** *mtx-nonzero*: *mtx-nonzero* ::  $\langle A \rangle$  *mtx-rel* → (*nat-rel* ×<sub>*r*</sub> *nat-rel*) *set-rel* **where** *IS-ID* (*A*::(- × (- :: *zero*)) *set*)  $\langle$  *proof*  $\rangle$

### 3.13.3 Patterns

**lemma** *pat-amtx-get*:  $c \equiv op\text{-}mtx\text{-}get\ c' e$   $\langle$  *proof*  $\rangle$

**lemma** *pat-amtx-set*:  $fun\text{-}upd\ c \equiv op\text{-}mtx\text{-}set\ c' e' v$   $\langle$  *proof*  $\rangle$

**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*]:  $p \neq p' \implies q\ (f\ p\ s)\ p' = q\ s\ p'$   
**assumes** *upd-indep2*[*simp*, *intro*]:  $p \neq p' \implies q\ (f\ p\ (f\ p'\ s))\ p = q\ (f\ p\ s)\ p$   
**begin**  
**lemma** *pointwise-upd-fold*:  $distinct\ ps \implies q\ (fold\ f\ ps\ s)\ p = (if\ p \in set\ ps\ then\ q\ (f\ p\ s)\ p\ else\ q\ s\ p)$   
 $\langle$  *proof*  $\rangle$

**end**

**lemma** *pointwise-fun-fold*:

**fixes**  $f :: 'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$

**fixes**  $s :: 'a \Rightarrow 'b$

**assumes** *indep1*:  $\bigwedge x x' s. x \neq x' \implies f x s x' = s x'$

**assumes** *indep2*:  $\bigwedge x x' s. x \neq x' \implies f x (f x' s) x = f x s x$

**assumes** [*simp*]: *distinct xs*

**shows**  $\text{fold } f \text{ } xs \text{ } s \text{ } x = (\text{if } x \in \text{set } xs \text{ then } f x s x \text{ else } s x)$

*<proof>*

**lemma** *list-prod-divmod-eq*:  $\text{List.product } [0..<M] [0..<N] = \text{map } (\lambda i. (i \text{ div } N, i \text{ mod } N)) [0..<N * M]$

*<proof>*

**lemma** *nfoldli-prod-divmod-conv*:

$\text{nfoldli } (\text{List.product } [0..<N] [0..<M]) \text{ ctd } (\lambda(i,j). f i j) = \text{nfoldli } [0..<N * M] \text{ ctd } (\lambda i. f (i \text{ div } M) (i \text{ mod } M))$

*<proof>*

**lemma** *nfoldli-prod-divmod-conv'*:

$\text{nfoldli } [0..<M] \text{ ctd } (\lambda i. \text{nfoldli } [0..<N] \text{ ctd } (f i)) = \text{nfoldli } [0..<N * M] \text{ ctd } (\lambda i. f (i \text{ div } N) (i \text{ mod } N))$

*<proof>*

**lemma** *foldli-prod-divmod-conv'*:

$\text{foldli } [0..<M] \text{ ctd } (\lambda i. \text{foldli } [0..<N] \text{ ctd } (f i)) = \text{foldli } [0..<N * M] \text{ ctd } (\lambda i. f (i \text{ div } N) (i \text{ mod } N))$

(**is** ?lhs=?rhs)

*<proof>*

**lemma** *fold-prod-divmod-conv'*:  $\text{fold } (\lambda i. \text{fold } (f i) [0..<N]) [0..<M] = \text{fold } (\lambda i. f (i \text{ div } N) (i \text{ mod } N)) [0..<N * M]$

*<proof>*

**lemma** *mtx-nonzero-cases*[*consumes 0, case-names nonzero zero*]:

**obtains**  $(i,j) \in \text{mtx-nonzero } m \mid m (i,j) = 0$

*<proof>*

## Unary Pointwise

**definition** *mtx-pointwise-unop* ::  $(\text{nat} \times \text{nat} \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a \text{ mtx} \Rightarrow 'a \text{ mtx}$   
**where**

$\text{mtx-pointwise-unop } f \text{ } m \equiv \lambda(i,j). f (i,j) (m(i,j))$

**context** **fixes**  $f :: \text{nat} \times \text{nat} \Rightarrow 'a \Rightarrow 'a$  **begin**

**sepref-register** *PR-CONST* (*mtx-pointwise-unop f*) :: 'a i-mtx ⇒ 'a i-mtx  
**lemma** [*def-pat-rules*]: *mtx-pointwise-unop*\$f ≡ UNPROTECT (*mtx-pointwise-unop f*) *<proof>*  
**end**

**locale** *mtx-pointwise-unop-loc* =  
**fixes** *N* :: nat **and** *M* :: nat  
**fixes** *f* :: (nat × nat) ⇒ 'a::{zero} ⇒ 'a  
**assumes** *pres-zero[simp]*:  $\llbracket i \geq N \vee j \geq M \rrbracket \implies f(i,j) = 0$   
**begin**  
**definition** *opr-fold-impl* ≡ fold ( $\lambda i. \text{fold} (\lambda j m. m(i,j) := f(i,j) (m(i,j)))$ )  
 $[0..<M]$   $[0..<N]$

**lemma** *opr-fold-impl-eq*:  
**assumes** *mtx-nonzero*  $m \subseteq \{0..<N\} \times \{0..<M\}$   
**shows** *mtx-pointwise-unop f m* = *opr-fold-impl m*  
*<proof>*

**lemma** *opr-fold-impl-refine*: (*opr-fold-impl*, *mtx-pointwise-unop f*) ∈  $[\lambda m. \text{mtx-nonzero } m \subseteq \{0..<N\} \times \{0..<M\}]_f \text{Id} \rightarrow \text{Id}$   
*<proof>*

**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* ⊆  $\{0..<N\} \times \{0..<M\}$   
**assumes** *get-impl-hnr*:  
 $(\text{uncurry } \text{get-impl}, \text{uncurry} (\text{RETURN } \text{oo } \text{op-mtx-get})) \in \text{assn}^k *_a (\text{prod-assn } (\text{nbn-assn } N) (\text{nbn-assn } M))^k \rightarrow_a A$   
**assumes** *set-impl-hnr*:  
 $(\text{uncurry2 } \text{set-impl}, \text{uncurry2} (\text{RETURN } \text{ooo } \text{op-mtx-set})) \in \text{assn}^d *_a (\text{prod-assn } (\text{nbn-assn } N) (\text{nbn-assn } M))^k *_a A^k \rightarrow_a \text{assn}$   
**assumes** *fi-hnr*:  
 $(\text{uncurry } \text{fi}, \text{uncurry} (\text{RETURN } \text{oo } f)) \in (\text{prod-assn } \text{nat-assn } \text{nat-assn})^k *_a A^k \rightarrow_a A$   
**begin**

**lemma** *this-loc*: *mtx-pointwise-unop-gen-impl N M f assn A get-impl set-impl fi*  
*<proof>*

**context**

**notes** [*sepref-register-adhoc f 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 set-impl-hnr fi-hnr
begin
  sepref-thm opr-fold-impl1 is RETURN o opr-fold-impl ::  $assn^d \rightarrow_a assn$ 
    ⟨proof⟩
end

concrete-definition (in -) mtz-pointwise-unnop-fold-impl1 uses mtz-pointwise-unop-gen-impl.opr-fold-impl
prepare-code-thms (in -) mtz-pointwise-unnop-fold-impl1-def

lemma op-hnr[sepref-fr-rules]: (mtz-pointwise-unnop-fold-impl1 N M get-impl
set-impl fi, RETURN o PR-CONST (mtz-pointwise-unop f)) ∈  $assn^d \rightarrow_a assn$ 
  ⟨proof⟩

end

```

## Binary Pointwise

```

definition mtz-pointwise-binop :: ('a ⇒ 'a ⇒ 'a) ⇒ 'a mtz ⇒ 'a mtz ⇒ 'a mtz
where
  mtz-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 (mtz-pointwise-binop f) :: 'a i-mtz ⇒ 'a i-mtz ⇒
    'a i-mtz
lemma [def-pat-rules]: mtz-pointwise-binop$f ≡ UNPROTECT (mtz-pointwise-binop
f) ⟨proof⟩
end

```

```

locale mtz-pointwise-binop-loc =
  fixes N :: nat and M :: nat
  fixes f :: 'a::{zero} ⇒ 'a ⇒ 'a
  assumes pres-zero[simp]: f 0 0 = 0
begin
  definition opr-fold-impl m n ≡ fold (λi. fold (λj m. m( (i,j) := f (m(i,j))
(n(i,j)) )) [0..

```

```

lemma opr-fold-impl-eq:
  assumes mtz-nonzero m ⊆ {0..assumes mtz-nonzero n ⊆ {0..shows mtz-pointwise-binop f m n = opr-fold-impl m n
  ⟨proof⟩

```

```

lemma opr-fold-impl-refine: (uncurry opr-fold-impl, uncurry (mtz-pointwise-binop
f)) ∈ [λ(m,n). mtz-nonzero m ⊆ {0..f
Id ×r Id → Id
  ⟨proof⟩

```

**end**

```

locale mtz-pointwise-binop-gen-impl = mtz-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)) ∈ assnk *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)) ∈ assnd *a (prod-assn
(nbn-assn N) (nbn-assn M))k *a Ak →a assn
assumes fi-hnr:
  (uncurry fi,uncurry (RETURN oo f)) ∈ Ak *a Ak →a A
begin

lemma this-loc: mtx-pointwise-binop-gen-impl N M f assn A get-impl set-impl
fi
  ⟨proof⟩

context
notes [sepref-register-adhoc f 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 set-impl-hnr fi-hnr
begin
sepref-thm opr-fold-impl1 is uncurry (RETURN oo opr-fold-impl) :: assnd*aassnk
→a assn
  ⟨proof⟩

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), uncurry (RETURN oo PR-CONST (mtx-pointwise-binop f)))
∈ assnd *a assnk →a assn
  ⟨proof⟩

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)))

```

```

context fixes  $f\ g :: 'a \Rightarrow 'a \Rightarrow \text{bool}$  begin
  sepredef-register PR-CONST (mtx-pointwise-cmpop  $f\ g$ ) :: ' $a$  i-mtx  $\Rightarrow 'a$  i-mtx
 $\Rightarrow \text{bool}$ 
  lemma [def-pat-rules]: mtx-pointwise-cmpop $f\ g \equiv \text{UNPROTECT}$  (mtx-pointwise-cmpop
 $f\ g$ )  $\langle \text{proof} \rangle$ 
end

```

**lemma** *mtx-nonzeroD*:

```

 $\llbracket \neg i < N; \text{mtx-nonzero } m \subseteq \{0..<N\} \times \{0..<M\} \rrbracket \Longrightarrow m(i,j) = 0$ 
 $\llbracket \neg j < M; \text{mtx-nonzero } m \subseteq \{0..<N\} \times \{0..<M\} \rrbracket \Longrightarrow m(i,j) = 0$ 
 $\langle \text{proof} \rangle$ 

```

**locale** *mtx-pointwise-cmpop-loc* =

```

fixes  $N :: \text{nat}$  and  $M :: \text{nat}$ 
fixes  $f\ g :: 'a :: \{\text{zero}\} \Rightarrow 'a \Rightarrow \text{bool}$ 
assumes pres-zero[simp]:  $f\ 0\ 0 = \text{True}$   $g\ 0\ 0 = \text{False}$ 

```

**begin**

```

definition opr-fold-impl  $m\ n \equiv \text{do}$  {
   $s \leftarrow \text{nfoldli}$  (List.product  $[0..<N]$   $[0..<M]$ ) ( $\lambda s. s \neq 2$ ) ( $\lambda(i,j) s. \text{do}$  {
    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
} ( $0 :: \text{nat}$ );
RETURN ( $s=1$ )
}

```

**lemma** *opr-fold-impl-eq*:

```

assumes mtx-nonzero  $m \subseteq \{0..<N\} \times \{0..<M\}$ 
assumes mtx-nonzero  $n \subseteq \{0..<N\} \times \{0..<M\}$ 
shows opr-fold-impl  $m\ n \leq \text{RETURN}$  (mtx-pointwise-cmpop  $f\ g\ m\ n$ )
 $\langle \text{proof} \rangle$ 

```

**lemma** *opr-fold-impl-refine*:

```

(uncurry opr-fold-impl, uncurry (RETURN oo mtx-pointwise-cmpop  $f\ g$ ))  $\in$ 
 $[\lambda(m,n). \text{mtx-nonzero } m \subseteq \{0..<N\} \times \{0..<M\} \wedge \text{mtx-nonzero } n \subseteq \{0..<N\} \times \{0..<M\}]_f$ 
 $\text{Id} \times_r \text{Id} \rightarrow \langle \text{bool-rel} \rangle \text{nres-rel}$ 
 $\langle \text{proof} \rangle$ 

```

**end**

**locale** *mtx-pointwise-cmpop-gen-impl* = *mtx-pointwise-cmpop-loc* +

```

fixes  $\text{assn} :: 'a\ \text{mtx} \Rightarrow 'i \Rightarrow \text{assn}$ 
fixes  $A :: 'a \Rightarrow 'ai \Rightarrow \text{assn}$ 

```

```

fixes get-impl :: 'i ⇒ nat×nat ⇒ 'ai Heap
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)) ∈ assnk *a (prod-assn
(nbn-assn N) (nbn-assn M))k →a A
assumes fi-hnr:
  (uncurry fi,uncurry (RETURN oo f)) ∈ Ak *a Ak →a bool-assn
assumes gi-hnr:
  (uncurry gi,uncurry (RETURN oo g)) ∈ Ak *a Ak →a bool-assn
begin

```

```

lemma this-loc: mtx-pointwise-cmpop-gen-impl N M f g assn A get-impl fi gi
  ⟨proof⟩

```

```

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 :: assnd*aassnk →a
bool-assn
  ⟨proof⟩

```

```

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))) ∈
assnd *a assnk →a bool-assn
  ⟨proof⟩

```

```

end

```

```

end

```

### 3.14 Matrices by Array (Row-Major)

```

theory IICF-Array-Matrix

```

```

imports ../Intf/IICF-Matrix Separation-Logic-Imperative-HOL.Array-Blit

```

```

begin

```

```

definition is-amtx N M c mtx ≡ ∃A l. mtx ↦a l * ↑(

```

$length\ l = N * M$   
 $\wedge (\forall i < N. \forall j < M. l(i * M + j) = c\ (i, j))$   
 $\wedge (\forall i\ j. (i \geq N \vee j \geq M) \longrightarrow c\ (i, j) = 0)$

**lemma** *is-amtx-precise*[safe-constraint-rules]: *precise (is-amtx N M)*  
 ⟨proof⟩

**lemma** *is-amtx-bounded*:  
**shows** *rdomp (is-amtx N M) m  $\implies$  mtx-nonzero m  $\subseteq$  {0.. $N$ } $\times$ {0.. $M$ }*  
 ⟨proof⟩

**definition** *mtx-tabulate N M c*  $\equiv$  *do* {  
 $m \leftarrow Array.new\ (N * M)\ 0$ ;  
 $(-, -, m) \leftarrow imp\text{-}for'\ 0\ (N * M)\ (\lambda k\ (i, j, m).\ do\ \{$   
 $\quad Array.upd\ k\ (c\ (i, j))\ m$ ;  
 $\quad let\ j = j + 1$ ;  
 $\quad if\ j < M\ then\ return\ (i, j, m)$   
 $\quad else\ return\ (i + 1, 0, m)$   
 $\})\ (0, 0, m)$ ;  
 $return\ m$   
 $\}$

**definition** *amtx-copy*  $\equiv$  *array-copy*

**definition** *amtx-dflt N M v*  $\equiv$  *Array.make (N \* M) ( $\lambda i.$  v)*

**definition** *mtx-get M mtx e*  $\equiv$  *Array.nth mtx (fst e \* M + snd e)*

**definition** *mtx-set M mtx e v*  $\equiv$  *Array.upd (fst e \* M + snd e) v mtx*

**lemma** *mtx-idx-valid*[simp]:  $\llbracket i < (N :: nat); j < M \rrbracket \implies i * M + j < N * M$   
 ⟨proof⟩

**lemma** *mtx-idx-unique-conv*[simp]:  
**fixes**  $M :: nat$   
**assumes**  $j < M\ j' < M$   
**shows**  $(i * M + j = i' * M + j') \longleftrightarrow (i = i' \wedge j = j')$   
 ⟨proof⟩

**lemma** *mtx-tabulate-rl*[sep-heap-rules]:  
**assumes** *NONZ: mtx-nonzero c  $\subseteq$  {0.. $N$ } $\times$ {0.. $M$ }*  
**shows**  $\langle emp \rangle\ mtx\text{-}tabulate\ N\ M\ c\ \langle IICF\text{-}Array\text{-}Matrix.is\text{-}amtx\ N\ M\ c \rangle$   
 ⟨proof⟩

**lemma** *mtx-copy-rl*[*sep-heap-rules*]:  
 $\langle is\text{-}amtx\ N\ M\ c\ mtx \rangle\ amtx\text{-}copy\ mtx\ \langle \lambda r. is\text{-}amtx\ N\ M\ c\ mtx * is\text{-}amtx\ N\ M\ c\ r \rangle$   
 $\langle proof \rangle$

**definition** *PRES-ZERO-UNIQUE*  $A \equiv (A\ \{\{0\}=\{0\}\} \wedge A^{-1}\ \{\{0\}=\{0\}\})$   
**lemma** *IS-ID-imp-PRES-ZERO-UNIQUE*[*constraint-rules*]: *IS-ID*  $A \implies PRES\text{-}ZERO\text{-}UNIQUE\ A$   
 $\langle proof \rangle$

**definition** *op-amtx-dfltNxM*  $::\ nat \Rightarrow nat \Rightarrow 'a::zero \Rightarrow nat \times nat \Rightarrow 'a$  **where**  
 $[simp]:\ op\text{-}amtx\text{-}dfltNxM\ N\ M\ v \equiv \lambda(i,j).\ if\ i < N \wedge j < M\ then\ v\ else\ 0$   
**context** *fixes*  $N\ M::nat$  **begin**  
**sepref-decl-op** (*no-def*) *op-amtx-dfltNxM*: *op-amtx-dfltNxM*  $N\ M :: A \rightarrow \langle A \rangle mtx\text{-}rel$   
**where** *CONSTRAINT PRES-ZERO-UNIQUE*  $A$   
 $\langle proof \rangle$   
**end**

**lemma** *mtx-dflt-rl*[*sep-heap-rules*]:  $\langle emp \rangle\ amtx\text{-}dflt\ N\ M\ k\ \langle is\text{-}amtx\ N\ M\ (op\text{-}amtx\text{-}dfltNxM\ N\ M\ k) \rangle$   
 $\langle proof \rangle$

**lemma** *mtx-get-rl*[*sep-heap-rules*]:  $\llbracket i < N; j < M \rrbracket \implies \langle is\text{-}amtx\ N\ M\ c\ mtx \rangle\ mtx\text{-}get\ M\ mtx\ (i,j)\ \langle \lambda r. is\text{-}amtx\ N\ M\ c\ mtx * \uparrow(r = c\ (i,j)) \rangle$   
 $\langle proof \rangle$

**lemma** *mtx-set-rl*[*sep-heap-rules*]:  $\llbracket i < N; j < M \rrbracket \implies \langle is\text{-}amtx\ N\ M\ c\ mtx \rangle\ mtx\text{-}set\ M\ mtx\ (i,j)\ v\ \langle \lambda r. is\text{-}amtx\ N\ M\ (c((i,j) := v))\ r \rangle$   
 $\langle proof \rangle$

**definition** *amtx-assn*  $N\ M\ A \equiv hr\text{-}comp\ (is\text{-}amtx\ N\ M)\ ((the\text{-}pure\ A)\ mtx\text{-}rel)$   
**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*  $A\ TYPE('a) \implies intf\text{-}of\text{-}assn\ (amtx\text{-}assn\ N\ M\ A)\ TYPE('a\ i\text{-}mtx)$   
 $\langle proof \rangle$

**abbreviation** *asmtx-assn*  $N\ A \equiv amtx\text{-}assn\ N\ N\ A$

**lemma** *mtx-rel-pres-zero*:  
**assumes** *PRES-ZERO-UNIQUE*  $A$   
**assumes**  $(m,m') \in \langle A \rangle mtx\text{-}rel$   
**shows**  $m\ ij = 0 \iff m'\ ij = 0$   
 $\langle proof \rangle$   
**apply1** (*clarsimp simp: IS-PURE-def PRES-ZERO-UNIQUE-def is-pure-conv mtx-rel-def*)

$\langle \text{proof} \rangle$  **applyS** (rule *IdI*[of *ij*]) **applyS** *auto*  
 $\langle \text{proof} \rangle$

**lemma** *amtx-assn-bounded*:

**assumes** *CONSTRAINT (IS-PURE PRES-ZERO-UNIQUE) A*  
**shows** *rdomp (amtx-assn N M A) m  $\implies$  mtx-nonzero m  $\subseteq$   $\{0..<N\} \times \{0..<M\}$*   
 $\langle \text{proof} \rangle$

**lemma** *mtx-tabulate-aref*:

(*mtx-tabulate N M, RETURN o op-mtx-new*)  
 $\in [\lambda c. \text{mtx-nonzero } c \subseteq \{0..<N\} \times \{0..<M\}]_a \text{id-assn}^k \rightarrow \text{ICF-Array-Matrix.is-amtx}$   
*N M*  
 $\langle \text{proof} \rangle$

**lemma** *mtx-copy-aref*:

(*amtx-copy, RETURN o op-mtx-copy*)  $\in (\text{is-amtx } N M)^k \rightarrow_a \text{is-amtx } N M$   
 $\langle \text{proof} \rangle$

**lemma** *mtx-nonzero-bid-eq*:

**assumes** *R  $\subseteq$  Id*  
**assumes** (*a, a'*)  $\in$  *Id*  $\rightarrow$  *R*  
**shows** *mtx-nonzero a = mtx-nonzero a'*  
 $\langle \text{proof} \rangle$

**lemma** *mtx-nonzero-zu-eq*:

**assumes** *PRES-ZERO-UNIQUE R*  
**assumes** (*a, a'*)  $\in$  *Id*  $\rightarrow$  *R*  
**shows** *mtx-nonzero a = mtx-nonzero a'*  
 $\langle \text{proof} \rangle$

**lemma** *op-mtx-new-fref'*:

*CONSTRAINT PRES-ZERO-UNIQUE A  $\implies$  (RETURN  $\circ$  op-mtx-new, RETURN  $\circ$  op-mtx-new)  $\in$  (nat-rel  $\times_r$  nat-rel  $\rightarrow$  A)  $\rightarrow_f \langle \langle A \rangle \text{mtx-rel} \rangle \text{nres-rel}$*   
 $\langle \text{proof} \rangle$

**sempref-decl-impl** (*no-register*) *amtx-new-by-tab: mtx-tabulate-aref uses op-mtx-new-fref'*  
 $\langle \text{proof} \rangle$

**sempref-decl-impl** *amtx-copy: mtx-copy-aref*  $\langle \text{proof} \rangle$

**definition** [*simp*]: *op-amtx-new (N::nat) (M::nat)  $\equiv$  op-mtx-new*

**lemma** *amtx-fold-custom-new*:

*op-mtx-new  $\equiv$  op-amtx-new N M*  
*mop-mtx-new  $\equiv$   $\lambda c. \text{RETURN (op-amtx-new N M c)}$*   
 $\langle \text{proof} \rangle$

**context fixes**  $N M :: \text{nat}$  **begin**  
**sepref-register**  $PR\text{-}CONST$  ( $op\text{-}amtx\text{-}new\ N\ M$ )  $:: (nat \times nat \Rightarrow 'a) \Rightarrow 'a$   
*i-mtx*  
**end**

**lemma**  $amtx\text{-}new\text{-}hnr$ [ $sepref\text{-}fr\text{-}rules$ ]:  
**fixes**  $A :: 'a :: zero \Rightarrow 'b :: \{zero, heap\} \Rightarrow \text{assn}$   
**shows**  $CONSTRAINT$  ( $IS\text{-}PURE\ PRES\text{-}ZERO\text{-}UNIQUE$ )  $A \implies$   
 $(mtx\text{-}tabulate\ N\ M, (RETURN \circ PR\text{-}CONST (op\text{-}amtx\text{-}new\ N\ M)))$   
 $\in [\lambda x. mtx\text{-}nonzero\ x \subseteq \{0..<N\} \times \{0..<M\}]_a (pure (nat\text{-}rel \times_r\ nat\text{-}rel \rightarrow$   
 $the\text{-}pure\ A))^k \rightarrow amtx\text{-}assn\ N\ M\ A$   
 $\langle proof \rangle$

**lemma** [ $def\text{-}pat\text{-}rules$ ]:  $op\text{-}amtx\text{-}new\ \$N\ \$M \equiv UNPROTECT (op\text{-}amtx\text{-}new\ N\ M)$   $\langle proof \rangle$

**context fixes**  $N M :: \text{nat}$  **notes** [ $param$ ] =  $IdI[of\ N]\ IdI[of\ M]$  **begin**

**lemma**  $mtx\text{-}dflt\text{-}aref$ :  
 $(amtx\text{-}dflt\ N\ M, RETURN \circ PR\text{-}CONST (op\text{-}amtx\text{-}dflt\ N\ M\ N\ M)) \in id\text{-}assn^k$   
 $\rightarrow_a is\text{-}amtx\ N\ M$   
 $\langle proof \rangle$   
**sepref-decl-impl**  $amtx\text{-}dflt$ :  $mtx\text{-}dflt\text{-}aref$   $\langle proof \rangle$

**lemma**  $amtx\text{-}get\text{-}aref$ :  
 $(uncurry (mtx\text{-}get\ M), uncurry (RETURN \circ op\text{-}mtx\text{-}get)) \in [\lambda(-, (i, j)). i < N$   
 $\wedge j < M]_a (is\text{-}amtx\ N\ M)^k *_a (prod\text{-}assn\ nat\text{-}assn\ nat\text{-}assn)^k \rightarrow id\text{-}assn$   
 $\langle proof \rangle$   
**sepref-decl-impl**  $amtx\text{-}get$ :  $amtx\text{-}get\text{-}aref$   $\langle proof \rangle$

**lemma**  $amtx\text{-}set\text{-}aref$ :  $(uncurry2 (mtx\text{-}set\ M), uncurry2 (RETURN \circ op\text{-}mtx\text{-}set))$   
 $\in [\lambda((-, (i, j)), -). i < N \wedge j < M]_a (is\text{-}amtx\ N\ M)^d *_a (prod\text{-}assn\ nat\text{-}assn$   
 $nat\text{-}assn)^k *_a id\text{-}assn^k \rightarrow is\text{-}amtx\ N\ M$   
 $\langle proof \rangle$

**sepref-decl-impl**  $amtx\text{-}set$ :  $amtx\text{-}set\text{-}aref$   $\langle proof \rangle$

**lemma**  $amtx\text{-}get\text{-}aref'$ :  
 $(uncurry (mtx\text{-}get\ M), uncurry (RETURN \circ op\text{-}mtx\text{-}get)) \in (is\text{-}amtx\ N\ M)^k$   
 $*_a (prod\text{-}assn (pure (nbn\text{-}rel\ N)) (pure (nbn\text{-}rel\ M)))^k \rightarrow_a id\text{-}assn$   
 $\langle proof \rangle$

**sepref-decl-impl**  $amtx\text{-}get'$ :  $amtx\text{-}get\text{-}aref'$   $\langle proof \rangle$

**lemma**  $amtx\text{-}set\text{-}aref'$ :  $(uncurry2 (mtx\text{-}set\ M), uncurry2 (RETURN \circ op\text{-}mtx\text{-}set))$   
 $\in (is\text{-}amtx\ N\ M)^d *_a (prod\text{-}assn (pure (nbn\text{-}rel\ N)) (pure (nbn\text{-}rel\ M)))^k *_a$



$id-assn^k \rightarrow_a is-amtx\ N\ M$   
 $\langle proof \rangle$

**sepref-decl-impl**  $amtx-set'$ :  $amtx-set-aref'$   $\langle proof \rangle$

**end**

### 3.14.1 Pointwise Operations

**context**

**fixes**  $M\ N :: nat$

**begin**

**sepref-decl-op**  $amtx-lin-get$ :  $\lambda f\ i. op-mtx-get\ f\ (i\ div\ M, i\ mod\ M) :: \langle A \rangle mtx-rel$   
 $\rightarrow nat-rel \rightarrow A$   
 $\langle proof \rangle$

**sepref-decl-op**  $amtx-lin-set$ :  $\lambda f\ i\ x. op-mtx-set\ f\ (i\ div\ M, i\ mod\ M)\ x ::$   
 $\langle A \rangle mtx-rel \rightarrow nat-rel \rightarrow A \rightarrow \langle A \rangle mtx-rel$   
 $\langle proof \rangle$

**lemma**  $op-amtx-lin-get-aref$ :  $(uncurry\ Array.nth, uncurry\ (RETURN\ oo\ PR-CONST\ op-amtx-lin-get)) \in [\lambda(-,i). i < N * M]_a (is-amtx\ N\ M)^k *_a nat-assn^k \rightarrow id-assn$   
 $\langle proof \rangle$

**sepref-decl-impl**  $amtx-lin-get$ :  $op-amtx-lin-get-aref$   $\langle proof \rangle$

**lemma**  $op-amtx-lin-set-aref$ :  $(uncurry2\ (\lambda m\ i\ x. Array.upd\ i\ x\ m), uncurry2\ (RETURN\ ooo\ PR-CONST\ op-amtx-lin-set)) \in [\lambda((-,i),-). i < N * M]_a (is-amtx\ N\ M)^d *_a nat-assn^k *_a id-assn^k \rightarrow is-amtx\ N\ M$   
 $\langle proof \rangle$

**sepref-decl-impl**  $amtx-lin-set$ :  $op-amtx-lin-set-aref$   $\langle proof \rangle$

**end**

**lemma**  $amtx-fold-lin-get$ :  $m\ (i\ div\ M, i\ mod\ M) = op-amtx-lin-get\ M\ m\ i$   $\langle proof \rangle$

**lemma**  $amtx-fold-lin-set$ :  $m\ ((i\ div\ M, i\ mod\ M) := x) = op-amtx-lin-set\ M\ m\ i\ x$   $\langle proof \rangle$

**locale**  $amtx-pointwise-unop-impl = mtx-pointwise-unop-loc +$

**fixes**  $A :: 'a \Rightarrow 'ai :: \{zero, heap\} \Rightarrow assn$

**fixes**  $fi :: nat \times nat \Rightarrow 'ai \Rightarrow 'ai\ Heap$

**assumes**  $fi-hnr$ :

$(uncurry\ fi, uncurry\ (RETURN\ oo\ f)) \in (prod-assn\ nat-assn\ nat-assn)^k *_a A^k$   
 $\rightarrow_a A$

**begin**

**lemma**  $this-loc$ :  $amtx-pointwise-unop-impl\ N\ M\ f\ A\ fi$   $\langle proof \rangle$

```

context
  assumes PURE: CONSTRAINT (IS-PURE PRES-ZERO-UNIQUE) A
begin
  context
    notes  $[[\text{sepref-register-adhoc } f \ N \ M]]$ 
    notes  $[\text{sepref-import-param}] = \text{IdI}[of \ N] \ \text{IdI}[of \ M]$ 
    notes  $[\text{sepref-fr-rules}] = \text{fi-hnr}$ 
    notes  $[\text{safe-constraint-rules}] = \text{PURE}$ 
    notes  $[\text{simp}] = \text{algebra-simps}$ 
  begin
    sepref-thm opr-fold-impl1 is RETURN o opr-fold-impl ::  $(\text{amtx-assn } N \ M \ A)^d \rightarrow_a \text{amtx-assn } N \ M \ A$ 
       $\langle \text{proof} \rangle$ 
    end
  end
  concrete-definition (in -) amtx-pointwise-unnop-fold-impl1 uses amtx-pointwise-unop-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  $(\text{amtx-pointwise-unnop-fold-impl1 } N \ M \ \text{fi}, \ \text{RETURN} \circ \text{PR-CONST} (\text{mtx-pointwise-unop } f)) \in (\text{amtx-assn } N \ M \ A)^d \rightarrow_a \text{amtx-assn } N \ M \ A$ 
       $\langle \text{proof} \rangle$ 
  end

  locale amtx-pointwise-binop-impl = mtx-pointwise-binop-loc +
    fixes  $A :: 'a \Rightarrow 'ai :: \{\text{zero, heap}\} \Rightarrow \text{assn}$ 
    fixes  $\text{fi} :: 'ai \Rightarrow 'ai \Rightarrow 'ai \ \text{Heap}$ 
    assumes  $\text{fi-hnr}: (\text{uncurry } \text{fi}, \text{uncurry } (\text{RETURN} \circ \text{f})) \in A^k *_{\text{a}} A^k \rightarrow_a A$ 
  begin

    lemma this-loc: amtx-pointwise-binop-impl f A fi
       $\langle \text{proof} \rangle$ 

    context
      notes  $[[\text{sepref-register-adhoc } f \ N \ M]]$ 
      notes  $[\text{sepref-import-param}] = \text{IdI}[of \ N] \ \text{IdI}[of \ M]$ 
      notes  $[\text{sepref-fr-rules}] = \text{fi-hnr}$ 
      assumes PURE[safe-constraint-rules]: CONSTRAINT (IS-PURE PRES-ZERO-UNIQUE)
    end
    begin
      sepref-thm opr-fold-impl1 is uncurry (RETURN oo opr-fold-impl) ::  $(\text{amtx-assn } N \ M \ A)^d *_{\text{a}} (\text{amtx-assn } N \ M \ A)^k \rightarrow_a \text{amtx-assn } N \ M \ A$ 
         $\langle \text{proof} \rangle$ 
    end
  end

```

**concrete-definition** (in  $-$ ) *amtx-pointwise-binop-fold-impl1* for  $fi\ N\ M$   
 uses *amtx-pointwise-binop-impl.opr-fold-impl1.refine-raw* is (uncurry  $?f,-$ ) $\in$ -  
 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)*))  $\in$  (*amtx-assn*  $N\ M\ A$ )<sup>d</sup> \*<sub>a</sub> (*amtx-assn*  $N\ M\ A$ )<sup>k</sup>  $\rightarrow_a$  *amtx-assn*  $N\ M\ A$   
 $\langle$ *proof* $\rangle$

end

**locale** *amtx-pointwise-mpop-impl* = *mtx-pointwise-mpop-loc* +

fixes  $A :: 'a \Rightarrow 'ai::\{zero,heap\} \Rightarrow$  *assn*

fixes  $fi :: 'ai \Rightarrow 'ai \Rightarrow$  *bool Heap*

fixes  $gi :: 'ai \Rightarrow 'ai \Rightarrow$  *bool Heap*

assumes *fi-hnr*:

(uncurry  $fi$ , uncurry (*RETURN* oo  $f$ ))  $\in$   $A^k$  \*<sub>a</sub>  $A^k$   $\rightarrow_a$  *bool-assn*

assumes *gi-hnr*:

(uncurry  $gi$ , uncurry (*RETURN* oo  $g$ ))  $\in$   $A^k$  \*<sub>a</sub>  $A^k$   $\rightarrow_a$  *bool-assn*

**begin**

**lemma** *this-loc: amtx-pointwise-mpop-impl f g A fi gi*

$\langle$ *proof* $\rangle$

**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*  $N\ M$   
 $A$ )<sup>d</sup>\*<sub>a</sub>(*amtx-assn*  $N\ M\ A$ )<sup>k</sup>  $\rightarrow_a$  *bool-assn*

$\langle$ *proof* $\rangle$

end

**concrete-definition** (in  $-$ ) *amtx-pointwise-mpop-fold-impl1* for  $N\ M\ fi\ gi$

uses *amtx-pointwise-mpop-impl.opr-fold-impl1.refine-raw* is (uncurry  $?f,-$ ) $\in$ -

prepare-code-thms (in  $-$ ) *amtx-pointwise-mpop-fold-impl1-def*

**lemma** *op-hnr*[*sepref-fr-rules*]:

assumes *PURE: CONSTRAINT (IS-PURE PRES-ZERO-UNIQUE) A*

shows (uncurry (*amtx-pointwise-mpop-fold-impl1*  $N\ M\ fi\ gi$ ), uncurry (*RETURN*  
 oo *PR-CONST (mtx-pointwise-mpop f g)*))  $\in$  (*amtx-assn*  $N\ M\ A$ )<sup>d</sup> \*<sub>a</sub> (*amtx-assn*  
 $N\ M\ A$ )<sup>k</sup>  $\rightarrow_a$  *bool-assn*

$\langle$ *proof* $\rangle$

**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**  $[sepref-import-param] = IdI[of N] IdI[of M]$

— 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**  $scmul :: 'a \Rightarrow 'a\ mtx \Rightarrow 'a\ mtx\ nres$  **where**

$scmul\ x\ m \equiv nfoldli\ [0..<N]\ (\lambda-. True)\ (\lambda i\ m.$

$ifoldli\ [0..<M]\ (\lambda-. True)\ (\lambda j\ m.\ do\ \{$

$let\ mij = m(i,j);$

$RETURN\ (m((i,j) := x * mij))$

$\}$

$)\ m$

$)\ m$

After declaration of an implementation for multiplication, refinement is straightforward. Note that we use the fixed  $N$  in the refinement assertions.

**private lemma**  $times-param: ((*),(*))::'a\Rightarrow- \in Id \rightarrow Id \rightarrow Id$   $\langle proof \rangle$

**context**

**notes**  $[sepref-import-param] = times-param$

**begin**

**sepref-definition**  $scmul-impl$

**is**  $uncurry\ scmul :: (id-assn^k *_{\mathbf{a}} (amtx-assn\ N\ M\ id-assn))^d \rightarrow_{\mathbf{a}} amtx-assn$   
 $N\ M\ id-assn)$

$\langle proof \rangle$

**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*<sup>k</sup> →<sub>a</sub> *nat-assn*  
 ⟨*proof*⟩

Initialization from function diagonal is more complicated: First, we have to define the function as a new constant

**qualified definition** *diagonalN* *k* ≡ λ(*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*]: *IICF-Array-Matrix.diagonalN*\$*N* ≡ *UNPROTECT diagonalN* ⟨*proof*⟩

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** (*return o diagonalN, RETURN o (PR-CONST diagonalN)*) ∈ *A*<sup>k</sup>  
 →<sub>a</sub> *pure (nat-rel ×<sub>r</sub> nat-rel → the-pure A)*  
 ⟨*proof*⟩

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*) ∈ *mtx-nonzero (diagonalN k)* ⇒  
*a* < *N* ∧ *b* < *N*

⟨*proof*⟩ **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*<sup>k</sup> →<sub>a</sub> *int-assn*  
 ⟨*proof*⟩

**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-Sepref-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**

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  $\rightarrow$  'v)  $\Rightarrow$  'm  $\Rightarrow$  *assn*

**fixes** *contains-key* :: 'k  $\Rightarrow$  'm  $\Rightarrow$  *bool Heap*

**assumes** *contains-key-rule*[*sep-heap-rules*]:

$\langle is-map\ m\ p \rangle\ contains-key\ k\ p\ \langle \lambda r. is-map\ m\ p\ * \uparrow(r \leftarrow k \in dom\ m) \rangle_t$

**locale** *gen-contains-key-by-lookup* = *imp-map-lookup*

**begin**

**definition** *contains-key* *k m*  $\equiv$  *do* {*r*  $\leftarrow$  *lookup k m*; *return* ( $\neg is-None\ r$ )}

**sublocale** *imp-map-contains-key is-map contains-key*

*<proof>*

**end**

**locale** *imp-list-tail* = *imp-list* +

**constrains** *is-list* :: 'a list  $\Rightarrow$  'l  $\Rightarrow$  assn  
**fixes** *tail* :: 'l  $\Rightarrow$  'l Heap  
**assumes** *tail-rule*[*sep-heap-rules*]:  
 $l \neq [] \Longrightarrow \langle \text{is-list } l \ p \rangle \text{ tail } p \langle \text{is-list } (tl \ l) \rangle_t$

**definition** *os-head* :: 'a::heap *os-list*  $\Rightarrow$  ('a) Heap **where**  
*os-head* *p*  $\equiv$  case *p* of  
 None  $\Rightarrow$  raise STR "os-Head: Empty list"  
 | Some *p*  $\Rightarrow$  do { *m*  $\leftarrow$  !*p*; return (val *m*) }

**primrec** *os-tl* :: 'a::heap *os-list*  $\Rightarrow$  ('a *os-list*) Heap **where**  
*os-tl* None = raise STR "os-tl: Empty list"  
 | *os-tl* (Some *p*) = do { *m*  $\leftarrow$  !*p*; return (next *m*) }

**interpretation** *os*: *imp-list-head* *os-list* *os-head*  
 $\langle \text{proof} \rangle$

**interpretation** *os*: *imp-list-tail* *os-list* *os-tl*  
 $\langle \text{proof} \rangle$

**definition** *cs-is-empty* :: 'a::heap *cs-list*  $\Rightarrow$  bool Heap **where**  
*cs-is-empty* *p*  $\equiv$  return (is-None *p*)  
**interpretation** *cs*: *imp-list-is-empty* *cs-list* *cs-is-empty*  
 $\langle \text{proof} \rangle$

**definition** *cs-head* :: 'a::heap *cs-list*  $\Rightarrow$  'a Heap **where**  
*cs-head* *p*  $\equiv$  case *p* of  
 None  $\Rightarrow$  raise STR "cs-head: Empty list"  
 | Some *p*  $\Rightarrow$  do { *n*  $\leftarrow$  !*p*; return (val *n*) }  
**interpretation** *cs*: *imp-list-head* *cs-list* *cs-head*  
 $\langle \text{proof} \rangle$

**definition** *cs-tail* :: 'a::heap *cs-list*  $\Rightarrow$  'a *cs-list* Heap **where**  
*cs-tail* *p*  $\equiv$  do { (-, *r*)  $\leftarrow$  *cs-pop* *p*; return *r* }  
**interpretation** *cs*: *imp-list-tail* *cs-list* *cs-tail*  
 $\langle \text{proof} \rangle$

**lemma** *is-hashmap-finite*[*simp*]:  $h \models \text{is-hashmap } m \ mi \Longrightarrow \text{finite } (\text{dom } m)$   
 $\langle \text{proof} \rangle$

**lemma** *is-hashset-finite*[*simp*]:  $h \models \text{is-hashset } s \ si \Longrightarrow \text{finite } s$   
 $\langle \text{proof} \rangle$

**definition** *ias-is-it*  $s$   $a$   $si \equiv \lambda(a',i).$   
 $\exists_A l. a \mapsto_a l * \uparrow(a'=a \wedge s=ias\text{-of-list } l \wedge (i=length\ l \wedge si=\{\}) \vee i < length\ l \wedge i \in s$   
 $\wedge si=s \cap \{x. x \geq i\} )$

**context begin**

**private function** *first-memb* **where**

```

first-memb  $lmax$   $a$   $i = do \{$ 
  if  $i < lmax$  then do {
     $x \leftarrow Array.nth\ a\ i;$ 
    if  $x$  then return  $i$  else first-memb  $lmax$   $a$  (Suc  $i$ )
  } else
  return  $i$ 
}

```

$\langle proof \rangle$

**termination**  $\langle proof \rangle$

**declare** *first-memb.simps*[*simp del*]

**private lemma** *first-memb-rl-aux*:

**assumes**  $lmax \leq length\ l$   $i \leq lmax$

**shows**

$\langle a \mapsto_a l \rangle$

*first-memb*  $lmax$   $a$   $i$

$\langle \lambda k. a \mapsto_a l * \uparrow(k \leq lmax \wedge (\forall j. i \leq j \wedge j < k \longrightarrow \neg !!j) \wedge i \leq k \wedge (k=lmax \vee$   
 $!!k)) \rangle$

$\langle proof \rangle$  **lemma** *first-memb-rl*[*sep-heap-rules*]:

**assumes**  $lmax \leq length\ l$   $i \leq lmax$

**shows**  $\langle a \mapsto_a l \rangle$

*first-memb*  $lmax$   $a$   $i$

$\langle \lambda k. a \mapsto_a l * \uparrow(ias\text{-of-list } l \cap \{i..<k\} = \{\}) \wedge i \leq k \wedge (k < lmax \wedge k \in ias\text{-of-list } l$   
 $\vee k=lmax) \rangle$

$\langle proof \rangle$

**definition** *ias-it-init*  $a = do \{$

$l \leftarrow Array.len\ a;$

$i \leftarrow first-memb\ l\ a\ 0;$

return  $(a,i)$

$\}$

**definition** *ias-it-has-next*  $\equiv \lambda(a,i). do \{$

$l \leftarrow Array.len\ a;$

return  $(i < l)$

$\}$

**definition** *ias-it-next*  $\equiv \lambda(a,i). do \{$

$l \leftarrow Array.len\ a;$

$i' \leftarrow first-memb\ l\ a\ (Suc\ i);$

return  $(i,(a,i'))$



}

**lemma** *ias-of-list-bound*: *ias-of-list*  $l \subseteq \{0..<length\ l\}$   $\langle proof \rangle$

**end**

**interpretation** *ias*: *imp-set-iterate is-ias ias-is-it ias-it-init ias-it-has-next ias-it-next*  
 $\langle proof \rangle$

**lemma** *ias-of-list-finite*[*simp*, *intro!*]: *finite* (*ias-of-list*  $l$ )  
 $\langle proof \rangle$

**lemma** *is-ias-finite*[*simp*]:  $h \models is-ias\ S\ x \implies finite\ S$   
 $\langle proof \rangle$

**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  $\cap$  it =  $\{\}$*

**shows**

$\langle is-it\ s\ si\ it\ iti\ * is-list\ l\ li \rangle$

*to-list-ga-rec it-has-next it-next l-prepend iti li*

$\langle \lambda r. \exists_A l'. is-set\ s\ si$

$* is-list\ l'\ r$

$* \uparrow(distinct\ l' \wedge set\ l' = set\ l \cup it) \rangle_t$

$\langle proof \rangle$

**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**

$\langle is-set\ s\ si \rangle$

*to-list-ga it-init it-has-next it-next*

*l-empty l-prepend si*

$\langle \lambda r. \exists_A l. is-set\ s\ si * is-list\ l\ r * true * \uparrow(distinct\ l \wedge set\ l = s) \rangle$

$\langle proof \rangle$

### 3.15.1 Binding Locales

**method** *solve-sepl-binding* = (  
  *unfold-locales*;

  (*unfold option-assn-pure-conv*)?;

*sep-auto*

*intro!*: *hfrefI 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*  $\rightarrow$  *'vi*)  $\Rightarrow$  *'m*  $\Rightarrow$  *assn*  
**begin**

**definition** *assn K V*  $\equiv$  *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)*)  
 $\in$  *unit-assn<sup>k</sup>  $\rightarrow_a$  is-map*  
*<proof>*

**sepref-decl-impl** (*no-register*) *empty*: *empty-hnr-aux <proof>*

**end**

**locale** *bind-map-is-empty* = *imp-map-is-empty + bind-map*

**begin**

**lemma** *is-empty-hnr-aux*: (*is-empty, RETURN o op-map-is-empty*)  $\in$  *is-map<sup>k</sup>*  
 $\rightarrow_a$  *bool-assn*  
*<proof>*

**sepref-decl-impl** *is-empty*: *is-empty-hnr-aux <proof>*

**end**

**locale** *bind-map-update* = *imp-map-update + bind-map*

**begin**

**lemma** *update-hnr-aux*: (*uncurry2 update, uncurry2 (RETURN ooo op-map-update)*)  
 $\in$  *id-assn<sup>k</sup> \*\_a id-assn<sup>k</sup> \*\_a is-map<sup>d</sup>  $\rightarrow_a$  is-map*  
*<proof>*

**sepref-decl-impl** *update*: *update-hnr-aux <proof>*

**end**

**locale** *bind-map-delete* = *imp-map-delete + bind-map*

**begin**

**lemma** *delete-hnr-aux*: (*uncurry delete, uncurry (RETURN oo op-map-delete)*)  
 $\in$  *id-assn<sup>k</sup> \*\_a is-map<sup>d</sup>  $\rightarrow_a$  is-map*  
*<proof>*

**sepref-decl-impl** *delete: delete-hnr-aux*  $\langle$ proof $\rangle$   
**end**

**locale** *bind-map-lookup* = *imp-map-lookup* + *bind-map*  
**begin**

**lemma** *lookup-hnr-aux*: (*uncurry lookup,uncurry (RETURN oo op-map-lookup)*)  
 $\in id-assn^k *_a is-map^k \rightarrow_a id-assn$   
 $\langle$ proof $\rangle$

**sepref-decl-impl** *lookup: lookup-hnr-aux*  $\langle$ proof $\rangle$   
**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)*)  
 $\in id-assn^k *_a is-map^k \rightarrow_a bool-assn$   
 $\langle$ proof $\rangle$

**sepref-decl-impl** *contains-key: contains-key-hnr-aux*  $\langle$ proof $\rangle$   
**end**

## Set

**locale** *bind-set* = *imp-set is-set* **for** *is-set* :: (*'ai set*)  $\Rightarrow$  *'m*  $\Rightarrow$  *assn* +  
**fixes** *A* :: *'a*  $\Rightarrow$  *'ai*  $\Rightarrow$  *assn*

**begin**

**definition** *assn*  $\equiv hr-comp is-set (\langle the-pure A \rangle set-rel)$

**lemmas** [*safe-constraint-rules*] = *CN-FALSEI*[*of is-pure assn*]

**end**

**locale** *bind-set-setup* = *bind-set*

**begin**

**lemmas** [*fcomp-norm-unfold*] = *assn-def*[*symmetric*]

**lemma** *APA*:  $\llbracket PROP Q; CONSTRAINT is-pure A \rrbracket \Longrightarrow PROP Q$   $\langle$ proof $\rangle$

**lemma** *APAlu*:  $\llbracket PROP Q; CONSTRAINT (IS-PURE IS-LEFT-UNIQUE) A \rrbracket$   
 $\Longrightarrow PROP Q$   $\langle$ proof $\rangle$

**lemma** *APArU*:  $\llbracket PROP Q; CONSTRAINT (IS-PURE IS-RIGHT-UNIQUE) A \rrbracket$   
 $\Longrightarrow PROP Q$   $\langle$ proof $\rangle$

**lemma** *APAbu*:  $\llbracket PROP Q; CONSTRAINT (IS-PURE IS-LEFT-UNIQUE) A; CONSTRAINT (IS-PURE IS-RIGHT-UNIQUE) A \rrbracket$   
 $\Longrightarrow PROP Q$   $\langle$ proof $\rangle$

**end**

**locale** *bind-set-empty* = *imp-set-empty* + *bind-set*

**begin**

**lemma** *hnr-empty-aux*: (*uncurry0 empty,uncurry0 (RETURN op-set-empty)*) $\in unit-assn^k$

$\rightarrow_a$  *is-set*  
 ⟨proof⟩

**interpretation** *bind-set-setup* ⟨proof⟩

**lemmas** *hnr-op-empty* = *hnr-empty-aux*[*FCOMP op-set-empty.fref*[**where** *A=the-pure A*]]

**lemmas** *hnr-mop-empty* = *hnr-op-empty*[*FCOMP mk-mop-rl0-np*[*OF mop-set-empty-alt*]]  
**end**

**locale** *bind-set-is-empty* = *imp-set-is-empty* + *bind-set*  
**begin**

**lemma** *hnr-is-empty-aux*: (*is-empty*, *RETURN o op-set-is-empty*) ∈ *is-set*<sup>k</sup>  $\rightarrow_a$   
*bool-assn*  
 ⟨proof⟩

**interpretation** *bind-set-setup* ⟨proof⟩

**lemmas** *hnr-op-is-empty*[*sepref-fr-rules*] = *hnr-is-empty-aux*[*THEN APA,FCOMP op-set-is-empty.fref*[**where** *A=the-pure A*]]

**lemmas** *hnr-mop-is-empty*[*sepref-fr-rules*] = *hnr-op-is-empty*[*FCOMP mk-mop-rl1-np*[*OF mop-set-is-empty-alt*]]  
**end**

**locale** *bind-set-member* = *imp-set-memb* + *bind-set*  
**begin**

**lemma** *hnr-member-aux*: (*uncurry memb*, *uncurry (RETURN oo op-set-member)*) ∈ *id-assn*<sup>k</sup>  
 $*_a$  *is-set*<sup>k</sup>  $\rightarrow_a$  *bool-assn*  
 ⟨proof⟩

**interpretation** *bind-set-setup* ⟨proof⟩

**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-op-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-assn*<sup>k</sup>  
 $*_a$  *is-set*<sup>d</sup>  $\rightarrow_a$  *is-set*  
 ⟨proof⟩

**interpretation** *bind-set-setup* ⟨proof⟩

**lemmas** *hnr-op-insert*[*sepref-fr-rules*] = *hnr-insert-aux*[*THEN APArU,FCOMP op-set-insert.fref*[**where** *A=the-pure A*]]

**lemmas** *hnr-mop-insert*[*sepref-fr-rules*] = *hnr-op-insert*[*FCOMP mk-mop-rl2-np*[*OF 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-assn*<sup>k</sup>  
<sup>\*<sub>a</sub></sup> *is-set*<sup>d</sup> →<sub>a</sub> *is-set*  
    ⟨*proof*⟩  
  
**interpretation** *bind-set-setup* ⟨*proof*⟩  
**lemmas** *hnr-op-delete*[*sepref-fr-rules*] = *hnr-delete-aux*[*THEN APAbu,FCOMP*  
*op-set-delete.fref*[**where** *A=the-pure A*]]  
**lemmas** *hnr-mop-delete*[*sepref-fr-rules*] = *hnr-op-delete*[*FCOMP mk-mop-rl2-np*[*OF*  
*mop-set-delete-alt*]]  
**end**  
  
**locale** *bind-set-size* = *imp-set-size* + *bind-set*  
**begin**  
**lemma** *hnr-size-aux*: (*size*, (*RETURN o op-set-size*)) ∈ *is-set*<sup>k</sup> →<sub>a</sub> *nat-assn*  
    ⟨*proof*⟩  
  
**interpretation** *bind-set-setup* ⟨*proof*⟩  
**lemmas** *hnr-op-size*[*sepref-fr-rules*] = *hnr-size-aux*[*THEN APAbu,FCOMP*  
*op-set-size.fref*[**where** *A=the-pure A*]]  
**lemmas** *hnr-mop-size*[*sepref-fr-rules*] = *hnr-op-size*[*FCOMP mk-mop-rl1-np*[*OF*  
*mop-set-size-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*⟩  
  
**lemma** *param-sorted-wrt*[*param*]: (*sorted-wrt*, *sorted-wrt*) ∈ (*A* → *A* → *bool-rel*)  
→ (*A*)*list-rel* → *bool-rel*  
    ⟨*proof*⟩  
  
**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*  
    ⟨*proof*⟩  
  
**lemma** *param-it-to-sorted-list*[*param*]: [*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*  
    ⟨*proof*⟩

**locale** *bind-set-iterate* = *imp-set-iterate* + *bind-set* +  
**assumes** *is-set-finite*:  $h \models is-set\ S\ x \implies finite\ S$   
**begin**  
**context begin**  
**private lemma** *is-imp-set-iterate*: *imp-set-iterate is-set is-it it-init it-has-next it-next*  $\langle proof \rangle$  **lemma** *is-imp-list-empty*: *imp-list-empty (list-assn id-assn)* (return [])  
 $\langle proof \rangle$  **lemma** *is-imp-list-prepend*: *imp-list-prepend (list-assn id-assn)* (return oo *List.Cons*)  
 $\langle proof \rangle$   
  
**definition** *to-list*  $\equiv to-list-ga\ it-init\ it-has-next\ it-next$  (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* ( $\lambda - . True$ )))  $\in is-set^k \rightarrow_a list-assn\ id-assn$   
 $\langle proof \rangle$  **lemma** *to-list-sorted2*:  $\llbracket CONSTRAINT\ (IS-PURE\ IS-LEFT-UNIQUE)\ A;$   
 $CONSTRAINT\ (IS-PURE\ IS-RIGHT-UNIQUE)\ A \rrbracket \implies$   
(*PR-CONST (it-to-sorted-list* ( $\lambda - . True$ )), *PR-CONST (it-to-sorted-list* ( $\lambda - . True$ )))  $\in \langle the-pure\ A \rangle set-rel \rightarrow \langle \langle the-pure\ A \rangle list-rel \rangle nres-rel$   
 $\langle proof \rangle$   
  
**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*]:  $\llbracket CONSTRAINT\ (IS-PURE\ IS-LEFT-UNIQUE)\ A;$   
 $CONSTRAINT\ (IS-PURE\ IS-RIGHT-UNIQUE)\ A \rrbracket \implies GEN-ALGO\ to-list\ (IS-TO-SORTED-LIST\ (\lambda - . True)\ (bind-set.assn\ is-set\ A)\ A)$   
 $\langle proof \rangle$   
  
**end**  
**end**  
  
**locale** *bind-set-union* = *imp-set-union* + *bind-set* +  
**assumes** *is-prime-set-finite*:  $h \models is-set\ S\ x \implies finite\ S$   
**begin**  
**lemma** *hnr-union-aux*: (*uncurry union, uncurry (RETURN oo op-set-union)*)  
 $\in is-set^d *_{\alpha} is-set^k \rightarrow_a is-set$   
 $\langle proof \rangle$   
  
**interpretation** *bind-set-setup*  $\langle proof \rangle$   
**lemmas** *hnr-op-union*[*sepref-fr-rules*] = *hnr-union-aux*[*FCOMP op-set-union.fref*][**where**  $A = the-pure\ A$ ]  
**lemmas** *hnr-mop-union*[*sepref-fr-rules*] = *hnr-op-union*[*FCOMP mk-mop-rl2-np*][*OF mop-set-union-alt*]

end

## List

**locale** *bind-list* = *imp-list is-list* **for** *is-list* :: ('ai list)  $\Rightarrow$  'm  $\Rightarrow$  *assn* +  
**fixes** *A* :: 'a  $\Rightarrow$  'ai  $\Rightarrow$  *assn*  
**begin**

**definition** *assn*  $\equiv$  *hr-comp is-list* (*the-pure A*)*list-rel*)  
**lemmas** [*safe-constraint-rules*] = *CN-FALSEI*[*of is-pure assn*]

end

**locale** *bind-list-empty* = *imp-list-empty* + *bind-list*  
**begin**

**lemma** *hnr-aux*: (*uncurry0 empty,uncurry0 (RETURN op-list-empty)*) $\in$ (*pure unit-rel*)<sup>k</sup>  $\rightarrow_a$  *is-list*  
*<proof>*

**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*) $\in$ (*is-list*)<sup>k</sup>  $\rightarrow_a$  *pure bool-rel*  
*<proof>*

**lemmas** *hnr*[*sepref-fr-rules*]  
= *hnr-aux*[*FCOMP op-list-is-empty.fref*, *of the-pure A*, *folded assn-def*]

**lemmas** *hnr-mop*[*sepref-fr-rules*] = *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)*)  
 $\in$ (*is-list*)<sup>d</sup> \*<sub>a</sub> (*pure Id*)<sup>k</sup>  $\rightarrow_a$  *is-list* *<proof>*

**lemmas** *hnr*[*sepref-fr-rules*]  
= *hnr-aux*[*FCOMP op-list-append.fref*, *of A*, *folded assn-def*]

**lemmas** *hnr-mop*[*sepref-fr-rules*] = *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)*)  
 $\in (\text{pure Id})^k *_a (\text{is-list})^d \rightarrow_a \text{is-list} \langle \text{proof} \rangle$

**lemmas** *hnr*[*sepref-fr-rules*]  
 $= \text{hnr-aux}[FCOMP \text{op-list-prepend.fref, of } A, \text{folded assn-def}]$

**lemmas** *hnr-mop*[*sepref-fr-rules*] = *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*)  
 $\in [\lambda l. l \neq []]_a (\text{is-list})^d \rightarrow \text{pure Id} \langle \text{proof} \rangle$

**lemmas** *hnr*[*sepref-fr-rules*] = *hnr-aux*[*FCOMP op-list-hd.fref, of A, folded assn-def*]  
**lemmas** *hnr-mop*[*sepref-fr-rules*] = *hnr*[*FCOMP mk-mop-rl1*[*OF mop-list-hd-alt*]]  
**end**

**locale** *bind-list-tl* = *imp-list-tail* + *bind-list*  
**begin**  
**lemma** *hnr-aux*: (*tail, RETURN o op-list-tl*)  
 $\in [\lambda l. l \neq []]_a (\text{is-list})^d \rightarrow \text{is-list} \langle \text{proof} \rangle$

**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*)  
 $\in (\text{is-list})^d \rightarrow_a \text{is-list} \langle \text{proof} \rangle$

**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*)  
 $\in (\text{is-list})^d \rightarrow_a \text{is-list} \langle \text{proof} \rangle$

**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*  $\equiv$  *IICF-Map.op-map-empty*  
**interpretation** *iam*: *bind-map-empty is-iam iam-new*  
 ⟨*proof*⟩

**interpretation** *iam*: *map-custom-empty op-iam-empty*  
 ⟨*proof*⟩

**lemmas** [*sepref-fr-rules*] = *iam.empty-hnr*[*folded op-iam-empty-def*]

**definition** [*simp*]: *op-iam-empty-sz* (*N::nat*)  $\equiv$  *IICF-Map.op-map-empty*

**lemma** [*def-pat-rules*]: *op-iam-empty-sz* \$*N*  $\equiv$  *UNPROTECT* (*op-iam-empty-sz* *N*)  
 ⟨*proof*⟩

**interpretation** *iam-sz*: *map-custom-empty PR-CONST* (*op-iam-empty-sz N*)  
 ⟨*proof*⟩

**lemma** [*sepref-fr-rules*]: (*uncurry0 iam-new*, *uncurry0* (*RETURN* (*PR-CONST* (*op-iam-empty-sz N*))))  $\in$  *unit-assn*<sup>*k*</sup>  $\rightarrow_a$  *iam.assn K V*  
 ⟨*proof*⟩

**interpretation** *iam*: *bind-map-update is-iam Array-Map-Impl.iam-update*  
 ⟨*proof*⟩

**interpretation** *iam*: *bind-map-delete is-iam Array-Map-Impl.iam-delete*  
 ⟨*proof*⟩

**interpretation** *iam*: *bind-map-lookup is-iam Array-Map-Impl.iam-lookup*  
 ⟨*proof*⟩

⟨*ML*⟩

**interpretation** *iam*: *gen-contains-key-by-lookup is-iam Array-Map-Impl.iam-lookup*  
 ⟨*proof*⟩

⟨*ML*⟩

**interpretation** *iam*: *bind-map-contains-key is-iam iam.contains-key*  
 ⟨*proof*⟩

### 3.15.3 Array Set (*ias*)

**definition** [*simp*]: *op-ias-empty*  $\equiv$  *op-set-empty*

**interpretation** *ias*: *bind-set-empty is-ias ias-new for A*  
 ⟨*proof*⟩

**interpretation** *ias*: *set-custom-empty ias-new op-ias-empty*

$\langle proof \rangle$

**lemmas** [sepref-fr-rules] = *ias.hnr-op-empty*[folded *op-ias-empty-def*]

**definition** [simp]: *op-ias-empty-sz* ( $N::nat$ )  $\equiv$  *op-set-empty*

**lemma** [def-pat-rules]: *op-ias-empty-sz* \$N  $\equiv$  *UNPROTECT* (*op-ias-empty-sz* N)

$\langle proof \rangle$

**interpretation** *ias-sz*: *bind-set-empty is-ias ias-new-sz* N **for** N A

$\langle proof \rangle$

**interpretation** *ias-sz*: *set-custom-empty ias-new-sz* N *PR-CONST* (*op-ias-empty-sz* N) **for** A

$\langle proof \rangle$

**lemma** [sepref-fr-rules]:

(*uncurry0* (*ias-new-sz* N), *uncurry0* (*RETURN* (*PR-CONST* (*op-ias-empty-sz* N))))  $\in$  *unit-assn*<sup>k</sup>  $\rightarrow_a$  *ias.assn* A

$\langle proof \rangle$

**interpretation** *ias*: *bind-set-member is-ias Array-Set-Impl.ias-memb* **for** A

$\langle proof \rangle$

**interpretation** *ias*: *bind-set-insert is-ias Array-Set-Impl.ias-ins* **for** A

$\langle proof \rangle$

**interpretation** *ias*: *bind-set-delete is-ias Array-Set-Impl.ias-delete* **for** A

$\langle proof \rangle$

$\langle ML \rangle$

**interpretation** *ias*: *bind-set-iterate is-ias ias-is-it ias-it-init ias-it-has-next ias-it-next* **for** A

$\langle proof \rangle$

$\langle ML \rangle$

### 3.15.4 Hash Map (hm)

**interpretation** *hm*: *bind-map-empty is-hashmap hm-new*

$\langle proof \rangle$

**definition** *op-hm-empty*  $\equiv$  *IICF-Map.op-map-empty*

**interpretation** *hm*: *map-custom-empty op-hm-empty*

$\langle proof \rangle$

**lemmas** [sepref-fr-rules] = *hm.empty-hnr*[folded *op-hm-empty-def*]

**interpretation** *hm*: *bind-map-is-empty is-hashmap Hash-Map.hm-isEmpty*

$\langle proof \rangle$

**interpretation** *hm*: *bind-map-update is-hashmap Hash-Map.hm-update*

$\langle proof \rangle$

**interpretation** *hm: bind-map-delete is-hashmap Hash-Map.hm-delete*  
⟨proof⟩

**interpretation** *hm: bind-map-lookup is-hashmap Hash-Map.hm-lookup*  
⟨proof⟩

⟨ML⟩

**interpretation** *hm: gen-contains-key-by-lookup is-hashmap Hash-Map.hm-lookup*  
⟨proof⟩  
⟨ML⟩

**interpretation** *hm: bind-map-contains-key is-hashmap hm.contains-key*  
⟨proof⟩

### 3.15.5 Hash Set (hs)

**interpretation** *hs: bind-set-empty is-hashset hs-new for A*  
⟨proof⟩

**definition** *op-hs-empty*  $\equiv$  *IICF-Set.op-set-empty*

**interpretation** *hs: set-custom-empty hs-new op-hs-empty for A*  
⟨proof⟩

**lemmas** [*sepref-fr-rules*] = *hs.hnr-op-empty*[*folded op-hs-empty-def*]

**interpretation** *hs: bind-set-is-empty is-hashset Hash-Set-Impl.hs-isEmpty for A*  
⟨proof⟩

**interpretation** *hs: bind-set-member is-hashset Hash-Set-Impl.hs-memb for A*  
⟨proof⟩

**interpretation** *hs: bind-set-insert is-hashset Hash-Set-Impl.hs-ins for A*  
⟨proof⟩

**interpretation** *hs: bind-set-delete is-hashset Hash-Set-Impl.hs-delete for A*  
⟨proof⟩

**interpretation** *hs: bind-set-size is-hashset hs-size for A*  
⟨proof⟩

⟨ML⟩

**interpretation** *hs: bind-set-iterate is-hashset hs-is-it hs-it-init hs-it-has-next*  
*hs-it-next for A*

⟨proof⟩

⟨ML⟩

**interpretation** *hs: bind-set-union is-hashset hs-is-it hs-it-init hs-it-has-next hs-it-next*  
*hs-union*  
**for A**

*<proof>*

### 3.15.6 Open Singly Linked List (osll)

**interpretation** *osll*: *bind-list os-list for A* *<proof>*

**interpretation** *osll-empty*: *bind-list-empty os-list os-empty for A*  
*<proof>*

**definition** *osll-empty*  $\equiv$  *op-list-empty*

**interpretation** *osll*: *list-custom-empty osll.assn A os-empty osll-empty*  
*<proof>*

**interpretation** *osll-is-empty*: *bind-list-is-empty os-list os-is-empty for A*  
*<proof>*

**interpretation** *osll-prepend*: *bind-list-prepend os-list os-prepend for A*  
*<proof>*

**interpretation** *osll-hd*: *bind-list-hd os-list os-head for A*  
*<proof>*

**interpretation** *osll-tl*: *bind-list-tl os-list os-tl for A*  
*<proof>*

**interpretation** *osll-rev*: *bind-list-rev os-list os-reverse for A*  
*<proof>*

### 3.15.7 Circular Singly Linked List (csll)

**interpretation** *csll*: *bind-list cs-list for A* *<proof>*

**interpretation** *csll-empty*: *bind-list-empty cs-list cs-empty for A*  
*<proof>*

**definition** *csll-empty*  $\equiv$  *op-list-empty*

**interpretation** *csll*: *list-custom-empty csll.assn A cs-empty csll-empty*  
*<proof>*

**interpretation** *csll-is-empty*: *bind-list-is-empty cs-list cs-is-empty for A*  
*<proof>*

**interpretation** *csll-prepend*: *bind-list-prepend cs-list cs-prepend for A*  
*<proof>*

**interpretation** *csll-append*: *bind-list-append cs-list cs-append for A*  
*<proof>*

**interpretation** *csll-hd*: *bind-list-hd cs-list cs-head for A*  
*<proof>*

**interpretation** *csl*-*tl*: *bind-list-tl cs-list cs-tail* **for** *A*  
⟨*proof*⟩

**interpretation** *csl*-*rotate1*: *bind-list-rotate1 cs-list cs-rotate* **for** *A*  
⟨*proof*⟩

**schematic-goal** *hn-refine* (*emp*) (?*c*::?'*c Heap*) ? $\Gamma$ ' ?*R* (do {  
  *x* ← *mop-list-empty*;  
  RETURN (*1* ∈ dom [*1*::*nat* ↦ *True*, *2*↦*False*], {*1,2*::*nat*}, *1*#(*2*::*nat*)#*x*)  
})  
⟨*proof*⟩

**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 ../HICF/HICF  
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. Sepref 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:

**definition**  $min\text{-of-list} :: 'a::linorder\ list \Rightarrow 'a\ nres$  **where**  
 $min\text{-of-list}\ l \equiv ASSERT\ (l \neq []) \gg SPEC\ (\lambda x. \forall y \in set\ l. x \leq y)$

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,  $min\text{-of-list}$  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.

**definition**  $min\text{-of-list1} :: 'a::linorder\ list \Rightarrow 'a\ nres$   
**where**  $min\text{-of-list1}\ l \equiv ASSERT\ (l \neq []) \gg RETURN\ (fold\ min\ (tl\ l)\ (hd\ l))$

Note that  $RETURN$  returns exactly one (deterministic) result.

We have to show that our implementation actually refines the specification

**lemma**  $min\text{-of-list1-refine}: (min\text{-of-list1}, min\text{-of-list}) \in Id \rightarrow \langle Id \rangle nres\text{-rel}$

This lemma has to be read as follows: If the argument given to  $min\text{-of-list1}$  and  $min\text{-of-list}$  are related by  $Id$  (i.e. are identical), then the result of  $min\text{-of-list1}$  is a refinement of the result of  $min\text{-of-list}$ , wrt. relation  $Id$ .

For an explanation, let's simplify the statement first:

$\langle proof \rangle$

A more concise proof of the same lemma omits the initial simplification, which we only inserted to explain the refinement ordering:

**lemma**  $(min\text{-of-list1}, min\text{-of-list}) \in Id \rightarrow \langle Id \rangle nres\text{-rel}$   
 $\langle proof \rangle$



## 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 *hd* and *tl*.

**definition** *min-of-list2* :: 'a::linorder list  $\Rightarrow$  'a nres  
**where** *min-of-list2* l  $\equiv$  ASSERT (l $\neq$ [])  $\gg$  RETURN (fold ( $\lambda i. \text{min } (l(i+1))$ ) [0..*length* l - 1] (l 0))

Proving refinement is straightforward, using the *fold-idx-conv* lemma.

**lemma** *min-of-list2-refine*: (*min-of-list2*, *min-of-list1*) $\in$ Id  $\rightarrow$  (Id)nres-rel  
*<proof>*

## Imperative Algorithm

The version *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 *min-of-list3* of the algorithm for natural numbers, which uses an array instead of a list:

**sepref-definition** *min-of-list3* is *min-of-list2* :: (array-assn nat-assn)<sup>k</sup>  $\rightarrow_a$  nat-assn  
*<proof>*

The generated constant represents an Imperative/HOL program, and is executable:

**thm** *min-of-list3-def*  
**export-code** *min-of-list3* **checking** *SML-imp*

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 id-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*:  $(\text{min-of-list3}, \text{min-of-list}) \in (\text{array-assn nat-assn})^k \rightarrow_a \text{nat-assn}$   
 $\langle \text{proof} \rangle$

While the above statement is suited to re-use the algorithm within the sepre 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:

**lemma**  $l \neq [] \implies \langle \text{array-assn nat-assn } l \ a \rangle \text{min-of-list3 } a \ \langle \lambda x. \text{array-assn nat-assn } l \ a \ * \ \uparrow(\forall y \in \text{set } l. x \leq y) \rangle_t$

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:

$\langle \text{proof} \rangle$

If the automatic method does not work, here is a more explicit proof, that can be adapted for proving similar statements:

**lemma**  $l \neq [] \implies \langle \text{array-assn nat-assn } l \ a \rangle \text{min-of-list3 } a \ \langle \lambda x. \text{array-assn nat-assn } l \ a \ * \ \uparrow(\forall y \in \text{set } l. x \leq y) \rangle_t$   
 $\langle \text{proof} \rangle$

### 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, it's 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

```
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 232 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* ⟨*proof*⟩

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*<sup>k</sup> →<sub>a</sub> *nat-assn*  
⟨*proof*⟩

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*⟩

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*  $x\ xs \equiv \text{ASSERT } (\text{sorted } xs) \gg \text{RETURN } (x \in \text{set } xs)$

And give a standard iterative implementation:

**definition** *in-sorted-list1-invar*  $x\ xs \equiv \lambda(l,u,\text{found}).$   
 $(l \leq u \wedge u \leq \text{length } xs)$   
 $\wedge (\text{found} \longrightarrow x \in \text{set } xs)$   
 $\wedge (\neg \text{found} \longrightarrow (x \notin \text{set } (\text{take } l\ xs) \wedge x \notin \text{set } (\text{drop } u\ xs)))$   
 $)$

**definition** *in-sorted-list1*  $x\ xs \equiv \text{do } \{$   
 $\text{let } l=0;$   
 $\text{let } u=\text{length } xs;$   
 $(-,r) \leftarrow \text{WHILEIT } (\text{in-sorted-list1-invar } x\ xs)$   
 $(\lambda(l,u,\text{found}). l < u \wedge \neg \text{found}) (\lambda(l,u,\text{found}). \text{do } \{$   
 $\text{let } i = (l+u) \text{ div } 2;$   
 $\text{ASSERT } (i < \text{length } xs);$  — Added here to help synthesis to prove precondition  
for array indexing  
 $\text{let } xi = xs[i];$   
 $\text{if } x=xi \text{ then}$   
 $\text{RETURN } (l,u,\text{True})$   
 $\text{else if } x < xi \text{ then}$   
 $\text{RETURN } (l,i,\text{False})$   
 $\text{else}$   
 $\text{RETURN } (i+1,u,\text{False})$   
 $\}) (l,u,\text{False});$   
 $\text{RETURN } r$   
 $\}$

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  $(? \Phi \implies ?M \leq ?M') \implies ?M \leq \text{ASSERT } ? \Phi \gg (\lambda-. ?M')$ ).

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 *mop-list-get* (Look at it's simplified definition to get an impression what it does).

**thm** *mop-list-get-alt*

We first prove the refinement correct

**context begin**

**private lemma** *isl1-measure*: *wf (measure (λ(l,u,f). u-l + (if f then 0 else 1)))*

*<proof>* **lemma** *neq-nlt-is-gt*:

**fixes** *a b :: 'a::linorder*

**shows**  $a \neq b \implies \neg(a < b) \implies a > b$  *<proof>* **lemma** *isl1-aux1*:

**assumes** *sorted xs*

**assumes**  $i < \text{length } xs$

**assumes**  $xs!i < x$

**shows**  $x \notin \text{set } (\text{take } i \text{ } xs)$

*<proof>* **lemma** *isl1-aux2*:

**assumes**  $x \notin \text{set } (\text{take } n \text{ } xs)$

**shows**  $x \notin \text{set } (\text{drop } n \text{ } xs) \longleftrightarrow x \notin \text{set } xs$

*<proof>*

**lemma** *in-sorted-list1-refine*:  $(\text{in-sorted-list1}, \text{in-sorted-list}) \in \text{Id} \rightarrow \text{Id} \rightarrow \langle \text{Id} \rangle \text{nres-rel}$

*<proof>*

**end**

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 *uncurry* function, and we also provide abbreviations *uncurry2* up to  $\lambda f. \text{uncurry2 } (\text{uncurry2 } (\text{uncurry } f))$ . If a function has no parameters, *uncurry0* adds a unit parameter.

**sepref-definition** *in-sorted-list2* **is**  $\text{uncurry } \text{in-sorted-list1} :: \text{nat-assign}^k *_{\alpha} (\text{array-assign } \text{nat-assign})^k \rightarrow_{\alpha} \text{bool-assign}$

*<proof>*

**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 *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.

```
definition in-sorted-list1' x xs  $\equiv$  do {  
  let  $l=0$ ;  
  let  $u=\text{length } xs$ ;  
   $(-, -, r) \leftarrow \text{WHILEIT } (\text{in-sorted-list1-invar } x \text{ } xs)$   
   $(\lambda(l, u, \text{found}). l < u \wedge \neg \text{found}) (\lambda(l, u, \text{found}). \text{do } \{$   
    let  $i = (l+u) \text{ div } 2$ ;  
    let  $xi = xs[i]$ ; — It's not trivial to show that  $i$  is in range  
    if  $x=xi$  then  
      RETURN  $(l, u, \text{True})$   
    else if  $x < xi$  then  
      RETURN  $(l, i, \text{False})$   
    else  
      RETURN  $(i+1, u, \text{False})$   
   $\}) (l, u, \text{False});$   
  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-assnk *a (array-assn nat-assn)k →a bool-assn  
   $\langle \text{proof} \rangle$ 
```

## 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.  
   $\langle \text{proof} \rangle$ 
```

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  $\lambda x. RETURN (2+x) :: nat-assert^k \rightarrow_a nat-assert$   
   $\langle proof \rangle$ 
```

## Translation Phase

In most cases, the translation phase will fail. Let's try the following refinement:

```
sepref-thm test is  $\lambda l. RETURN (!1 + 2) :: (array-assert nat-assert)^k \rightarrow_a nat-assert$ 
```

The *sepref* method will just fail. To investigate further, we use *sepref-dbg-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.

```
 $\langle proof \rangle$ 
```

Inserting an assertion into the abstract program solves the problem:

```
sepref-thm test is  $\lambda l. ASSERT (length l > 1) \gg RETURN (!1 + 2) :: (array-assert$   
 $nat-assert)^k \rightarrow_a nat-assert$   
   $\langle proof \rangle$ 
```

Here is an example for an unimplemented operation:

```
sepref-thm test is  $\lambda l. RETURN (Min (set l)) :: (array-assert nat-assert)^k \rightarrow_a nat-assert$   
   $\langle proof \rangle$ 
```

### 4.1.5 The Isabelle Imperative Collection Framework (IICF)

The IICF provides a library of imperative data structures, and some management infrastructure. The main idea is to have interfaces and implementations.

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 standard 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*  $s \equiv$   
*ASSERT* (*finite*  $s$ )  $\gg$  *SPEC* ( $\lambda m. \text{dom } m = s \wedge \text{ran } m = \{0..<\text{card } s\}$ )

We implement the function by iterating over the set, and building the map

**definition** *nat-seg-map1*  $s \equiv \text{do } \{$   
  *ASSERT* (*finite*  $s$ );  
   $(m, -) \leftarrow \text{FOREACHi } (\lambda it (m, i). \text{dom } m = s - it \wedge \text{ran } m = \{0..<i\} \wedge i = \text{card } (s - it))$   
   $s (\lambda x (m, i). \text{RETURN } (m(x \mapsto i), i + 1)) (\text{Map.empty}, 0)$ ;  
  *RETURN*  $m$   
}

**lemma** *nat-seg-map1-refine*:  $(\text{nat-seg-map1}, \text{nat-seg-map}) \in \text{Id} \rightarrow \langle \text{Id} \rangle \text{nres-rel}$   
*<proof>*

We use hashsets *hs.assn* and hashmaps (*hm.assn*).

**sempref-definition** *nat-seg-map2* **is** *nat-seg-map1*  $:: (\text{hs.assn } \text{id-assn})^k \rightarrow_a \text{hm.assn}$   
*id-assn nat-assn*  
*<proof>*

Assignment of implementations to constructor operations is done by rewriting them to synonyms which are bound to a specific implementation. For hashmaps, we have *op-hm-empty*, and the rules *hm.fold-custom-empty*.

**sempref-definition** *nat-seg-map2* **is** *nat-seg-map1*  $:: (\text{hs.assn } \text{id-assn})^k \rightarrow_a \text{hm.assn}$   
*id-assn nat-assn*  
*<proof>*

**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*  $l \equiv \text{map } ((+) 1) l$

We might want to implement it as follows

**definition** *incr-list1*  $l \equiv \text{fold } (\lambda i l. l[i:=1 + !i]) [0..<\text{length } l] l$

**lemma** *incr-list1-refine*:  $(\text{incr-list1}, \text{incr-list}) \in \text{Id} \rightarrow \text{Id}$   
*<proof>*



Trying to refine this reveals a problem:

**sepref-thm** *incr-list2* is *RETURN o incr-list1* ::  $(array-assn\ nat-assn)^d \rightarrow_a array-assn\ nat-assn$   
*<proof>*

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 *nfoldli* combinator, which generalizes *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*  $l \equiv nfoldli$   
 $[0..<length\ l]$   
 $(\lambda-. True)$   
 $(\lambda i\ l. ASSERT\ (i < length\ l) \gg RETURN\ (l[i:=1+l!i]))$   
 $l$

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*:  $(incr-list2, RETURN\ o\ incr-list) \in Id \rightarrow \langle Id \rangle nres-rel$   
*<proof>*

**sepref-definition** *incr-list3* is *incr-list2* ::  $(array-assn\ nat-assn)^d \rightarrow_a array-assn\ nat-assn$   
*<proof>*

**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*  $(\lambda l\ i\ j. do\ \{$

```

  ASSERT (i < length l ∧ j < length l);
  RETURN (l[i:=!j, j:=!i])
}) :: (array-assn id-assn)d *a nat-assnk *a nat-assnk →a array-assn id-assn
⟨proof⟩

```

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 = !i;
  RETURN (l[i:=!j, j:=tmp])
}) :: (array-assn id-assn)d *a nat-assnk *a nat-assnk →a array-assn id-assn
⟨proof⟩

```

Note that also the argument must be marked as destroyed (<sup>d</sup>). 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 = !i;
  RETURN (l[i:=!j, j:=tmp])
}) :: (array-assn id-assn)k *a nat-assnk *a nat-assnk →a array-assn id-assn
⟨proof⟩

```

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 (<sup>k</sup>) annotation):

```

sepref-thm incr-list3-preserve is incr-list2 :: (array-assn nat-assn)k →a array-assn
nat-assn
⟨proof⟩

```

#### 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 *snd* would have to destroy the product, losing 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 *List-MsetO* and *List-SetO* (Here, the *O* stands for *own*, which means that the data-structure owns its elements.).

Works fine:

```
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-assnk →a bool-assn
⟨proof⟩
```

Fails: We cannot access compound type inside case distinction

```
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-assnk →a bool-assn
⟨proof⟩
```

Works fine, as components of product are pure, such that product can be restored inside case.

```
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)
}) :: unit-assnk →a bool-assn
⟨proof⟩
```

Trying to create a list of arrays, first attempt:

```
sepref-thm set-of-arrays-ex is uncurry0 (RETURN (op-list-append [] op-array-empty))
:: unit-assnk →a arl-assn (array-assn nat-assn)
⟨proof⟩
```

So lets choose a circular singly linked list (csll), which does not require its elements to be of default type class

```
sepref-thm set-of-arrays-ex is uncurry0 (RETURN (op-list-append [] op-array-empty))
:: unit-assnk →a csll.assn (array-assn nat-assn)
⟨proof⟩
```

Finally, there are a few data structures that already support nested element types, for example, functional lists:

```
sepref-thm set-of-arrays-ex is uncurry0 (RETURN (op-list-append [] op-array-empty))
:: unit-assnk →a list-assn (array-assn nat-assn)
⟨proof⟩
```

#### 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  $\{0..<N\}$ , and drop duplicate numbers. The following abstract algorithm may work:

```
definition remdup l  $\equiv$  do {  
  (s,r)  $\leftarrow$  nfoldli l ( $\lambda$ -. True)  
  ( $\lambda$ x (s,r). do {  
    ASSERT (distinct r  $\wedge$  set r  $\subseteq$  set l  $\wedge$  s = set r); — Will be required to prove  
    that list does not grow too long  
    if x  $\in$  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.

**sempref-register** *remdup*

The straightforward version with dynamic data-structures is:

```
sempref-definition remdup1 is remdup :: (list-assn nat-assn)k  $\rightarrow_a$  arl-assn nat-assn  
   $\langle$ proof $\rangle$ 
```

## Initialization 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.

**context** *fixes* *N* :: *nat* **begin**

```
sempref-definition remdup1-init is remdup :: (list-assn nat-assn)k  $\rightarrow_a$  arl-assn  
nat-assn  
   $\langle$ proof $\rangle$ 
```

**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-init* (*N*::*nat*)  $\equiv$  *remdup*

**lemma** *remdup-init-hnr*:

```
(uncurry remdup1-init, uncurry remdup-init)  $\in$  nat-assnk *a (list-assn nat-assn)k  
 $\rightarrow_a$  arl-assn nat-assn  
 $\langle$ proof $\rangle$ 
```

## 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  $N$   $\langle proof \rangle$ 
```

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]:  $(N, N) \in nat\text{-rel}$   $\langle proof \rangle$ 
thm N-hnr
```

Alternatively, we could directly prove the following rule, which, however, is more cumbersome:

```
lemma N-hnr':  $(uncurry0 (return\ N), uncurry0 (RETURN\ N)) \in unit\text{-assn}^k \rightarrow_a$ 
 $nat\text{-assn}$ 
 $\langle proof \rangle$ 
```

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 ::  $(list\text{-assn}\ nat\text{-assn})^k \rightarrow_a\ marl\text{-assn}$ 
 $N\ nat\text{-assn}$ 
 $\langle proof \rangle$ 
```

Moreover, we add a precondition on the list

```
sepref-definition remdup1-fixed is remdup ::  $[\lambda l. set\ l \subseteq \{0..<N\}]_a\ (list\text{-assn}$ 
 $nat\text{-assn})^k \rightarrow\ marl\text{-assn}\ N\ nat\text{-assn}$ 
 $\langle proof \rangle$ 
```

We can prove the remaining subgoal, e.g., by *auto* with the following lemma declared as introduction rule:

```
lemma aux1[intro]:  $\llbracket set\ l \subseteq \{0..<N\};\ distinct\ l \rrbracket \implies length\ l < N$ 
 $\langle proof \rangle$ 
```

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
→ marl-assn N nat-assn
  ⟨proof⟩
```

```
concrete-definition (in -) remdup1-fixed uses my-remdup-impl-loc.remdup1-fixed.refine-raw
is (?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-assnk →a nat-assn
```

```
  ⟨proof⟩
```

```
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*:  $(test-remdup1, my-remdup-impl-loc.test-remdup) \in [my-remdup-impl-loc]_a \text{ nat-assn}^k \rightarrow \text{nat-assn}$   
*<proof>*

We can also write a more direct precondition, as long as it implies the locale

**lemma** *test-remdup1-refine*:  $(test-remdup1, my-remdup-impl-loc.test-remdup) \in [\lambda N. N > 0]_a \text{ nat-assn}^k \rightarrow \text{nat-assn}$   
*<proof>*

**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** [*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, *b-assn id-assn*  $(\lambda x. x < N)$  relates natural numbers less than *N* to themselves.

We will repeat the above development, using the bounded relation approach:

**definition** *bremdup l*  $\equiv$  *do* {  
   $(s,r) \leftarrow \text{nfoldli } l (\lambda-. \text{True})$   
   $(\lambda x (s,r). \text{do}$  {  
    *ASSERT* (*distinct r*  $\wedge$  *s = set r*); — Less assertions than last time  
    *if*  $x \in s$  *then RETURN* (*s,r*) *else RETURN* (*insert x s, r@[x]*)  
  })  
   $(\{\}, [])$ ;  
  *RETURN r*  
}

**sepref-register** *bremdup*

**locale** *my-bremdup-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**

**lemma** *my-bremdup-impl-loc-this*: *my-bremdup-impl-loc N* *<proof>*

**sepref-register** *N*

**lemma** *N-hnr*[*sepref-import-param*]:  $(N,N) \in \text{nat-rel}$  *<proof>*

Conceptually, what we insert in our list are elements, and these are less than  $N$ .

**abbreviation**  $elem-assn \equiv nbn-assn\ N$

**lemma**  $aux1[intro]: \llbracket set\ l \subseteq \{0..<N\};\ distinct\ l \rrbracket \implies length\ l < N$   
 $\langle proof \rangle$

**sepref-thm**  $remdup1-fixed\ is\ remdup :: [\lambda l. set\ l \subseteq \{0..<N\}]_a (list-assn\ elem-assn)^k$   
 $\rightarrow\ marl-assn\ N\ elem-assn$   
 $\langle proof \rangle$

**concrete-definition** (in  $-$ )  $bremdup1-fixed$  uses  $my-bremdup-impl-loc.remdup1-fixed.refine-raw$   
is  $(?f,-)\in-$

**prepare-code-thms** (in  $-$ )  $bremdup1-fixed-def$

**lemmas**  $remdup1-fixed-refine[sepref-fr-rules] = bremdup1-fixed.refine[OF\ my-bremdup-impl-loc-this]$

**definition**  $test-remdup \equiv do\ \{l \leftarrow remdup\ [0..<N];\ RETURN\ (length\ l)\}$   
**sepref-register**  $test-remdup$

This refinement depends on the (somewhat experimental) subtyping feature to convert from  $id-assn$  to  $b-assn\ id-assn\ (\lambda x. x < N)$ , based on context information

**sepref-thm**  $test-remdup1\ is$   
 $uncurry0\ (PR-CONST\ test-remdup) :: unit-assn^k \rightarrow_a\ nat-assn$   
 $\langle proof \rangle$

**concrete-definition** (in  $-$ )  $test-bremdup1$  uses  $my-bremdup-impl-loc.test-remdup1.refine-raw$   
is  $(uncurry0\ ?f,-)\in-$

**prepare-code-thms** (in  $-$ )  $test-bremdup1-def$

**lemmas**  $test-remdup1-refine[sepref-fr-rules] = test-bremdup1.refine[of\ N]$

**end**

**lemma**  $test-bremdup1-refine-aux: (test-bremdup1, my-bremdup-impl-loc.test-remdup)$   
 $\in [my-bremdup-impl-loc]_a\ nat-assn^k \rightarrow nat-assn$   
 $\langle proof \rangle$

**lemma**  $test-bremdup1-refine: (test-bremdup1, my-bremdup-impl-loc.test-remdup) \in$   
 $[\lambda N. N > 0]_a\ nat-assn^k \rightarrow nat-assn$   
 $\langle proof \rangle$

**export-code**  $test-bremdup1\ checking\ SML$

We can also register the abstract constant and the refinement, to use it in further refinements

**sepref-register**  $test-bremdup: my-bremdup-impl-loc.test-remdup$  — Specifying a base-name for the theorems here, as default name clashes with existing names.



**lemmas** [*sepref-fr-rules*] = *test-bremdup1-refine*

## Fixed-Value Restriction

Initialization only works with fixed values, not with dynamically computed values

**sepref-definition** *copy-list-to-array* is  $\lambda l$ . do {  
*let*  $N = \text{length } l$ ; — Introduce a *let*, such that we have a single variable as size-init  
*let*  $l' = \text{op-arl-empty-sz } N$ ;  
*nfoldli*  $l (\lambda x. \text{True}) (\lambda x s. \text{mop-list-append } s x) l'$   
} ::  $(\text{list-assn nat-assn})^k \rightarrow_a \text{arl-assn nat-assn}$   
*<proof>*

## 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  $'a \text{ mtx}$  (which is  $\text{nat} \times \text{nat} \Rightarrow 'a$ ) as the abstract representation. The (currently only) implementation is by arrays, mapping points at coordinates out of range to 0.

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** *mtx-dup-diag-f*::  $\text{nat} \times \text{nat} \Rightarrow 'a :: \{\text{numeral}, \text{times}, \text{mult-zero}\} \Rightarrow 'a$   
**where** *mtx-dup-diag-f*  $\equiv \lambda(i,j) x. \text{if } i=j \text{ then } x*(2) \text{ else } x$

We refine this function to a heap-function, using the identity mapping for values.

### context

**fixes** *dummy* ::  $'a :: \{\text{numeral}, \text{times}, \text{mult-zero}\}$

**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  $(*) :: 'a \Rightarrow -, \text{folded fun-rel-id-simp}$ ]

### begin

**sepref-definition** *mtx-dup-diag-f1* is *uncurry* (*RETURN* oo (*mtx-dup-diag-f*:: $\Rightarrow 'a \Rightarrow -$ ))  
::  $(\text{prod-assn nat-assn nat-assn})^k *_a \text{id-assn}^k \rightarrow_a \text{id-assn}$   
*<proof>*

### 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*  
 ⟨*proof*⟩  
**applyS** (*simp add: mtx-dup-diag-f-def*) []  
**applyS** (*rule mtx-dup-diag-f1.refine*)  
 ⟨*proof*⟩

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*  $\equiv$  *mtx-pointwise-unop* *mtx-dup-diag-f*

The operation is usable now:

**sepref-thm** *mtx-dup-test* **is**  $\lambda m. RETURN (mtx-dup-diag (mtx-dup-diag m)) :: (asm\tx-assn\ N\ int-assn)^d \rightarrow_a\ asm\tx-assn\ N\ int-assn$   
 ⟨*proof*⟩

Similarly, there are operations to combine to matrices, and to compare two matrices:

**interpretation** *pw-add*: *amtx-pointwise-binop-impl*  $N$   $M$   $((+))::(-::monoid-add) \Rightarrow -$  *id-assn* *return* *oo*  $((+))$   
**for**  $N$   $M$   
 ⟨*proof*⟩

**abbreviation** *mtx-add*  $\equiv$  *mtx-pointwise-binop*  $((+))$

**sepref-thm** *mtx-add-test* **is** *uncurry2*  $(\lambda m1\ m2\ m3. RETURN (mtx-add\ m1\ (mtx-add\ m2\ m3)))$   
 $:: (asm\tx-assn\ N\ M\ int-assn)^d *_a (asm\tx-assn\ N\ M\ int-assn)^d *_a (asm\tx-assn\ N\ M\ int-assn)^k \rightarrow_a\ asm\tx-assn\ N\ M\ int-assn$   
 ⟨*proof*⟩

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)

**sepref-thm** *mtx-dup-alt-test* **is**  $(\lambda m. RETURN (mtx-add\ m\ m))$   
 $:: (asm\tx-assn\ N\ M\ int-assn)^d \rightarrow_a\ asm\tx-assn\ N\ M\ int-assn$   
 ⟨*proof*⟩

Of course, you can always copy the matrix manually:

**sepref-thm** *mtx-dup-alt-test* **is**  $(\lambda m. RETURN (mtx-add (op-mtx-copy\ m)\ m))$   
 $:: (asm\tx-assn\ N\ M\ int-assn)^k \rightarrow_a\ asm\tx-assn\ N\ M\ int-assn$   
 ⟨*proof*⟩

A compare operation checks that all pairs of entries fulfill some property *f*, and at least one entry fullfills a property *g*.

**interpretation** *pw-lt*: *amtx-pointwise-cmpop-impl*  $N M ((\leq)::(-::order) \Rightarrow -) ((\neq)::(-::order) \Rightarrow -)$  *id-assn return oo*  $(\leq)$  *return oo*  $(\neq)$   
**for**  $N M$   
 $\langle proof \rangle$   
**abbreviation** *mtx-lt*  $\equiv$  *mtx-pointwise-cmpop*  $(\leq)$   $(\neq)$

**sepref-thm** *test-mtx-cmp* **is**  $(\lambda m. do \{ RETURN (mtx-lt (op-amtx-dftNxM N M 0) m) \}) :: (amtx-assn N M int-assn)^k \rightarrow_a bool-assn$   
 $\langle proof \rangle$

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**  $N M :: nat$   
**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*  $\equiv$  *prod-assn*  $(nbn-assn N)$   $(nbn-assn M)$

And one for integer matrices

**abbreviation** *mtx-assn*  $\equiv$  *amtx-assn*  $N M int-assn$

**definition** *co-set-gen*  $\equiv$   $do \{$   
 $ifoldli [0..<N] (\lambda-. True) (\lambda i. ifoldli [0..<M] (\lambda-. True) (\lambda j s.$   
 $if \max i j - \min i j \leq 1$  then  $RETURN (insert (i,j) s)$   
 $else RETURN s$   
 $) \}$   
 $\}$

**sepref-definition** *co-set-gen1* **is** *uncurry0* *co-set-gen*  $:: unit-assn^k \rightarrow_a hs.assn$   
*co-assn*  
 $\langle proof \rangle$

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*  $(PR-CONST co-set-gen) :: unit-assn^k \rightarrow_a hs.assn co-assn$   
 $\langle proof \rangle$   
**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 ( $\lambda m.$  do {
  co  $\leftarrow$  co-set-gen;
  FOREACH co ( $\lambda(i,j)$  m. RETURN (m((i,j) := 1))) m
}) :: mtx-assnd  $\rightarrow_a$  mtx-assn
  <proof>

```

**end**

#### 4.1.10 Type Classes

TBD

#### 4.1.11 Higher-Order

TBD

#### 4.1.12 A-Posteriori Optimizations

The theorem collection *sepref-opt-simps* and *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.

**thm** *sepref-opt-simps*

**thm** *sepref-opt-simps2*

#### 4.1.13 Short-Circuit Evaluation

Consider

```

sepref-thm test-sc-eval is RETURN o ( $\lambda l.$  length l > 0  $\wedge$  hd l) :: (list-assn
bool-assn)k  $\rightarrow_a$  bool-assn
  <proof>

```

```

sepref-thm test-sc-eval is RETURN o ( $\lambda l.$  length l > 0  $\wedge$  hd l) :: (list-assn
bool-assn)k  $\rightarrow_a$  bool-assn
  <proof>

```

**end**

## 4.2 Reference Guide

**theory** *Sepref-Guide-Reference*

```

imports ../HICF/HICF
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 *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 *hn-ctxt* tag.

Alternatively, a term of the form  $(\ ?c, f) \in [P]_a \ A \rightarrow R$  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) })
  <proof>

```

The above command asks Sepref to synthesize a program, in a heap context where there is a natural number, refined by *nat-assn*, and a list of booleans, refined by *array-assn bool-assn*. The *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 ( $\lambda x \ a. \ do \ \{ \text{ASSERT} \ (x < \text{length} \ a); \ \text{RETURN} \ (a!x) \} \}))
  \in \ \text{nat-assn}^k \ *_{\ a} \ (\text{array-assn} \ \text{bool-assn})^k \ \rightarrow_{\ a} \ \text{bool-assn}
  <proof>$ 
```

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.

**thm** *intf-of-assn*

The *sepref*-method is split into various phases, which we will explain now

### 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.

**thm** *sepref-preproc*

**thm** *intf-of-assn*

**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!

### 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.

### Operation Identification Phase

The purpose of this phase is to identify the conceptual operations in the given program. Consider, for example, a map  $m::'k \Rightarrow 'v \text{ option}$ . If one writes  $m(k \mapsto v)$ , this is a map update. However, in Isabelle/HOL maps are encoded as functions  $'k \Rightarrow 'v \text{ option}$ , and the map update is just syntactic sugar for *fun-upd m k (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 *Some x*. Shall *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. *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 *id-rules*. Moreover, it stores two sets of rewrite rules, in *pat-rules* and *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 *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 *pat-rules* are backtracked over, while rewrite rules from *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  $(\ 'k, \ 'v) \ i\text{-map}$ . The *id-rule* for map lookup is *op-map-lookup*  $::_i \text{TYPE}(\ 'a \Rightarrow (\ 'a, \ 'b) \ i\text{-map} \Rightarrow \ 'b \ \text{option})$ . Moreover, there is a rule to rewrite function application to map lookup  $( \ ?m \ \$ \ ?k \equiv \ \text{op-map-lookup} \ \$' \ ?k \ \$' \ ?m)$ . It can be backtracked over, such that also functions into the option type are possible.

**thm** *op-map-lookup.itype*

**thm** *pat-map-lookup*

**thm** *id-rules*

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  $\lambda x. (\#t \ x\#)$  (syntax:  $\lambda x. (\#t \ x\#)$ , input syntax:  $\lambda x. (\#t \ x\#)$ ). This is required to tame Isabelle's higher-order unification. However, it makes tagged terms quite

unreadable, and it may be helpful to *unfold 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  $t :::i \text{ TYPE}(a)$ ).

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 *PR-CONST* tag, which causes them to be treated like a single constant. Such constants must always have *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 *id-rule* for a free variable or complex *PR-CONST* constant, or the identification rule is malformed. Note that, in practice, identification rules are registered by the **sepref-register** (see below), which catches many malformed rules, and handles *PR-CONST* 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.,  $'k \Rightarrow 'v \text{ option}$  with  $('k, 'v) \text{ i-map}$ .

The method *sepref-dbg-id* invokes the id-phase in isolation. The method *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 **back**.

The methods *sepref-dbg-id-init*, *sepref-dbg-id-step*, and *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
   $\langle \textit{proof} \rangle$ 

```



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 $k$ 
→ $a$  nat-assn ⟨proof⟩
end
end

```

**definition** *my-map* ≡ *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→1))) :: unit-assn $k$ 
→ $a$  hm.assn bool-assn nat-assn
  ⟨proof⟩

```

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→1))) :: unit-assn $k$ 
→ $a$  hm.assn bool-assn nat-assn
  ⟨proof⟩

```

## 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<sup>1</sup>.

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

---

<sup>1</sup>Using fractional permissions or some other more fine grained ownership model might lift this restriction in the future.

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 ( $fold \equiv \lambda x. (\# \lambda xa. (\# \lambda xb. (\# SP \text{ fold } \$ (\lambda xa. (\# \lambda xb. (\# x \$ xa \$ xb\#)\#)) \$ xa \$ xb\#)\#)\#)$ ).

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.

**thm** *sepref-monadify-arity*

## 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.  $EVAL \$ (If \$ ?b \$ ?t \$ ?e) \equiv (\gg) \$ (EVAL \$ ?b) \$ (\lambda x. (\# If \$ x \$ (EVAL \$ ?t) \$ (EVAL \$ ?e)\#))$ )

$EVAL \$ (case-list \$ ?fn \$ (\lambda x. (\# \lambda xa. (\# ?fc x xa\#)\#)) \$ ?l) \equiv (\gg) \$ (EVAL \$ ?l) \$ (\lambda x. (\# case-list \$ (EVAL \$ ?fn) \$ (\lambda x. (\# \lambda xa. (\# EVAL \$ ?fc x xa\#)\#)) \$ x\#)$ )

$EVAL \$ (case-prod \$ (\lambda x. (\# \lambda xa. (\# ?fp x xa\#)\#)) \$ ?p) \equiv (\gg) \$ (EVAL \$ ?p) \$ (\lambda x. (\# case-prod \$ (\lambda x. (\# \lambda xa. (\# EVAL \$ ?fp x xa\#)\#)) \$ x\#)$ )

$EVAL \$ (case-option \$ ?fn \$ (\lambda x. (\# ?fs x\#)) \$ ?ov) \equiv (\gg) \$ (EVAL \$ ?ov) \$ (\lambda x. (\# case-option \$ (EVAL \$ ?fn) \$ (\lambda x. (\# EVAL \$ ?fs x\#)) \$ x\#)$ )

$EVAL \$ (Let \$ ?v \$ (\lambda x. (\# ?f x\#))) \equiv (\gg) \$ (EVAL \$ ?v) \$ (\lambda x. (\# EVAL \$ ?f x\#))$ ). 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-dbg-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-dbg-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*

**sepref-thm** *my-fold-test* is  $\lambda l. do \{ RETURN (my-fold (\lambda x y. x+y*2) l 0) \} :: (list-assn \text{ nat-assn})^k \rightarrow_a \text{ nat-assn}$   
 $\langle proof \rangle$

Solution: Register appropriate arity and combinator-rules

**lemma** *my-fold-arity*[*sepref-monadify-arity*]: *my-fold*  $\equiv$   $\lambda_2 f l s. SP \text{ my-fold} (\lambda_2 x s. f \$x \$s) \$l \$s$   $\langle proof \rangle$

The combinator-rule rewrites to the already existing and set up combinator *nfoldli*:

**lemma** *monadify-plain-my-fold*[*sepref-monadify-comb*]:

$EVAL \$ (my-fold \$ (\lambda_2 x s. f x s) \$l \$s) \equiv (\gg) \$ (EVAL \$ l) \$ (\lambda_2 l. (\gg) \$ (EVAL \$ s) \$ (\lambda_2 s. nfoldli \$l \$ (\lambda_2 -. True) \$ (\lambda_2 x s. EVAL \$ (f x s) \$s)))$   
 $\langle proof \rangle$

**sepref-thm** *my-fold-test* is  $\lambda l. do \{ RETURN (my-fold (\lambda x y. x+y*2) l 0) \} :: (list-assn \text{ nat-assn})^k \rightarrow_a \text{ nat-assn}$   
 $\langle proof \rangle$

### 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  $i = \text{length } l$ ;  
  RETURN (fold ( $\lambda x.$  (+) ( $x*x$ ))  $l$   $i$ )  
} :: (list-assn nat-assn)k  $\rightarrow_a$  nat-assn  
(proof)
```

## Optimization Init Phase

This phase, accessed by *sepref-dbg-opt-init*, just applies the rule  $\llbracket hn-refine \ ?\Gamma \ ?c \ ?\Gamma' \ ?R \ ?a; CNV \ ?c \ ?c' \rrbracket \Longrightarrow hn-refine \ ?\Gamma \ ?c' \ ?\Gamma' \ ?R \ ?a$  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 *hn-ctxt*  $A \ x \ xi$ , indicating how this variable is implemented. Common abbreviations are *hn-val*  $R \ x \ xi \equiv hn-val \ R \ x \ xi$  and *hn-invalid*  $A \ x \ xi \equiv hn-invalid \ A \ x \ xi$ .

## 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 *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 *hn-invalid*  $A x xi$ , 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 *sepref-fr-rules* to the subgoal. If there is no matching rule, matching is retried after rewriting the

precondition with *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 *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 *sepref-dbg-trans-step* performs a single translation step. The method *sepref-dbg-trans-step-keep* presents the internal goal state on failure. If it fails in the *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** ( $-\vee_A-\Longrightarrow_t-$ ). 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** ( $-\Longrightarrow_t-$ ). 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** (*INDEP* ( $?R\ x_1 \dots x_n$ )). 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** (*CONSTRAINT* - -) Apply solver for deferrable constraints (see section on constraints).

**Monotonicity** (*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** (*PREFER-tag* -/*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** *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 algorithm, 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 assertions. 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 condition solver encounters a constraint side condition, it first removes the constraint tag ( $?P \ ?x \Longrightarrow \text{CONSTRAINT } ?P \ ?x$ ) 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 backtracking).

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 preprocessing 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  $\Gamma \Longrightarrow_t \Gamma'$ . 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  $?P \Longrightarrow_t ?P$

$?F \Longrightarrow_t ?F' \Longrightarrow ?F * \text{hn-ctxt } ?A \ ?x \ ?y \Longrightarrow_t ?F' * \text{hn-ctxt } ?A \ ?x \ ?y$

$?F \Longrightarrow_t ?F' \Longrightarrow ?F * \text{hn-ctxt } ?A \ ?x \ ?y \Longrightarrow_t ?F'$

$?P \Longrightarrow_t \text{emp}$ . The matching pairs are resolved by repeatedly applying rules from  $?P \Longrightarrow_t ?P$

$\llbracket ?P \Longrightarrow_t ?P'; ?F \Longrightarrow_t ?F' \rrbracket \Longrightarrow ?F * ?P \Longrightarrow_t ?F' * ?P'$

$\text{hn-ctxt } ?R \ ?x \ ?y \Longrightarrow_t \text{hn-invalid } ?R \ ?x \ ?y$

$\text{hn-ctxt } ?R \ ?x \ ?y \Longrightarrow_t \text{hn-ctxt } (\lambda- \ . \ \text{true}) \ ?x \ ?y$

*CONSTRAINT is-pure*  $?R \Longrightarrow \text{hn-invalid } ?R \ ?x \ ?y \Longrightarrow_t \text{hn-ctxt } ?R \ ?x \ ?y$

and *sepref-frame-match-rules*. Any non-frame premise of these rules must be solved immediately by the side-condition's constraint or fallback tactic (see above). The tool backtracks over rules. If no rule matches (or side-conditions cannot be solved), it simplifies the goal with *sepref-frame-normrel-eqs* and tries again.

For merge rules, the theorems  $?F \vee_A ?F' \Longrightarrow_t ?F$

$\llbracket \text{hn-ctxt } ?R1.0 \ ?x \ ?x' \vee_A \text{hn-ctxt } ?R2.0 \ ?x \ ?x' \Longrightarrow_t \text{hn-ctxt } ?R \ ?x \ ?x'; ?Fl \vee_A ?Fr \Longrightarrow_t ?F \rrbracket \Longrightarrow ?Fl * \text{hn-ctxt } ?R1.0 \ ?x \ ?x' \vee_A ?Fr * \text{hn-ctxt } ?R2.0 \ ?x \ ?x' \Longrightarrow_t ?F * \text{hn-ctxt } ?R \ ?x \ ?x'$

$\text{hn-invalid } ?R \ ?x \ ?x' \vee_A \text{hn-ctxt } ?R \ ?x \ ?x' \Longrightarrow_t \text{hn-invalid } ?R \ ?x \ ?x'$

$\text{hn-ctxt } ?R \ ?x \ ?x' \vee_A \text{hn-invalid } ?R \ ?x \ ?x' \Longrightarrow_t \text{hn-invalid } ?R \ ?x \ ?x'$  and *sepref-frame-merge-rules* are used.



Note that a smart setup of frame and match rules together with side conditions makes the frame matcher a powerful tool for encoding structural and semantic information into relations. An example for structural information are the match rules for lists, which forward matching of list assertions to matching of the element assertions, maintaining the congruence assumption that the refined elements are actually elements of the list:  $(\bigwedge x x'. \llbracket x \in \text{set } ?l; x' \in \text{set } ?l \rrbracket \Longrightarrow \text{hn-ctxt } ?A \ x \ x' \Longrightarrow_t \text{hn-ctxt } ?A' \ x \ x') \Longrightarrow \text{hn-ctxt } (\text{list-assn } ?A) \ ?l \ ?l' \Longrightarrow_t \text{hn-ctxt } (\text{list-assn } ?A') \ ?l \ ?l'$ . An example for semantic information is the bounded assertion, which intersects any given assertion with a predicate on the abstract domain. The frame matcher is set up such that it can convert between bounded assertions, generating semantic side conditions to discharge implications between bounds  $(\llbracket \text{hn-ctxt } (b\text{-assn } ?A \ ?P) \ ?x \ ?y \Longrightarrow_t \text{hn-ctxt } ?A' \ ?x \ ?y; \llbracket \text{vassn-tag } (\text{hn-ctxt } ?A \ ?x \ ?y); \text{vassn-tag } (\text{hn-ctxt } ?A' \ ?x \ ?y); ?P \ ?x \rrbracket \Longrightarrow ?P' \ ?x \rrbracket \Longrightarrow \text{hn-ctxt } (b\text{-assn } ?A \ ?P) \ ?x \ ?y \Longrightarrow_t \text{hn-ctxt } (b\text{-assn } ?A' \ ?P') \ ?x \ ?y$ ).

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 *id-assn* restricted to indices that are in range:  $b\text{-assn } id\text{-assn } (\lambda x. x < N)$ . When inserting natural numbers into this list, one has to prove that they are actually in range (conversion from *id-assn* to  $\lambda n. b\text{-assn } id\text{-assn } (\lambda x. x < n)$ ). Elements of the list can be used as natural numbers (conversion from  $\lambda n. b\text{-assn } id\text{-assn } (\lambda x. x < n)$  to *id-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**  $[\text{sepref-register-adhoc } N]$

**notes**  $[\text{sepref-import-param}] = IdI[\text{of } N]$

**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  $\lambda l. \text{RETURN } (\text{fold } (\lambda x s. \text{if } x < N \text{ then insert } (\text{ASSN-ANNOT } (\text{nbN-assn } N) x) s \text{ else } s) l \text{ op-hs-empty}) :: (\text{list-assn nat-assn})^k \rightarrow_a \text{hs.assn } (\text{nbN-assn } N)$

*<proof>*

**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  $\lambda n. \text{do } \{ \text{let } r = \text{fold } (+) [1..<n] 0; \text{RETURN } (n*n+2) \}$   
 $:: \text{nat-assn}^k \rightarrow_a \text{nat-assn}$   
*<proof>*

## 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  $P \Longrightarrow \text{hn-refine } \Gamma c$

$\Gamma' R a$  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  $hn-ctxt A_1 x_1 xi_1 * \dots * hn-ctxt A_n x_n xi_n$
- $\Gamma'$  must have the form  $hn-ctxt B_1 x_1 xi_1 * \dots * hn-ctxt B_n x_n xi_n$  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  $hfref$  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  $\lambda f. uncurry2 (uncurry2 (uncurry f))$ ). The general form is  $PC \implies (uncurry_x f, uncurry_x g) \in [P]_a A_1^{k_1} *_a \dots *_a A_n^{k_n} \rightarrow R$ , where  $k_i$  is  $k$  if the argument is preserved (kept) or  $d$  if 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  $RETURN o \dots o g'$ , 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  $CONSTRAINT\ is-pure\ A \implies (uncurry2\ arl-swap, uncurry2 (RETURN\ \circ\circ\circ\ op-list-swap)) \in [\lambda((l, i), j). i < length\ l \wedge j < length\ l]_a (arl-assn\ A)^d *_a id-assn^k *_a id-assn^k \rightarrow arl-assn\ A$

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**  $arl-swap-hnr[unfolded\ pre-list-swap-def, no-vars]$

### 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** *to-hnr-post*

**thm** *to-hfref-post*

## 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 their corresponding assertions, e.g.,  $pure (\langle ?R \rangle list-rel) = list-assn (pure ?R)$  converts a list relation to a list assertion.

For debugging purposes, the attribute *sepref-dbg-import-rl-only* converts a parametricity theorem to a hnr-theorem. This is the first step of the standard conversion, followed by a conversion to hfref form.

**thm** *sepref-import-rewrite*

**thm** *param-append* — Parametricity theorem for append

**thm** *param-append[sepref-param]* — Converted to hfref-form. *list-rel* is rewritten to *list-assn*, and the relation variable is replaced by an assertion variable and a *is-pure* constraint.

**thm** *param-append*[*sepref-dbg-import-rl-only*]

For re-using Lifting/Transfer style theorems, the constants *p2rel* and *rel2p* may be helpful, however, there is no automation available yet.

Usage examples can be found in, e.g., *Refine-Imperative-HOL.IICF-Multiset*, where we import parametricity lemmas for multisets from the Lifting/Transfer package.

**thm** *p2rel* — Simp rules to convert predicate to relational style

**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., *hd* is only parametric for non-empty lists.

As of Isabelle2016, the Lifting/Transfer package cannot specify such preconditions at all.

Autoref's parametricity tool can specify such preconditions by using first-order rules, (cf.  $\llbracket ?l \neq []; (?l', ?l) \in \langle ?A \rangle \text{list-rel} \rrbracket \implies (\text{hd } ?l', \text{hd } ?l) \in ?A$ ). However, currently, *sepref-import-param* cannot handle these first-order rules.

Instead, Sepref supports the fref-format for parametricity rules, which resembles the hfref-format: Abstract and concrete objects are functions with exactly one parameter, uncurried if necessary. Moreover, there is an explicit precondition. The syntax is  $(\text{uncurry}_x f, \text{uncurry}_x g) \in [P]_f (\dots (R_1 \times_r R_2) \times_r \dots) \times_r R_n \rightarrow R$ , and without precondition, we have  $(\dots (R_1 \times_r R_2) \times_r \dots) \times_r R_n \rightarrow_f R$ . Note the left-bracketing 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 *to-fref* can convert (higher-order form) parametricity theorems to the fref-form.

#### Composition of hfref and fref theorems

fref and hfref 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 hfref with an fref theorem, yielding a hfref 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 hfref theorems.

The attribute *FCOMP* does these compositions and normalizes the result. Normalization consists of precondition simplification, and distributing com-

position 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 href/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*.

**thm** *fcomp-norm-unfold*

**thm** *fcomp-norm-simps*

**thm** *array-get-hnr-aux* — Array indexing, array elements are refined by identity

**thm** *op-list-get.fref* — Parametricity theorem for list indexing

**thm** *array-get-hnr-aux*[*FCOMP op-list-get.fref*] — Composed theorem

— Note the definition *array-assn ?A*  $\equiv$  *hr-comp is-array ((the-pure ?A)list-rel)*

**context**

**notes** [*fcomp-norm-unfold*] = *array-assn-def*[*symmetric*]

**begin**

**thm** *array-get-hnr-aux*[*FCOMP op-list-get.fref*] — Composed theorem, *array-assn* folded.

**end**

#### 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  $(\text{'}k, \text{'}) i\text{-map}$  represents maps, which are encoded as  $\text{'}k \Rightarrow \text{'}) option$  in HOL.

New interface types must be registered by the command **sepref-decl-intf**.

**sepref-decl-intf**  $(\text{'}a, \text{'}) i\text{-my-intf}$  **is**  $\text{'}a * \text{'}a \Rightarrow \text{'}) option$

— Declares  $(\text{'}a, \text{'}) i\text{-my-intf}$  as new interface type, and registers it to correspond to  $\text{'}a \times \text{'}a \Rightarrow \text{'}) option$ . Note: For HOL, the interface type is just an arbitrary new type, which is not related to the corresponding HOL type.

**sepref-decl-intf**  $(\text{'}a, \text{'}) i\text{-my-intf}2$  (**infix**  $\langle * \rightarrow_i \rangle 0$ ) **is**  $\text{'}a * \text{'}a \Rightarrow \text{'}) option$

— There is also a version that declares infix-syntax for the interface type. In this case we have  $\text{'}a * \rightarrow_i \text{'})$ .  $\text{'}a \rightarrow \text{'})$  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 *sepref-register* tool. Currently, it only works for operations, not for combinators.

The **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 *map-type-eqs*. This rewriting is useful for bulk-setup of many constants with conceptual types different from their 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 *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 (*- nres*), also an arity and combinator rule for the monadify phase are generated.

There is also an attribute version *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

```
definition map-op1 m n  $\equiv$  m( $n \rightarrow n+1$ )
definition map-op2 m n  $\equiv$  m( $n \rightarrow n+2$ )
definition map-op3 m n  $\equiv$  m( $n \rightarrow n+3$ )
definition map-op-to-map (m:: $'a \rightarrow 'b$ )  $\equiv$  m
```

**context**

```
  notes [map-type-eqs] = map-type-eqI[of TYPE( $'a \rightarrow 'b$ ) TYPE(( $'a, 'b$ )i-map)]
```

**begin**

```
  sepref-register map-op1 map-op2 map-op3
```

```
  — Registered interface types use i-map
```

```
  sepref-register map-op-to-map :: ( $'a \rightarrow 'b$ )  $\Rightarrow$  ( $'a, 'b$ ) i-map
```

```
  — Explicit type annotation is not rewritten
```

**end**

Example for insertion of *PR-CONST* tag and attribute-version

**context**

```
  fixes N :: nat and D :: int
```

```
  notes [[sepref-register-adhoc N D]]
```

```
  — In order to use N and D as operators (constant) inside this context, they
```

have to be registered. However, issuing a *sepref-register* command inside the context would export meaningless registrations to the global theory.

**notes** [*sepref-import-param*] = *IdI*[of *N*] *IdI*[of *D*]

- For declaring refinement rules, the *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 *sepref-fr-rules* on the second level.

**begin**

**definition** *newlist*  $\equiv$  *replicate* *N* *D*

**sepref-register** *newlist*

**print-theorems**

- *PR-CONST* tag is added, pattern rule is generated

**sepref-register** *other-basename-newlist: newlist*

**print-theorems**

- The base name for the generated theorems can be overridden

**sepref-register** *yet-another-basename-newlist: PR-CONST newlist*

**print-theorems**

- If *PR-CONST* tag is specified, no pattern rule is generated automatically

**end**

Example for *mcomb*/arity theorems

**definition** *select-a-one* *l*  $\equiv$  *SPEC* ( $\lambda i. i < \text{length } l \wedge !i = (1::\text{nat})$ )

**sepref-register** *select-a-one*

**print-theorems**

- Arity and *mcomb* theorem is generated

The following command fails, as the specified interface type does not correspond to the HOL type of the term: **sepref-register** *hd* :: (*nat, nat*) *i-map*

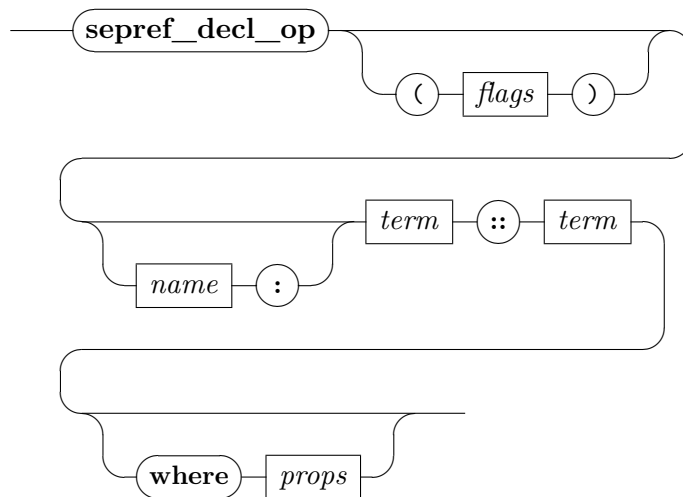
#### 4.2.6 High-Level tools for Interface/Implementation Declaration

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 **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 *and*) can be specified, which get constraint-preconditions of the relation.

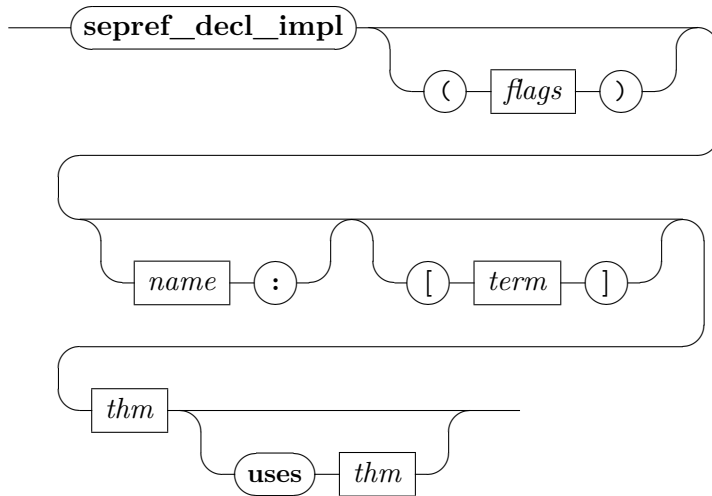
Finally, the following flags can be specified. Each flag can be prefixed by *no-* to invert its meaning:

**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.
- rawgoals** (default: false) Do not attempt to solve or simplify the goals
- register** (default: true) Register the generated theorems as operation rules.

#### 4.2.7 Defining synthesized Constants

The **sepref-definition** allows one to specify a name, an abstract term and a desired refinement relation in href-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.

**end**

## 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 *rprefs* 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  
  <proof>
```

**end**

#### Clarsimp all goals

*clarsimp-all* is a *clarsimp* on all goals. It takes the same arguments as *clarsimp*.

#### VCG solver

*vc-solve* clarsimps all subgoals. Then, it tries to apply a rule specified in the *solve:* argument, and tries to solve the result by *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 *auto* shall be tried on all subgoals, specify *solve: asm-rl*.

### 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 **subgoal**, and fully support schematic variables.

**applyS** applies a method to the current subgoal, and fails if the subgoal is not solved.

**apply1** applies a method to the current subgoal, and fails if the goal is solved or additional goals are created.

**focus** selects the current subgoal, and optionally applies a method.

**applyF** selects the current subgoal and applies a method.

**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).

**end**

# 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 <proof>
```

```
definition [to-relAPP]: graph-rel A ≡ ⟨A ×r 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
```

```
  <proof>
```

```

context begin
private definition graph-α1 l ≡ { (i,j). i < length l ∧ j ∈ !!i }

private definition graph-rel1 ≡ br graph-α1 (λ-. True)

private definition succ1 l i ≡ if i < length l then !!i else {}

private lemma succ1-refine: (succ1, op-graph-succ) ∈ graph-rel1 → Id → ⟨Id⟩set-rel
  ⟨proof⟩ 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 [sepref-import-param] = list-set-autoref-empty[folded op-set-empty-def]
  notes [fcomp-norm-unfold] = adjg-assn-def[symmetric]
begin
sepref-definition succ2 is (uncurry (RETURN oo succ1)) :: (assn2k*aid-assnk
  →a pure ((Id)list-set-rel))
  ⟨proof⟩

lemma adjg-succ-hnr[sepref-fr-rules]: [[CONSTRAINT (IS-PURE IS-LEFT-UNIQUE)
A; CONSTRAINT (IS-PURE IS-RIGHT-UNIQUE) A]
  ⇒ (uncurry succ2, uncurry (RETURN oo op-graph-succ)) ∈ (adjg-assn A)k*a
Ak →a pure ((the-pure A)list-set-rel)
  ⟨proof⟩

end

end

lemma [intf-of-assn]:
  intf-of-assn A (i::'I itself) ⇒ intf-of-assn (adjg-assn A) TYPE('I i-graph) ⟨proof⟩

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 ← Array.upd u l a;
      return a
    }) a;
    return a
  }

```

```

export-code cr-graph checking SML-imp
end

```

## 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**  $E :: 'v \text{ rel}$  **and**  $v0 :: 'v$  **and**  $tgt :: 'v \Rightarrow \text{bool}$

**begin**

**definition**  $dfs :: ('v \text{ set} \times \text{bool}) \text{ nres}$  **where**

```

  dfs  $\equiv$  do {
    (V,r)  $\leftarrow$  RECT ( $\lambda dfs$  (V,v).
      if  $v \in V$  then RETURN (V,False)
      else do {
        let  $V = \text{insert } v \ V$ ;
        if  $tgt \ v$  then
          RETURN (V,True)
        else
          FOREACHC (E“{v}”) ( $\lambda(-,b).$   $\neg b$ ) ( $\lambda v'$  (V,-).  $dfs$  (V,v')) (V,False)
      }
    ) ({},v0);
    RETURN (V,r)
  }

```

**definition**  $reachable \equiv \{v. (v0,v) \in E^*\}$

**definition**  $dfs\text{-spec} \equiv SPEC (\lambda(V,r). (r \longleftrightarrow reachable \cap Collect \ tgt \neq \{\}) \wedge (\neg r \longrightarrow V = reachable))$

**lemma**  $dfs\text{-correct}$ :

**assumes**  $fr$ : *finite reachable*

**shows**  $dfs \leq dfs\text{-spec}$

*<proof>*

**end**

**lemma** *dfs-correct'*: (*uncurry2 dfs*, *uncurry2 dfs-spec*)  
∈ [ $\lambda((E,s),t). \text{finite} (\text{reachable } E s)$ ]<sub>f</sub> ((*Id* ×<sub>r</sub> *Id*) ×<sub>r</sub> *Id*) → *Id* nres-rel  
<proof>

## 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*)<sup>k</sup>\*<sub>a</sub>*nat-assn*<sup>k</sup>\*<sub>a</sub>(*pure (nat-rel → bool-rel)*)<sup>k</sup>  
→<sub>a</sub> *prod-assn (ias.assn nat-assn) bool-assn*  
<proof>

**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) ⇒*  
<*adjg-assn nat-assn E Ei*>  
*dfs-impl Ei s tgt*  
<  $\lambda(Vi,r). \exists_A V. \text{adjg-assn nat-assn } E Ei * \text{ias.assn nat-assn } V Vi * \uparrow((r \longleftrightarrow \text{reachable } E s \cap \text{Collect } \text{tgt} \neq \{\}) \wedge (\neg r \longrightarrow V = \text{reachable } E s))$  ><sub>t</sub>  
<proof>

**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 infty :: (heap) heap
begin
  instance
    ⟨proof⟩
end

```

```

fun infty-assn where
  infty-assn A (Num x) (Num y) = A x y
| infty-assn A Infty Infty = emp
| infty-assn - - - = false

```

Connection with *infty-rel*

```

lemma infty-assn-pure-conv: infty-assn (pure A) = pure (⟨A⟩infty-rel)
  ⟨proof⟩

```

```

lemmas [sepref-import-rewrite, fcomp-norm-unfold, sepref-frame-normrel-eqs] =
  infty-assn-pure-conv[symmetric]

```

```

lemmas [constraint-simps] = infty-assn-pure-conv

```

```

lemma infty-assn-pure[safe-constraint-rules]: is-pure A  $\implies$  is-pure (infty-assn A)
  ⟨proof⟩

```

```

lemma infty-assn-id[simp]: infty-assn id-assn = id-assn
  ⟨proof⟩

```

```

lemma [safe-constraint-rules]: IS-BELOW-ID R  $\implies$  IS-BELOW-ID (⟨R⟩infty-rel)
  ⟨proof⟩

```

**sepref-register** *Num* *Infty*

```

lemma Num-hnr[sepref-fr-rules]: (return o Num, RETURN o Num)  $\in A^d \rightarrow_a$  infty-assn A
  ⟨proof⟩

```

```

lemma Infty-hnr[sepref-fr-rules]: (uncurry0 (return Infty), uncurry0 (RETURN Infty))  $\in$  unit-assnk  $\rightarrow_a$  infty-assn A
  ⟨proof⟩

```

**sepref-register** *case-infty*

```

lemma [sepref-monadify-arity]: case-infty  $\equiv$   $\lambda_2 f1\ f2\ x.\ SP\ case-infty\ \$f1\ (\lambda_2 x.\ f2\ \$x)\ \$x$ 
  ⟨proof⟩

```

```

lemma [sepref-monadify-comb]: case-infty  $\$f1\ \$f2\ \$x \equiv (\gg)\$(EVAL\ \$x)\$(\lambda_2 x.\ SP\ case-infty\ \$f1\ \$f2\ \$x)$ 
  ⟨proof⟩

```

**lemma** [sepref-monadify-comb]:  $EVAL\$(case\text{-infty}\$f1\$(\lambda_2x. f2\ x)\$x)$   
 $\equiv (\gg)\$(EVAL\$x)\$(\lambda_2x. SP\ case\text{-infty}\$(EVAL\ \$\ f1)\$(\lambda_2x. EVAL\ \$\ f2\ x)\$x)$   
 <proof>

**lemma** *infty-assn-ctxt*:  $infty\text{-assn}\ A\ x\ y = z \implies hn\text{-ctxt}\ (infty\text{-assn}\ A)\ x\ y = z$   
 <proof>

**lemma** *infty-cases-hnr*[sepref-prep-comb-rule, sepref-comb-rules]:

**fixes**  $A\ e\ e'$   
**defines** [simp]:  $INVe \equiv hn\text{-invalid}\ (infty\text{-assn}\ A)\ e\ e'$   
**assumes**  $FR: \Gamma \implies_t hn\text{-ctxt}\ (infty\text{-assn}\ A)\ e\ e' * F$   
**assumes**  $Infty: \llbracket e = Infty; e' = Infty \rrbracket \implies hn\text{-refine}\ (hn\text{-ctxt}\ (infty\text{-assn}\ A)\ e\ e' * F)\ f1'\ (hn\text{-ctxt}\ XX1\ e\ e' * \Gamma 1')\ R\ f1$   
**assumes**  $Num: \bigwedge x1\ x1a. \llbracket e = Num\ x1; e' = Num\ x1a \rrbracket \implies hn\text{-refine}\ (hn\text{-ctxt}\ A\ x1\ x1a * INVe * F)\ (f2'\ x1a)\ (hn\text{-ctxt}\ A'\ x1\ x1a * hn\text{-ctxt}\ XX2\ e\ e' * \Gamma 2')\ R\ (f2\ x1)$   
**assumes**  $MERGE2[unfolding\ hn\text{-ctxt}\text{-def}]: \Gamma 1' \vee_A \Gamma 2' \implies_t \Gamma'$   
**shows**  $hn\text{-refine}\ \Gamma\ (case\text{-infty}\ f1'\ f2'\ e')\ (hn\text{-ctxt}\ (infty\text{-assn}\ A')\ e\ e' * \Gamma')\ R\ (case\text{-infty}\$f1\$(\lambda_2x. f2\ x)\$e)$   
 <proof>  
**apply1** *extract-hnr-invalids*  
 <proof>  
**applyS** (*simp add: hn-ctxt-def*)  
 <proof>  
**applyS** (*simp add: hn-ctxt-def*)  
 <proof>  
**apply1** (*rule entt-fr-drop*)  
**applyS** (*simp add: hn-ctxt-def*)  
**apply1** (*rule entt-trans[OF - MERGE2]*)  
**applyS** (*simp add:*)  
 <proof>

**lemma** *hnr-val*[sepref-fr-rules]:  $(return\ o\ Weight.val, RETURN\ o\ Weight.val) \in [\lambda x. x \neq Infty]_a\ (infty\text{-assn}\ A)^d \rightarrow A$   
 <proof>

**context**

**fixes**  $A :: 'a::weight \Rightarrow 'b \Rightarrow assn$   
**fixes** *plusi*  
**assumes**  $GA[unfolding\ GEN\text{-ALGO}\text{-def},\ sepref\text{-fr}\text{-rules}]: GEN\text{-ALGO}\ plusi\ (\lambda f. (uncurry\ f, uncurry\ (RETURN\ oo\ (+))) \in A^k *_a A^k \rightarrow_a A)$   
**begin**  
**sepref-thm** *infty-plus-impl* **is**  $uncurry\ (RETURN\ oo\ (+)) :: ((infty\text{-assn}\ A)^k *_a (infty\text{-assn}\ A)^k \rightarrow_a infty\text{-assn}\ A)$   
 <proof>  
**end**  
**concrete-definition** *infty-plus-impl* **uses** *infty-plus-impl.refine-raw* **is**  $(uncurry\ ?impl, -) \in -$   
**lemmas** [sepref-fr-rules] = *infty-plus-impl.refine*

**definition** *infty-less* where

*infty-less* *lt a b*  $\equiv$  case (*a*,*b*) of (*Num a*, *Num b*)  $\Rightarrow$  *lt a b* | (*Num -*, *Infty*)  $\Rightarrow$  *True* | -  $\Rightarrow$  *False*

**lemma** *infty-less-param*[*param*]:

(*infty-less*,*infty-less*)  $\in$  (*R* $\rightarrow$ *R* $\rightarrow$ *bool-rel*)  $\rightarrow$   $\langle$ *R* $\rangle$ *infty-rel*  $\rightarrow$   $\langle$ *R* $\rangle$ *infty-rel*  $\rightarrow$  *bool-rel*  
 $\langle$ *proof* $\rangle$

**lemma** *infty-less-eq-less*: *infty-less* (*<*) = (*<*)

$\langle$ *proof* $\rangle$

**context**

**fixes** *A* :: '*a*::*weight*  $\Rightarrow$  '*b*  $\Rightarrow$  *assn*

**fixes** *lessi*

**assumes** *GA*[*unfolded GEN-ALGO-def*, *sepref-fr-rules*]: *GEN-ALGO lessi* ( $\lambda$ *f*.  
(*uncurry f*,*uncurry (RETURN oo (<))*)) $\in$ *A*<sup>*k*</sup>\*<sub>*a*</sub>*A*<sup>*k*</sup>  $\rightarrow_a$  *bool-assn*)

**begin**

**sepref-thm** *infty-less-impl* **is** *uncurry (RETURN oo (<))* :: ((*infty-assn A*)<sup>*k*</sup> \*<sub>*a*</sub>  
(*infty-assn A*)<sup>*k*</sup>  $\rightarrow_a$  *bool-assn*)

$\langle$ *proof* $\rangle$

**end**

**concrete-definition** *infty-less-impl* **uses** *infty-less-impl.refine-raw* **is** (*uncurry ?impl*,-)  $\in$ -

**lemmas** [*sepref-fr-rules*] = *infty-less-impl.refine*

**lemma** *param-mpath'*: (*mpath'*,*mpath'*)

$\in$   $\langle\langle$ *A* $\times_r$  *B*  $\times_r$  *A* $\rangle$ *list-rel*  $\times_r$  *B* $\rangle$ *option-rel*  $\rightarrow$   $\langle\langle$ *A* $\times_r$  *B*  $\times_r$  *A* $\rangle$ *list-rel* $\rangle$ *option-rel*  
 $\langle$ *proof* $\rangle$

**lemmas** (**in** -) [*sepref-import-param*] = *param-mpath'*

**lemma** *param-mpath-weight'*:

(*mpath-weight'*, *mpath-weight'*)  $\in$   $\langle\langle$ *A* $\times_r$  *B* $\times_r$  *A* $\rangle$ *list-rel*  $\times_r$  *B* $\rangle$ *option-rel*  $\rightarrow$   $\langle$ *B* $\rangle$ *infty-rel*  
 $\langle$ *proof* $\rangle$

**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)*)  $\in$   $\langle$ *br*  $\alpha$  *r* *res-invarm* $\rangle$ *nres-rel*  
 $\langle$ *proof* $\rangle$

**end**

**locale** *Dijkstra-Impl* = **fixes** *w-dummy* :: '*W*::{*weight*,*heap*}

**begin**

*Weights*

**sepref-register**  $0::'W$   
**lemmas** [*sepref-import-param*] =  
 $IdI[of\ 0::'W]$

**abbreviation**  $weight-assn \equiv id-assn :: 'W \Rightarrow -$

**lemma** *w-plus-param*:  $((+), (+)::'W \Rightarrow -) \in Id \rightarrow Id \rightarrow Id \langle proof \rangle$   
**lemma** *w-less-param*:  $((<), (<)::'W \Rightarrow -) \in Id \rightarrow Id \rightarrow Id \langle proof \rangle$   
**lemmas** [*sepref-import-param*] = *w-plus-param w-less-param*  
**lemma** [*sepref-gen-algo-rules*]:  
 $GEN-ALGO (return\ oo\ (+)) (\lambda f. (uncurry\ f, uncurry\ (RETURN\ oo\ (+))) \in$   
 $id-assn^k *_a id-assn^k \rightarrow_a id-assn)$   
 $GEN-ALGO (return\ oo\ (<)) (\lambda f. (uncurry\ f, uncurry\ (RETURN\ oo\ (<))) \in$   
 $id-assn^k *_a id-assn^k \rightarrow_a id-assn)$   
 $\langle proof \rangle$

**lemma** *conv-prio-pop-min*:  $prio-pop-min\ m = do\ \{$   
 $ASSERT\ (dom\ m \neq \{\});$   
 $((k,v),m) \leftarrow mop-pm-pop-min\ id\ m;$   
 $RETURN\ (k,v,m)$   
 $\}$   
 $\langle proof \rangle$   
**end**

**context fixes**  $N :: nat$  **and**  $w-dummy::'W::\{heap,weight\}$  **begin**

**interpretation** *Dijkstra-Impl w-dummy*  $\langle proof \rangle$

**definition** *drmap-assn2*  $\equiv IICF-Sepl-Binding.iam.assn$   
 $(pure\ (node-rel\ N))$   
 $(prod-assn$   
 $(list-assn\ (prod-assn\ (pure\ (node-rel\ N))\ (prod-assn\ weight-assn\ (pure\ (node-rel$   
 $N))))))$   
 $weight-assn)$

**concrete-definition** *mdijkstra'* **uses** *Dijkstra.impl-aux*

**sepref-definition** *dijkstra-imp* **is**  $uncurry\ mdijkstra'$   
 $:: (is-graph\ N\ (Id::('W \times 'W)\ set))^k *_a (pure\ (node-rel\ N))^k \rightarrow_a\ drmap-assn2$   
 $\langle proof \rangle$   
**export-code** *dijkstra-imp* **checking** *SML-imp*  
**end**

The main correctness theorem

**thm** *Dijkstra.mdijkstra-correct*

**lemma** *mdijkstra'-aref*:  $(uncurry\ mdijkstra', uncurry\ (SPEC\ oo\ weighted-graph.is-shortest-path-map))$

$\in [\lambda(G, v0). \text{Dijkstra } G \ v0]_f \text{Id} \times_r \text{Id} \rightarrow \langle \text{br Dijkstra.c}r \ \text{Dijkstra.res-invar}m \rangle_{\text{nres-rel}}$   
 $\langle \text{proof} \rangle$

**definition**  $\text{drmap-assn } N \equiv \text{hr-comp } (\text{drmap-assn2 } N) \ (\text{br Dijkstra.c}r \ \text{Dijkstra.res-invar}m)$

**context notes**  $[\text{fcomp-norm-unfold}] = \text{drmap-assn-def}[\text{symmetric}]$  **begin**

**theorem**  $\text{dijkstra-imp-correct}$ :  $(\text{uncurry } (\text{dijkstra-imp } N), \text{uncurry } (\text{SPEC} \circ \circ \text{weighted-graph.is-shortest-path-m}$   
 $\in [\lambda(G, v0). v0 \in \text{nodes } G \wedge (\forall (v, w, v') \in \text{edges } G. 0 \leq w)]_a \ (\text{is-graph } N \ \text{Id})^k$   
 $*_a \ (\text{node-assn } N)^k \rightarrow \text{drmap-assn } N$   
 $\langle \text{proof} \rangle$

**end**

**corollary**  $\text{dijkstra-imp-rule}$ :

$\langle \text{is-graph } n \ \text{Id } G \ Gi \ * \ \uparrow(v0 \in \text{nodes } G \wedge (\forall (v, w, v') \in \text{edges } G. 0 \leq w)) \rangle$   
 $\text{dijkstra-imp } n \ Gi \ v0$   
 $\langle \lambda mi. (\text{is-graph } n \ \text{Id}) \ G \ Gi$   
 $* \ (\exists Am. \text{drmap-assn } n \ m \ mi \ * \ \uparrow(\text{weighted-graph.is-shortest-path-map } G \ v0$   
 $m)) \rangle_t$   
 $\langle \text{proof} \rangle$

**end**

## 5.4 Imperative Implementation of of Nested DFS (HPY-Improvement)

**theory**  $\text{Sepref-NDFS}$

**imports**

$\text{../Sepref}$   
 $\text{Collections-Examples.Nested-DFS}$   
 $\text{Sepref-Graph}$   
 $\text{HOL-Library.Code-Target-Numeral}$

**begin**

**sepref-decl-intf**  $'v \ i\text{-red-witness}$  **is**  $'v \ \text{list} \ * \ 'v$

**lemma**  $\text{id-red-witness}[\text{id-rules}]$ :

$\text{red-init-witness} \ ::_i \ \text{TYPE}('v \Rightarrow 'v \Rightarrow 'v \ i\text{-red-witness} \ \text{option})$   
 $\text{prep-wit-red} \ ::_i \ \text{TYPE}('v \Rightarrow 'v \ i\text{-red-witness} \ \text{option} \Rightarrow 'v \ i\text{-red-witness} \ \text{option})$   
 $\langle \text{proof} \rangle$

**definition**

$\text{red-witness-rel-def-internal}$ :  $\text{red-witness-rel } R \equiv \langle \langle R \rangle \text{list-rel}, R \rangle \text{prod-rel}$

**lemma**  $\text{red-witness-rel-def}$ :  $\langle R \rangle \text{red-witness-rel} \equiv \langle \langle R \rangle \text{list-rel}, R \rangle \text{prod-rel}$   
 $\langle \text{proof} \rangle$

**lemma** *red-witness-rel-sv*[*constraint-rules*]:  
*single-valued R*  $\implies$  *single-valued* ( $\langle R \rangle$ *red-witness-rel*)  
 $\langle$ *proof* $\rangle$

**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* ( $\langle R \rangle$ *red-witness-rel*)))  
(*RETURN*\$(*red-init-witness*\$*u*\$*v*))  
 $\langle$ *proof* $\rangle$

**lemma** [*sepref-fr-rules*]: *hn-refine*  
(*hn-val R u u' \* hn-ctxt* (*option-assn* (*pure* ( $\langle R \rangle$ *red-witness-rel*))) *w w'*)  
(*return* (*prep-wit-red u' w'*))  
(*hn-val R u u' \* hn-ctxt* (*option-assn* (*pure* ( $\langle R \rangle$ *red-witness-rel*))) *w w'*)  
(*option-assn* (*pure* ( $\langle R \rangle$ *red-witness-rel*)))  
(*RETURN*\$(*prep-wit-red*\$*u*\$*w*))  
 $\langle$ *proof* $\rangle$

**term** *red-dfs*

**sepref-definition** *red-dfs-impl* **is**  
(*uncurry2* (*uncurry red-dfs*))  
 $::$  (*adjg-assn nat-assn*)<sup>*k*</sup> \*<sub>*a*</sub> (*ias.assn nat-assn*)<sup>*k*</sup> \*<sub>*a*</sub> (*ias.assn nat-assn*)<sup>*d*</sup> \*<sub>*a*</sub> *nat-assn*<sup>*k*</sup>  
 $\rightarrow_a$  *UNSPEC*  
 $\langle$ *proof* $\rangle$

**export-code** *red-dfs-impl* **checking** *SML-imp*

**declare** *red-dfs-impl.refine*[*sepref-fr-rules*]

**sepref-register** *red-dfs*  $::$  '*a i-graph*  $\Rightarrow$  '*a set*  $\Rightarrow$  '*a set*  $\Rightarrow$  '*a*  
 $\Rightarrow$  ('*a set* \* '*a i-red-witness option*) *nres*

**lemma** *id-init-wit-blue*[*id-rules*]:  
*init-wit-blue*  $::_i$  *TYPE*('*a*  $\Rightarrow$  '*a i-red-witness option*  $\Rightarrow$  '*a blue-witness*)  
 $\langle$ *proof* $\rangle$

**lemma** *hn-blue-wit*[*sepref-import-param*]:  
(*NO-CYC,NO-CYC*) $\in$ *blue-wit-rel*  
(*prep-wit-blue,prep-wit-blue*) $\in$ *nat-rel* $\rightarrow$ *blue-wit-rel* $\rightarrow$ *blue-wit-rel*  
(*(=),(=)*) $\in$ *blue-wit-rel* $\rightarrow$ *blue-wit-rel* $\rightarrow$ *bool-rel*  
 $\langle$ *proof* $\rangle$

**lemma** *hn-init-wit-blue*[*sepref-fr-rules*]: *hn-refine*  
(*hn-val nat-rel v v' \* hn-ctxt* (*option-assn* (*pure* ( $\langle$ *nat-rel* $\rangle$ *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))
⟨proof⟩

```

**lemma** *hn-extract-res*[*sepref-import-param*]:  
*(extract-res, extract-res) ∈ blue-wit-rel → Id*  
⟨proof⟩

**thm** *red-dfs-impl.refine*

**sepref-definition** *blue-dfs-impl* is *uncurry2 blue-dfs* :: ((*adjg-assn nat-assn*)<sup>k</sup>\*<sub>a</sub>(*ias-assn nat-assn*)<sup>k</sup>\*<sub>a</sub>*nat-assn*<sup>k</sup>→<sub>a</sub>*id-assn*)  
⟨proof⟩

**export-code** *blue-dfs-impl checking SML-imp*

**definition** *blue-dfs-spec* *E A v0* ≡ *SPEC* ( $\lambda r$ . *case r of None* ⇒  $\neg$  *has-acc-cycle* *E A v0*  
| *Some (v, pc, pv)* ⇒ *is-acc-cycle* *E A v0 v pv pc*)

**lemma** *blue-dfs-correct'*: (*uncurry2 blue-dfs, uncurry2 blue-dfs-spec*) ∈ [ $\lambda((E,A),v0)$ .  
*finite (E\*“{v0}”)*]<sub>f</sub> ((*Id*×<sub>r</sub>*Id*)×<sub>r</sub>*Id*) → *⟨Id⟩nres-rel*  
⟨proof⟩

**lemmas** *blue-dfs-impl-correct' = blue-dfs-impl.refine[FCOMP blue-dfs-correct']*

**theorem** *blue-dfs-impl-correct*:

**fixes** *E*

**assumes** *finite (E\*“{v0}”)*

**shows** *<ias-assn id-assn A A-impl \* adjg-assn id-assn E succ-impl>*  
*blue-dfs-impl succ-impl A-impl v0*

*<\lambda r. ias-assn id-assn A A-impl \* adjg-assn id-assn E succ-impl*

*\* \uparrow*

*case r of None* ⇒  $\neg$ *has-acc-cycle* *E A v0*

| *Some (v,pc,pv)* ⇒ *is-acc-cycle* *E A v0 v pv pc*

*>\_t*

⟨proof⟩

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.

**context**

**fixes** *N :: nat*

**begin**

**lemma** *testsuite-blue-dfs-modify*:

```

({}::nat set, {}::nat set, {}::nat set, s)
= (op-ias-empty-sz N, op-ias-empty-sz N, op-ias-empty-sz N, s)
⟨proof⟩

```

**sepref-definition** *blue-dfs-impl-sz* is *uncurry2* *blue-dfs* :: ((*adjg-assn nat-assn*)<sup>k</sup>\*<sub>a</sub>(*ias-assn nat-assn*)<sup>k</sup>\*<sub>a</sub>*nat-assn*<sup>k</sup>→<sub>a</sub>*id-assn*)  
 ⟨proof⟩

**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*\*‘{*v0*}’)

**shows** <*ias-assn id-assn A A-impl \* adjg-assn id-assn E succ-impl*>

*blue-dfs-impl-sz N succ-impl A-impl v0*

< $\lambda r.$  *ias-assn id-assn A A-impl \* adjg-assn id-assn E succ-impl*

\*  $\uparrow$ (

*case r of None*  $\Rightarrow$   $\neg$ *has-acc-cycle E A v0*

| *Some (v,pc,pv)*  $\Rightarrow$  *is-acc-cycle E A v0 v pv pc*

)><sub>t</sub>

⟨proof⟩

**end**

⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨ML⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩

## 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*  $\neq$  {*x*})  $\gg$  *SPEC* ( $\lambda (x, s'). x \in s \wedge s' = s - \{x\}$ )

**lemma** *take-from-set-correct*:

**assumes** *s*  $\neq$  {*x*}

**shows** *take-from-set s*  $\leq$  *SPEC* ( $\lambda (x, s'). x \in s \wedge s' = s - \{x\}$ )

⟨proof⟩

**lemmas** [*refine-vcg*] = *take-from-set-correct*[*THEN order.trans*]



**definition** *take-from-mset* **where**

*take-from-mset*  $s = \text{ASSERT } (s \neq \{\#\}) \gg \text{SPEC } (\lambda (x, s'). x \in\# s \wedge s' = s - \{\#x\#})$

**lemma** *take-from-mset-correct*:

**assumes**  $s \neq \{\#\}$

**shows**  $\text{take-from-mset } s \leq \text{SPEC } (\lambda (x, s'). x \in\# s \wedge s' = s - \{\#x\#})$   
 $\langle \text{proof} \rangle$

**lemmas**  $[\text{refine-vcg}] = \text{take-from-mset-correct}[\text{THEN } \text{order.trans}]$

**lemma** *set-mset-mp*:  $\text{set-mset } m \subseteq s \implies n < \text{count } m \ x \implies x \in s$

$\langle \text{proof} \rangle$

**lemma** *pred-not-lt-is-zero*:  $(\neg n - \text{Suc } 0 < n) \longleftrightarrow n=0 \langle \text{proof} \rangle$

## 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**  $\text{subsumes} :: 'a \Rightarrow 'a \Rightarrow \text{bool}$  (**infix**  $\langle \preceq \rangle$  50) — Subsumption preorder

**begin**

**definition** *reachable* **where**

$\text{reachable} = E^{**} a_0$

**definition** *F-reachable*  $\equiv \exists a. \text{reachable } a \wedge F a$

**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*:  $\text{finite } \{a. \text{reachable } a\}$

**assumes** *refl*[*intro!*, *simp*]:  $a \preceq a$

**and** *trans*[*trans*]:  $a \preceq b \implies b \preceq c \implies a \preceq c$

**assumes** *mono*:  $a \preceq b \implies E a a' \implies \text{reachable } a \implies \text{reachable } b \implies \exists b'. E b b' \wedge a' \preceq b'$

**and** *F-mono*:  $a \preceq a' \implies F a \implies F a'$

**begin**

**lemma** *start-reachable*[*intro!*, *simp*]:  
  *reachable*  $a_0$   
  ⟨*proof*⟩

**lemma** *step-reachable*:  
  **assumes** *reachable*  $a$   $E$   $a$   $a'$   
  **shows** *reachable*  $a'$   
  ⟨*proof*⟩

**lemma** *finitely-branching*:  
  **assumes** *reachable*  $a$   
  **shows** *finite* (*Collect* ( $E$   $a$ ))  
  ⟨*proof*⟩

**end**

### 5.5.3 Worklist Algorithm

**term** *card*

**context** *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* =  
( $\forall a \in \text{passed}. \forall a'. E a a' \longrightarrow (\exists b' \in \text{passed} \cup \text{set-mset } \text{wait}. a' \preceq b')$ )

**definition** *start-subsumed* *passed* *wait* = ( $\exists a \in \text{passed} \cup \text{set-mset } \text{wait}. a_0 \preceq a$ )

**definition** *worklist-inv*  $\equiv \lambda$  (*passed*, *wait*, *brk*).

*passed*  $\subseteq$  *Collect* *reachable*  $\wedge$   
(*brk*  $\longrightarrow (\exists f. \text{reachable } f \wedge F f)$ )  $\wedge$   
( $\neg$  *brk*  $\longrightarrow$   
  *worklist-inv-frontier* *passed* *wait*  
   $\wedge (\forall a \in \text{passed} \cup \text{set-mset } \text{wait}. \neg F a)$   
   $\wedge$  *start-subsumed* *passed* *wait*  
   $\wedge$  *set-mset* *wait*  $\subseteq$  *Collect* *reachable*)

**definition** *add-succ-spec* *wait*  $a \equiv \text{SPEC } (\lambda(\text{wait}', \text{brk}).$

  if  $\exists a'. E a a' \wedge F a'$  then

*brk*

  else *set-mset* *wait'* = *set-mset* *wait*  $\cup \{a' . E a a'\} \wedge \neg \text{brk}$

)

**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*  
*<proof>*

**context**  
**begin**

**private lemma** *aux1*:

**assumes**  $\forall x \in \text{passed}. \neg a \preceq x$   
**and**  $\text{passed} \subseteq \text{Collect reachable}$   
**and** *reachable a*

**shows**

$((\text{insert } a \text{ passed}, \text{wait}', \text{brk}'),$   
 $\text{passed}, \text{wait}, \text{brk})$

$\in$  *worklist-var*  
 ⟨*proof*⟩ **lemma** *aux2*:  
   **assumes**  
      $a' \in$  *passed*  
      $a \preceq a'$   
      $a \in \#$  *wait*  
     *worklist-inv-frontier passed wait*  
   **shows** *worklist-inv-frontier passed* ( $wait - \{\#a\}$ )  
 ⟨*proof*⟩ **lemma** *aux5*:  
   **assumes**  
      $a' \in$  *passed*  
      $a \preceq a'$   
      $a \in \#$  *wait*  
     *start-subsumed passed wait*  
   **shows** *start-subsumed passed* ( $wait - \{\#a\}$ )  
 ⟨*proof*⟩ **lemma** *aux3*:  
   **assumes**  
      $set\text{-}mset\ wait \subseteq$  *Collect reachable*  
      $a \in \#$  *wait*  
      $set\text{-}mset\ wait' = set\text{-}mset\ (wait - \{\#a\}) \cup$  *Collect* ( $E\ a$ )  
     *worklist-inv-frontier passed wait*  
   **shows** *worklist-inv-frontier* (*insert a passed*) *wait'*  
 ⟨*proof*⟩ **lemma** *aux6*:  
   **assumes**  
      $a \in \#$  *wait*  
     *start-subsumed passed wait*  
      $set\text{-}mset\ wait' = set\text{-}mset\ (wait - \{\#a\}) \cup$  *Collect* ( $E\ a$ )  
   **shows** *start-subsumed* (*insert a passed*) *wait'*  
   ⟨*proof*⟩  
  
   **lemma** *aux4*:  
     **assumes** *worklist-inv-frontier passed*  $\{\#\}$  *reachable x start-subsumed passed*  
      $\{\#\}$   
       *passed*  $\subseteq$  *Collect reachable*  
     **shows**  $\exists x' \in$  *passed*.  $x \preceq x'$   
     ⟨*proof*⟩  
  
   **theorem** *worklist-algo-correct*:  
     *worklist-algo*  $\leq$  *SPEC* ( $\lambda\ brk.$   $brk \longleftrightarrow$  *F-reachable*)  
     ⟨*proof*⟩  
  
   **lemmas** [*refine-vcg*] = *worklist-algo-correct*[*THEN order-trans*]  
  
   **end** — Context  
  
   **end** — Search Space

### 5.5.4 Towards an Implementation

**locale** *Worklist1-Defs* = *Search-Space-Defs* +  
**fixes** *succs* :: 'a  $\Rightarrow$  'a list

**locale** *Worklist1* = *Worklist1-Defs* + *Search-Space* +  
**assumes** *succs-correct*: *reachable* a  $\Longrightarrow$  *set* (*succs* a) = *Collect* (*E* a)  
**begin**

**definition** *add-succ1* wait a  $\equiv$  *nfoldli* (*succs* a) ( $\lambda(-,brk). \neg brk$ ) ( $\lambda a (wait,brk).$   
*if* *F* a *then* *RETURN* (wait,True) *else* *RETURN* (wait + {#a#},False)) (wait,  
False)

**lemma** *add-succ1-ref*[*refine*]:  $\llbracket (wait,wait') \in Id; (a,a') \in b\text{-rel } Id \text{ reachable} \rrbracket \Longrightarrow$   
*add-succ1* wait a  $\leq \Downarrow (Id \times_r \text{bool-rel})$  (*add-succ-spec* wait' a')  
 $\langle \text{proof} \rangle$

**definition** *worklist-algo1* **where**  
*worklist-algo1* = *do*  
{  
  *if* *F* a<sub>0</sub> *then* *RETURN* True  
  *else* *do* {  
    let *passed* = {};  
    let *wait* = {#a<sub>0</sub>#};  
    (*passed*, *wait*, *brk*)  $\leftarrow$  *WHILEIT* *worklist-inv* ( $\lambda (passed, wait, brk). \neg brk$   
 $\wedge$  *wait*  $\neq$  {#})  
    ( $\lambda (passed, wait, brk).$  *do*  
      {  
        (*a*, *wait*)  $\leftarrow$  *take-from-mset* *wait*;  
        *if* ( $\exists a' \in passed. a \preceq a'$ ) *then* *RETURN* (*passed*, *wait*, *brk*) *else*  
        *do*  
          {  
            (*wait*,*brk*)  $\leftarrow$  *add-succ1* *wait* a;  
            let *passed* = *insert* a *passed*;  
            *RETURN* (*passed*, *wait*, *brk*)  
          }  
        }  
      )  
      (*passed*, *wait*, *False*);  
      *RETURN* *brk*  
    }  
  }  
}

**lemma** *worklist-algo1-ref*[*refine*]: *worklist-algo1*  $\leq \Downarrow Id$  *worklist-algo*  
 $\langle \text{proof} \rangle$

**end**

```

end — Theory
theory Worklist-Subsumption-Impl
imports ../HICF/HICF 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-assnk →a A
    assumes [sepref-fr-rules]: (Fi, RETURN o PR-CONST F) ∈ Ak →a bool-assn
    assumes [sepref-fr-rules]: (uncurry Lei, uncurry (RETURN oo PR-CONST
(≼))) ∈ Ak *a Ak →a bool-assn
    assumes [sepref-fr-rules]: (succsi, RETURN o PR-CONST succs) ∈ Ak →a
list-assn A
  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
      ⟨proof⟩

    lemma take-from-mset-as-mop-mset-pick: take-from-mset = mop-mset-pick
      ⟨proof⟩

    lemma [safe-constraint-rules]: CN-FALSE is-pure A ⇒ is-pure A ⟨proof⟩

    sepref-thm worklist-algo2 is uncurry0 worklist-algo1 :: unit-assnk →a bool-assn
      ⟨proof⟩

  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 (≼) succs A succsi a0i Fi Lei
      ⟨proof⟩
  end

```

**lemma** *hnr-F-reachable*: (*uncurry0* (*worklist-algo2* *Lei a<sub>0</sub>i Fi succsi*), *uncurry0* (*RETURN F-reachable*))  
 $\in \text{unit-assn}^k \rightarrow_a \text{bool-assn}$   
 $\langle \text{proof} \rangle$

**end**

**context** *Worklist1* **begin**

**sepref-decl-op** *F-reachable* :: *bool-rel*  $\langle \text{proof} \rangle$

**lemma** [*def-pat-rules*]: *F-reachable*  $\equiv$  *op-F-reachable*  $\langle \text{proof} \rangle$

**lemma** *hnr-op-F-reachable*:

**assumes** *GEN-ALGO a<sub>0</sub>i* ( $\lambda a_0 i. (\text{uncurry0 } a_0 i, \text{uncurry0 } (\text{RETURN } a_0)) \in \text{unit-assn}^k \rightarrow_a A$ )

**assumes** *GEN-ALGO Fi* ( $\lambda Fi. (Fi, \text{RETURN } o F) \in A^k \rightarrow_a \text{bool-assn}$ )

**assumes** *GEN-ALGO Lei* ( $\lambda Lei. (\text{uncurry } Lei, \text{uncurry } (\text{RETURN } oo (\preceq))) \in A^k *_a A^k \rightarrow_a \text{bool-assn}$ )

**assumes** *GEN-ALGO succsi* ( $\lambda succsi. (\text{succsi}, \text{RETURN } o \text{succs}) \in A^k \rightarrow_a \text{list-assn } A$ )

**shows** (*uncurry0* (*worklist-algo2* *Lei a<sub>0</sub>i Fi succsi*), *uncurry0* (*RETURN* (*PR-CONST op-F-reachable*)))

$\in \text{unit-assn}^k \rightarrow_a \text{bool-assn}$

$\langle \text{proof} \rangle$

**sepref-decl-impl** *hnr-op-F-reachable*  $\langle \text{proof} \rangle$

**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 '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]:  $is\text{-}pure\ A \implies is\text{-}pure\ (enum\text{-}assn\ A)$   
 ⟨proof⟩

An identity rule is required to easily prove trivial refinement theorems

**lemma** *enum-assn-id*[simp]:  $enum\text{-}assn\ id\text{-}assn = id\text{-}assn$   
 ⟨proof⟩

Structural rules.

Without congruence condition

**lemma** *enum-match-nocong*:  $\llbracket \bigwedge x\ y. hn\text{-}ctxt\ A\ x\ y \implies_t hn\text{-}ctxt\ A'\ x\ y \rrbracket \implies hn\text{-}ctxt\ (enum\text{-}assn\ A)\ e\ e' \implies_t hn\text{-}ctxt\ (enum\text{-}assn\ A')\ e\ e'$   
 ⟨proof⟩

**lemma** *enum-merge-nocong*:

**assumes**  $\bigwedge x\ y. hn\text{-}ctxt\ A\ x\ y \vee_A hn\text{-}ctxt\ A'\ x\ y \implies_A hn\text{-}ctxt\ Am\ x\ y$   
**shows**  $hn\text{-}ctxt\ (enum\text{-}assn\ A)\ e\ e' \vee_A hn\text{-}ctxt\ (enum\text{-}assn\ A')\ e\ e' \implies_A hn\text{-}ctxt\ (enum\text{-}assn\ Am)\ e\ e'$   
 ⟨proof⟩

With congruence condition

**lemma** *enum-match-cong*[sepref-frame-match-rules]:  
 $\llbracket \bigwedge x\ y. \llbracket x \in set\text{-}enum\ e; y \in set\text{-}enum\ e' \rrbracket \implies hn\text{-}ctxt\ A\ x\ y \implies_t hn\text{-}ctxt\ A'\ x\ y \rrbracket \implies hn\text{-}ctxt\ (enum\text{-}assn\ A)\ e\ e' \implies_t hn\text{-}ctxt\ (enum\text{-}assn\ A')\ e\ e'$   
 ⟨proof⟩

**lemma** *enum-merge-cong*[sepref-frame-merge-rules]:

**assumes**  $\bigwedge x\ y. \llbracket x \in set\text{-}enum\ e; y \in set\text{-}enum\ e' \rrbracket \implies hn\text{-}ctxt\ A\ x\ y \vee_A hn\text{-}ctxt\ A'\ x\ y \implies_t hn\text{-}ctxt\ Am\ x\ y$   
**shows**  $hn\text{-}ctxt\ (enum\text{-}assn\ A)\ e\ e' \vee_A hn\text{-}ctxt\ (enum\text{-}assn\ A')\ e\ e' \implies_t hn\text{-}ctxt\ (enum\text{-}assn\ Am)\ e\ e'$   
 ⟨proof⟩

Propagating invalid

**lemma** *entt-invalid-enum*:  $hn\text{-}invalid\ (enum\text{-}assn\ A)\ e\ e' \implies_t hn\text{-}ctxt\ (enum\text{-}assn\ (invalid\text{-}assn\ A))\ e\ e'$   
 ⟨proof⟩

**lemmas** *invalid-enum-merge*[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 *sepref-to-hoare*)

**lemma** [*sepref-fr-rules*]:  $(\text{return } o E1, \text{RETURN } o E1) \in A^d \rightarrow_a \text{enum-assn } A$   
 $\langle \text{proof} \rangle$

**lemma** [*sepref-fr-rules*]:  $(\text{return } o E2, \text{RETURN } o E2) \in A^d \rightarrow_a \text{enum-assn } A$   
 $\langle \text{proof} \rangle$

**lemma** [*sepref-fr-rules*]:  $(\text{uncurry0 } (\text{return } E3), \text{uncurry0 } (\text{RETURN } E3)) \in \text{unit-assn}^k \rightarrow_a \text{enum-assn } A$   
 $\langle \text{proof} \rangle$

**lemma** [*sepref-fr-rules*]:  $(\text{uncurry } (\text{return } oo E4), \text{uncurry } (\text{RETURN } oo E4)) \in A^d *_a A^d \rightarrow_a \text{enum-assn } A$   
 $\langle \text{proof} \rangle$

**lemma** [*sepref-fr-rules*]:  $(\text{uncurry } (\text{return } oo E5), \text{uncurry } (\text{RETURN } oo E5)) \in \text{bool-assn}^k *_a A^d \rightarrow_a \text{enum-assn } A$   
 $\langle \text{proof} \rangle$

### 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

**sepref-register** *case-enum*

In the monadify phase, this eta-expands to make visible all required arguments

**lemma** [*sepref-monadify-arity*]:  $\text{case-enum} \equiv \lambda_2 f1 f2 f3 f4 f5 x. SP \text{ case-enum} \$ (\lambda_2 x. f1 \$ x) \$ (\lambda_2 x. f2 \$ x) \$ f3 \$ (\lambda_2 x y. f4 \$ x \$ y) \$ (\lambda_2 x y. f5 \$ x \$ y) \$ x$   
 $\langle \text{proof} \rangle$

This determines an evaluation order for the first-order operands

**lemma** [*sepref-monadify-comb*]:  $\text{case-enum} \$ f1 \$ f2 \$ f3 \$ f4 \$ f5 \$ x \equiv (\gg) \$ (\text{EVAL} \$ x) \$ (\lambda_2 x. SP \text{ case-enum} \$ f1 \$ f2 \$ f3 \$ f4 \$ f5 \$ x)$   $\langle \text{proof} \rangle$

This enables translation of the case-distinction in a non-monadic context.

**lemma** [*sepref-monadify-comb*]:  $\text{EVAL} \$ (\text{case-enum} \$ (\lambda_2 x. f1 x) \$ (\lambda_2 x. f2 x) \$ f3) \$ (\lambda_2 x y. f4 x y) \$ (\lambda_2 x y. f5 x y) \$ x$   
 $\equiv (\gg) \$ (\text{EVAL} \$ x) \$ (\lambda_2 x. SP \text{ case-enum} \$ (\lambda_2 x. \text{EVAL} \$ f1 x) \$ (\lambda_2 x. \text{EVAL} \$ f2 x) \$ (\text{EVAL} \$ f3) \$ (\lambda_2 x y. \text{EVAL} \$ f4 x y) \$ (\lambda_2 x y. \text{EVAL} \$ f5 x y) \$ x)$   
 $\langle \text{proof} \rangle$

Auxiliary lemma, to lift simp-rule over *hn-ctxt*

**lemma** *enum-assn-ctxt*:  $\text{enum-assn } A \ x \ y = z \implies \text{hn-ctxt } (\text{enum-assn } A) \ x \ y = z$   
 $\langle \text{proof} \rangle$

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 destructured 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** [*simp*]:  $INVe \equiv hn\text{-invalid } (enum\text{-assn } A) e e'$

**assumes**  $FR: \Gamma \Longrightarrow_t hn\text{-ctxt } (enum\text{-assn } A) e e' * F$

**assumes**  $E1: \bigwedge x1 x1a. \llbracket e = E1 x1; e' = E1 x1a \rrbracket \Longrightarrow hn\text{-refine } (hn\text{-ctxt } A x1 x1a * INVe * F) (f1' x1a) (hn\text{-ctxt } A1' x1 x1a * hn\text{-ctxt } XX1 e e' * \Gamma1') R (f1 x1)$

**assumes**  $E2: \bigwedge x2 x2a. \llbracket e = E2 x2; e' = E2 x2a \rrbracket \Longrightarrow hn\text{-refine } (hn\text{-ctxt } A x2 x2a * INVe * F) (f2' x2a) (hn\text{-ctxt } A2' x2 x2a * hn\text{-ctxt } XX2 e e' * \Gamma2') R (f2 x2)$

**assumes**  $E3: \llbracket e = E3; e' = E3 \rrbracket \Longrightarrow hn\text{-refine } (hn\text{-ctxt } (enum\text{-assn } A) e e' * F) f3' (hn\text{-ctxt } XX3 e e' * \Gamma3') R f3$

**assumes**  $E4: \bigwedge x41 x42 x41a x42a.$   
 $\llbracket e = E4 x41 x42; e' = E4 x41a x42a \rrbracket$   
 $\Longrightarrow hn\text{-refine } (hn\text{-ctxt } A x41 x41a * hn\text{-ctxt } A x42 x42a * INVe * F) (f4' x41a x42a) (hn\text{-ctxt } A4a' x41 x41a * hn\text{-ctxt } A4b' x42 x42a * hn\text{-ctxt } XX4 e e' * \Gamma4') R$   
 $(f4 x41 x42)$

**assumes**  $E5: \bigwedge x51 x52 x51a x52a.$   
 $\llbracket e = E5 x51 x52; e' = E5 x51a x52a \rrbracket$   
 $\Longrightarrow hn\text{-refine } (hn\text{-ctxt } bool\text{-assn } x51 x51a * hn\text{-ctxt } A x52 x52a * INVe * F) (f5' x51a x52a)$   
 $(hn\text{-ctxt } bool\text{-assn } x51 x51a * hn\text{-ctxt } A5' x52 x52a * hn\text{-ctxt } XX5 e e' * \Gamma5') R (f5 x51 x52)$

**assumes**  $MERGE1[un\text{folded } hn\text{-ctxt}\text{-def}]: \bigwedge x x'. hn\text{-ctxt } A1' x x' \vee_A hn\text{-ctxt } A2' x x' \vee_A hn\text{-ctxt } A3' x x' \vee_A hn\text{-ctxt } A4a' x x' \vee_A hn\text{-ctxt } A4b' x x' \vee_A hn\text{-ctxt } A5' x x' \Longrightarrow_t hn\text{-ctxt } A' x x'$

**assumes**  $MERGE2[un\text{folded } hn\text{-ctxt}\text{-def}]: \Gamma1' \vee_A \Gamma2' \vee_A \Gamma3' \vee_A \Gamma4' \vee_A \Gamma5' \Longrightarrow_t \Gamma'$

**shows**  $hn\text{-refine } \Gamma (case\text{-enum } f1' f2' f3' f4' f5' e') (hn\text{-ctxt } (enum\text{-assn } A') e e' * \Gamma') R (case\text{-enum } \$(\lambda_2 x. f1 x) \$(\lambda_2 x. f2 x) \$f3 \$(\lambda_2 x y. f4 x y) \$(\lambda_2 x y. f5 x y) \$e)$   
 $\langle proof \rangle$

**apply1** *extract-hnr-invalids*  
 $\langle proof \rangle$

**applyS** (*simp add: hn-ctxt-def*) — Match precondition for case, get *enum-assn* from assumption generated by *extract-hnr-invalids*  
 $\langle proof \rangle$

**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')
⟨proof⟩
applyS (simp add: hn-ctxt-def)
⟨proof⟩
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')
⟨proof⟩
applyS (simp add: hn-ctxt-def)
⟨proof⟩
applyS (simp add: hn-ctxt-def)
⟨proof⟩
apply1 (rule entt-fr-drop)
⟨proof⟩

apply1 (rule entt-trans[OF - MERGE1])
applyS (simp add: hn-ctxt-def entt-disjI1' entt-disjI2')

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')
⟨proof⟩
applyS (simp add: hn-ctxt-def)
⟨proof⟩
apply1 (rule entt-fr-drop)
⟨proof⟩

apply1 (rule ent-imp-entt)
applyS (simp add: hn-ctxt-def)

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')
⟨proof⟩

```

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

```

lemmas [sepref-comb-rules] = enum-cases-hnr[sepref-prep-comb-rule]

```

## 5.6.4 Regression Test

**definition** *test1* (*e::bool enum*)  $\equiv$  *RETURN e*

**sepref-definition** *test1-impl* **is** *test1* :: (*enum-assn bool-assn*)<sup>d</sup>  $\rightarrow_a$  *enum-assn bool-assn*

*<proof>*

**sepref-register** *test1*

**lemmas** [*sepref-fr-rules*] = *test1-impl.refine*

**definition** *test*  $\equiv$  *do* {

*let* *x* = *E1 True*;

-  $\leftarrow$  *case* *x* *of*

*E1 a*  $\Rightarrow$  *RETURN (Some a)* — Access and invalidate compound inside case

| -  $\Rightarrow$  *RETURN (Some True)*;

-  $\leftarrow$  *test1 x*; — Rely on structure being there, with valid compound

— Same thing again, with merge

-  $\leftarrow$  *if True then*

*case* *x* *of*

*E1 a*  $\Rightarrow$  *RETURN (Some a)* — Access and invalidate compound inside case

| -  $\Rightarrow$  *RETURN (Some True)*

*else RETURN None*;

-  $\leftarrow$  *test1 x*; — Rely on structure being there, with valid compound

— Now test with non-pure

*let* *a* = *op-array-replicate 4 (3::nat)*;

*let* *x* = *E5 False a*;

-  $\leftarrow$  *case* *x* *of*

*E1 -*  $\Rightarrow$  *RETURN (0::nat)*

| *E2 -*  $\Rightarrow$  *RETURN 1*

| *E3*  $\Rightarrow$  *RETURN 0*

| *E4 - -*  $\Rightarrow$  *RETURN 0*

| *E5 - a*  $\Rightarrow$  *mop-list-get a 0*;

— Rely on that compound still exists (it's components are only read in the case above)

*case* *x* *of*

*E1 a*  $\Rightarrow$  *do* {*mop-list-set a 0 0*; *RETURN (0::nat)*}

| *E2 -*  $\Rightarrow$  *RETURN 1*

| *E3*  $\Rightarrow$  *RETURN 0*

| *E4 - -*  $\Rightarrow$  *RETURN 0*

| *E5 - -*  $\Rightarrow$  *RETURN 0*

}

**lemmas** [*safe-constraint-rules*] = *CN-FALSEI*[*of is-pure invalid-assn A for A*]

```

sepref-definition foo is uncurry0 test :: unit-assnk →a nat-assn
  ⟨proof⟩

```

end

## 5.7 Snippet to Define Custom Combinators

```

theory Sepref-Snip-Combinator
imports ../..//IICF/IICF
begin

```

### 5.7.1 Definition of the Combinator

Currently, when defining new combinators, you are largely on your own. If you can show your combinator equivalent to some other, already existing, combinator, you should apply this equivalence in the monadify phase.

In this example, we show the development of a map combinator from scratch.

We set ourselves in to a context where we fix the abstract and concrete arguments of the monadic map combinator, as well as the refinement assertions, and a frame, that represents the remaining heap content, and may be read by the map-function.

```

context
  fixes f :: 'a ⇒ 'b nres
  fixes l :: 'a list

  fixes fi :: 'ai ⇒ 'bi Heap
  fixes li :: 'ai list

  fixes A A' :: 'a ⇒ 'ai ⇒ assn — Refinement for list elements before and after
    map-function. Different, as map function may invalidate list elements!
  fixes B :: 'b ⇒ 'bi ⇒ assn

  fixes F :: assn — Symbolic frame, representing all heap content the map-function
    body may access

  notes [[sepref-register-adhoc f l]] — Register for operation id

  assumes f-rl: hn-refine (hn-ctxt A x xi * F) (fi xi) (hn-ctxt A' x xi * F) B
  (f$x)
  — Refinement for f

begin

```

We implement our combinator using the monadic refinement framework.

```

definition mmap ≡ RECT (λmmap.
  λ[] ⇒ RETURN []
  | x#xs ⇒ do { x ← f x; xs ← mmap xs; 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**  $F\text{-assn } (x::\text{unit}) (y::\text{unit}) \equiv F$

**lemma**  $F\text{-unf}: \text{hn-ctxt } F\text{-assn } x \ y = F$

$\langle \text{proof} \rangle$

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**  $f\text{-rl}'$ :  $\text{hn-refine } (\text{hn-ctxt } A \ x \ xi * \text{hn-ctxt } (F\text{-assn}) \ dx \ dxi) (fi \ xi) (\text{hn-ctxt } A' \ x \ xi * \text{hn-ctxt } (F\text{-assn}) \ dx \ dxi) \ B \ (f\$x)$

$\langle \text{proof} \rangle$

Then we use the `Sepref` tool to synthesize an implementation of `mmap`.

**schematic-goal**  $mmap\text{-impl}$ :

**notes**  $[\text{sepref-comb-rules}] = \text{hn-refine-frame}[\text{OF } f\text{-rl}']$

**shows**  $\text{hn-refine } (\text{hn-ctxt } (\text{list-assn } A) \ l \ li * \text{hn-ctxt } (F\text{-assn}) \ dx \ dxi) \ (?c::?'c \ \text{Heap}) \ ?\Gamma' \ ?R \ mmap$

$\langle \text{proof} \rangle$

We unfold the wrapped frame

**lemmas**  $mmap\text{-impl}' = mmap\text{-impl}[\text{unfolded } F\text{-unf}]$

**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.

**concrete-definition**  $mmap\text{-impl}$  **uses**  $mmap\text{-impl}'$

**prepare-code-thms**  $mmap\text{-impl-def}$

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.

**lemma**  $mmap\text{-arity}[\text{sepref-monadify-arity}]$ :  $mmap \equiv \lambda_2 f l. \text{SP } mmap \$ (\lambda_2 x. f \$ x) \$ l$   
 $\langle \text{proof} \rangle$

**lemma**  $mmap\text{-mcomb}[\text{sepref-monadify-comb}]$ :  $mmap \$ f \$ x \equiv (\gg) \$ (\text{EVAL} \$ x) \$ (\lambda_2 x. \text{SP } mmap \$ f \$ x)$   $\langle \text{proof} \rangle$

We can massage the refinement theorem  $(\bigwedge x \ xi. \text{hn-refine } (\text{hn-ctxt } ?A \ x \ xi * ?F) \ (?fi \ xi) (\text{hn-ctxt } ?A' \ x \ xi * ?F) \ ?B \ (?f \ \$ \ x)) \implies \text{hn-refine } (\text{hn-ctxt}$

$(list-assn ?A) ?l ?li * ?F) (mmap-impl ?fi ?li) (hn-ctxt (list-assn ?A') ?l ?li * ?F) (list-assn ?B) (mmap ?f ?l)$  a bit, to get a valid combinator rule

**print-statement** *hn-refine-cons-pre*[*OF - mmap-impl.refine, sepref-prep-comb-rule, no-vars*]

**lemma** *mmap-comb-rl*[*sepref-comb-rules*]:

**assumes**  $P \Longrightarrow_t hn-ctxt (list-assn A) l li * F$

— Initial frame

**and**  $\bigwedge x xi. hn-refine (hn-ctxt A x xi * F) (fi xi) (Q x xi) B (f x)$

— Refinement of map-function

**and**  $\bigwedge x xi. Q x xi \Longrightarrow_t hn-ctxt A' x xi * F$

— Recover refinement for list-element and original frame from what map-function produced

**shows**  $hn-refine P (mmap-impl fi li) (hn-ctxt (list-assn A') l li * F) (list-assn B) (mmap (\lambda_2 x. f x) l)$

*<proof>*

#### 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.

**sepref-thm** *test-mmap*

**is**  $\lambda l. do \{ let a = op-array-of-list [True, True, False]; mmap (\lambda x. do \{ mop-list-get a (x mod 3) \}) l \}$

$:: (list-assn nat-assn)^k \rightarrow_a list-assn bool-assn$

*<proof>*

#### 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**

**context**

**includes** *bit-operations-syntax*

**begin**

**definition** *rrand* :: *uint32*  $\Rightarrow$  *uint32*

**where** *rrand* *s*  $\equiv$  (*s* \* 1103515245 + 12345) AND 0x7FFFFFFF

**end**

**definition** *rand* :: *uint32*  $\Rightarrow$  *nat*  $\Rightarrow$  (*uint32* \* *nat*) **where**

*rand* *s* *m*  $\equiv$  *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*  $\leftarrow$  *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
  <proof>

```

```

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

nat-cr-graph-imp nat-dijkstra-imp
in SML-imp module-name Dijkstra
file  $\langle$ dijkstra-export.sml $\rangle$ 

end
theory NDFS-Benchmark
imports
  Collections-Examples.Nested-DFS
  ../..../Examples/Sepref-NDFS
  Separation-Logic-Imperative-HOL.From-List-GA
begin

locale bm-fun begin

  schematic-goal succ-of-list-impl:
    notes [autoref-tyrel] =
      ty-REL[where 'a=nat $\rightarrow$ nat set and R= $\langle$ nat-rel,R $\rangle$ dflt-rm-rel for R]
      ty-REL[where 'a=nat set and R= $\langle$ nat-rel $\rangle$ list-set-rel]

    shows (?f::?'c,succ-of-list)  $\in$  ?R
     $\langle$ proof $\rangle$ 

  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= $\langle$ nat-rel $\rangle$ dflt-rs-rel for R]

    shows (?f::?'c,acc-of-list)  $\in$  ?R
     $\langle$ proof $\rangle$ 

  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= $\langle$ nat-rel $\rangle$ dflt-rs-rel]
    assumes [autoref-rules]:
      (u,u') $\in$ nat-rel

```

$(V, V') \in \langle \text{nat-rel} \rangle \text{dflt-rs-rel}$   
 $(\text{onstack}, \text{onstack}') \in \langle \text{nat-rel} \rangle \text{dflt-rs-rel}$   
 $(E, E') \in \langle \text{nat-rel} \rangle \text{slg-rel}$   
**shows**  $(\text{RETURN } (?f::?'c), \text{red-dfs } E' \text{ onstack}' V' u') \in ?R$   
 $\langle \text{proof} \rangle$

**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*] =  
 $\text{ty-REL}[\text{where } 'a = \text{nat set and } R = \langle \text{nat-rel} \rangle \text{dflt-rs-rel}]$   
**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**  $(\text{RETURN } (?f::?'c), \text{blue-dfs } E A s) \in \langle ?R \rangle \text{nres-rel}$   
 $\langle \text{proof} \rangle$

**concrete-definition** *fun-ndfs-impl* **for** *succi Ai s* **uses** *ndfs-impl-refine-aux*  
**prepare-code-thms** *fun-ndfs-impl-def*

**definition** *fun-succ-of-list*  $\equiv$   
 $\text{succ-of-list-impl } o \text{ map } (\lambda(u,v). (\text{nat-of-integer } u, \text{nat-of-integer } v))$

**definition** *fun-acc-of-list*  $\equiv$   
 $\text{acc-of-list-impl } o \text{ map } \text{nat-of-integer}$

**end**

**interpretation** *fun*: *bm-fun*  $\langle \text{proof} \rangle$

**locale** *bm-funs* **begin**

**schematic-goal** *succ-of-list-impl*:  
**notes** [*autoref-tyrel*] =  
 $\text{ty-REL}[\text{where } 'a = \text{nat} \rightarrow \text{nat set and } R = \langle \text{nat-rel}, R \rangle \text{iam-map-rel for } R]$   
 $\text{ty-REL}[\text{where } 'a = \text{nat set and } R = \langle \text{nat-rel} \rangle \text{list-set-rel}]$   
**shows**  $(?f::?'c, \text{succ-of-list}) \in ?R$   
 $\langle \text{proof} \rangle$

**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*  
 ⟨*proof*⟩

**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*  
 ⟨*proof*⟩

**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*] = *IdI*[*of s*]  
**shows** (*RETURN* (*?f*::*?'c*), *blue-dfs E A s*) ∈ ⟨*?R*⟩*nres-rel*  
 ⟨*proof*⟩

**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* ( $\lambda(u,v). (nat-of-integer\ u, nat-of-integer\ v)$ )

**definition** *funs-acc-of-list* ≡

*acc-of-list-impl* *o map* *nat-of-integer*

**end**

**interpretation** *funs*: *bm-funs* ⟨*proof*⟩

**definition** *imp-ndfs-impl*  $\equiv$  *blue-dfs-impl*  
**definition** *imp-ndfs-sz-impl*  $\equiv$  *blue-dfs-impl-sz*  
**definition** *imp-acc-of-list* *l*  $\equiv$  *From-List-GA.ias-from-list* (*map nat-of-integer l*)  
**definition** *imp-graph-of-list* *n l*  $\equiv$  *cr-graph* (*nat-of-integer n*) (*map (pairsel self 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⟩*  
**end**