The Imperative Refinement Framework

Peter Lammich

June 24, 2019

Abstract

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

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

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

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

Moreover, this entry contains a quickstart guide and a reference manual, which provide an introduction to using the IRF for Isabelle/HOL experts. It also provids 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

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 *p*attern 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 λ-. *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 be the constants defined below.

definition [*simp*]: *PROTECT2 x* (*y*::*prop*) ≡ *x*
consts *DUMMY* :: *prop*

abbreviation *PROTECT2-syn* ('(#-#')) **where** *PROTECT2-syn t* ≡ *PROTECT2 t DUMMY*

abbreviation (*input*)*ABS2* :: ('*a*⇒'*b*)⇒'*a*⇒'*b* (**binder** λ$_2$ *10*)
where *ABS2 f* ≡ (λ*x*. *PROTECT2* (*f x*) *DUMMY*)

lemma *beta*: (λ$_2$*x*. *f x*)$*x* ≡ *f x* ⟨*proof*⟩

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

definition *APP'* (**infixl** $'' *900*) **where** [*simp*, *autoref-tag-defs*]: *f*$'*a* ≡ *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, an 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* ≡ *x* — Tag to protect constant
definition [*simp*, *autoref-tag-defs*]: *UNPROTECT x* ≡ *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-type* :: '*a* ⇒ '*b itself* ⇒ *bool* (**infix** ::$_i$ *10*) **where**
[*simp*]: *c*::$_i$*I* ≡ *True*

lemma *itypeI*: *c*::$_i$*I* ⟨*proof*⟩
lemma *itypeI'*: *intf-type c TYPE*('*T*) ⟨*proof*⟩

lemma *itype-self*: (*c*::'*a*) ::$_i$ *TYPE*('*a*) ⟨*proof*⟩

definition *CTYPE-ANNOT* :: '*b* ⇒ '*a itself* ⇒ '*b* (**infix** :::$_i$ *10*) **where**
[*simp*]: *c*:::$_i$*I* ≡ *c*

Wrapper predicate for an conceptual type inference

definition *ID* :: '*a* ⇒ '*a* ⇒ '*c itself* ⇒ *bool*
where [*simp*]: *ID t t' T* ≡ *t*=*t'*

Conceptual Typing Rules

lemma *ID-unfold-vars*: *ID x y T* ⟹ *x*≡*y* ⟨*proof*⟩

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

lemma *pat-rule*:
$[\![\ p{\equiv}p';\ ID\ p'\ t'\ T\ ]\!] \Longrightarrow ID\ p\ t'\ T\ \langle proof \rangle$

lemma *app-rule*:
$[\![\ ID\ f\ f'\ TYPE('a{\Rightarrow}'b);\ ID\ x\ x'\ TYPE('a)]\!] \Longrightarrow ID\ (f\$x)\ (f'\$x')\ TYPE('b)$
$\langle proof \rangle$

lemma *app'-rule*:
$[\![\ ID\ f\ f'\ TYPE('a{\Rightarrow}'b);\ ID\ x\ x'\ TYPE('a)]\!] \Longrightarrow ID\ (f\$'x)\ (f'\$x')\ TYPE('b)$
$\langle proof \rangle$

lemma *abs-rule*:
$[\![\ \bigwedge x\ x'.\ ID\ x\ x'\ TYPE('a) \Longrightarrow ID\ (t\ x)\ (t'\ x\ x')\ TYPE('b)\ ]\!] \Longrightarrow$
$ID\ (\lambda_2 x.\ t\ x)\ (\lambda_2 x'.\ t'\ x'\ x')\ TYPE('a{\Rightarrow}'b)$
$\langle proof \rangle$

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*: $ID\ (PR\text{-}CONST\ x)\ t\ T \Longrightarrow ID\ (UNPROTECT\ x)\ t\ T$
$\langle proof \rangle$

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*)) ::$_i$ *TYPE*(*nat*) ⟨*proof*⟩
lemma *id-int-const*[*id-rules*]: (*PR-CONST* (*a*::*int*)) ::$_i$ *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** :::$_i$ *10*)
no-notation *CONST-INTF* (**infixr** ::$_i$ *10*)

In this theory, we define the basic concept of refinement from a nondeterministic 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* $P \Longrightarrow$ *is-pure* (*hn-ctxt* P)
⟨*proof*⟩

definition *the-pure* $P \equiv$ *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* $R \Longrightarrow$ *pure* (*the-pure* R) $= R$
⟨*proof*⟩

lemma *is-pure-conv*: *is-pure* $R \longleftrightarrow (\exists R'.\ R =$ *pure* $R')$
⟨*proof*⟩

lemma *is-pure-the-pure-id-eq*[*simp*]: *is-pure* $R \Longrightarrow$ *the-pure* $R =$ *Id* $\longleftrightarrow$ R=*pure Id*
⟨*proof*⟩

lemma *is-pure-iff-pure-assn*: *is-pure* $P = (\forall x\ x'.$ *is-pure-assn* $(P\ x\ x'))$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

1.2.2 Constraints in Refinement Relations

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

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

abbreviation *rdom* $R \equiv$ *Collect* (*rdomp* R)

lemma *rdomp-ctxt*[*simp*]: *rdomp* (*hn-ctxt* R) = *rdomp* R
⟨*proof*⟩

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

lemma *rdom-pure*[*simp*]: *rdom* (*pure* R) = *Range* R
⟨*proof*⟩

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

1.2.3 Heap-Nres Refinement Calculus

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

abstract result from m

definition *hn-refine* $\Gamma\ c\ \Gamma'\ R\ m \equiv$ *nofail* $m \longrightarrow$
$<\Gamma>\ c\ <\lambda r.\ \Gamma' * (\exists_A x.\ R\ x\ r * \uparrow(RETURN\ x \leq m))>_t$

⟨*ML*⟩

lemma *hn-refineI*[*intro?*]:
assumes *nofail* m
$\Longrightarrow <\Gamma>\ c\ <\lambda r.\ \Gamma' * (\exists_A x.\ R\ x\ r * \uparrow(RETURN\ x \leq m))>_t$
shows *hn-refine* $\Gamma\ c\ \Gamma'\ R\ m$
⟨*proof*⟩

lemma *hn-refineD*:
assumes *hn-refine* $\Gamma\ c\ \Gamma'\ R\ m$
assumes *nofail* m
shows $<\Gamma>\ c\ <\lambda r.\ \Gamma' * (\exists_A x.\ R\ x\ r * \uparrow(RETURN\ x \leq m))>_t$
⟨*proof*⟩

lemma *hn-refine-preI*:
assumes $\bigwedge h.\ h \models \Gamma \Longrightarrow$ *hn-refine* $\Gamma\ c\ \Gamma'\ R\ a$
shows *hn-refine* $\Gamma\ c\ \Gamma'\ R\ a$
⟨*proof*⟩

lemma *hn-refine-nofailI*:
assumes *nofail* $a \Longrightarrow$ *hn-refine* $\Gamma\ c\ \Gamma'\ R\ a$
shows *hn-refine* $\Gamma\ c\ \Gamma'\ R\ a$
⟨*proof*⟩

lemma *hn-refine-false*[*simp*]: *hn-refine false* $c\ \Gamma'\ R\ m$
⟨*proof*⟩

lemma *hn-refine-fail*[*simp*]: *hn-refine* $\Gamma\ c\ \Gamma'\ R$ *FAIL*
⟨*proof*⟩

lemma *hn-refine-frame*:
assumes *hn-refine* $P'\ c\ Q'\ R\ m$
assumes $P \Longrightarrow_t F * P'$
shows *hn-refine* $P\ c\ (F * Q')\ R\ m$
⟨*proof*⟩

lemma *hn-refine-cons*:
assumes I: $P \Longrightarrow_t P'$
assumes R: *hn-refine* $P'\ c\ Q\ R\ m$
assumes I': $Q \Longrightarrow_t Q'$
assumes R': $\bigwedge x\ y.\ R\ x\ y \Longrightarrow_t R'\ x\ y$
shows *hn-refine* $P\ c\ Q'\ R'\ m$
⟨*proof*⟩

lemma *hn-refine-cons-pre*:
assumes *I*: $P \Longrightarrow_t P'$
assumes *R*: *hn-refine* P' *c* *Q* *R* *m*
shows *hn-refine* *P* *c* *Q* *R* *m*
⟨*proof*⟩

lemma *hn-refine-cons-post*:
assumes *R*: *hn-refine* *P* *c* *Q* *R* *m*
assumes *I*: $Q \Longrightarrow_t Q'$
shows *hn-refine* *P* *c* Q' *R* *m*
⟨*proof*⟩

lemma *hn-refine-cons-res*:
⟦ *hn-refine* Γ *f* Γ′ *R* *g*; $\bigwedge a\ c.\ R\ a\ c \Longrightarrow_t R'\ a\ c$ ⟧ ⟹ *hn-refine* Γ *f* Γ′ R' *g*
⟨*proof*⟩

lemma *hn-refine-ref*:
assumes *LE*: $m \leq m'$
assumes *R*: *hn-refine* *P* *c* *Q* *R* *m*
shows *hn-refine* *P* *c* *Q* *R* m'
⟨*proof*⟩

lemma *hn-refine-cons-complete*:
assumes *I*: $P \Longrightarrow_t P'$
assumes *R*: *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 *hn-refine* *P* *c* Q' R' m'
⟨*proof*⟩

lemma *hn-refine-augment-res*:
assumes *A*: *hn-refine* Γ *f* Γ′ *R* *g*
assumes *B*: $g \leq_n$ *SPEC* Φ
shows *hn-refine* Γ *f* Γ′ $(\lambda a\ c.\ R\ a\ c * \uparrow(\Phi\ a))$ *g*
⟨*proof*⟩

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*⇒'*c1*⇒*assn*) ⇒ ('*a2*⇒'*c2*⇒*assn*)
⇒ '*a1*∗'*a2* ⇒ '*c1*∗'*c2* ⇒ *assn* **where**
prod-assn P1 P2 a c ≡ *case* (*a*,*c*) *of* ((*a1*,*a2*),(*c1*,*c2*)) ⇒
P1 a1 c1 ∗ *P2 a2 c2*

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

lemma *prod-assn-pure-conv*[*simp*]: *prod-assn* (*pure R1*) (*pure R2*) = *pure* (*R1* $\times_r$ *R2*)
⟨*proof*⟩

lemma *prod-assn-pair-conv*[*simp*]:
prod-assn A B (*a1*,*b1*) (*a2*,*b2*) = *A a1 a2* ∗ *B b1 b2*
⟨*proof*⟩

lemma *prod-assn-true*[*simp*]: *prod-assn* (λ- -. *true*) (λ- -. *true*) = (λ- -. *true*)
⟨*proof*⟩

1.2.5 Convenience Lemmas

lemma *hn-refine-guessI*:
assumes *hn-refine P f P′ R f′*
assumes *f*=*f-conc*
shows *hn-refine P f-conc P′ R f′*
— To prove a refinement, first synthesize one, and then prove equality
⟨*proof*⟩

lemma *imp-correctI*:
assumes *R*: *hn-refine* Γ *c* Γ′ *R a*
assumes *C*: *a* ≤ *SPEC* Φ
shows <Γ> *c* <λ*r′*. ∃$_A$*r*. Γ′ ∗ *R r r′* ∗ ↑(Φ *r*)>$_t$
⟨*proof*⟩

lemma *hnr-pre-ex-conv*:
shows *hn-refine* (∃$_A$*x*. Γ *x*) *c* Γ′ *R a* ⟷ (∀*x*. *hn-refine* (Γ *x*) *c* Γ′ *R a*)
⟨*proof*⟩

lemma *hnr-pre-pure-conv*:
shows *hn-refine* (Γ ∗ ↑*P*) *c* Γ′ *R a* ⟷ (*P* ⟶ *hn-refine* Γ *c* Γ′ *R a*)
⟨*proof*⟩

lemma *hn-refine-split-post*:
assumes *hn-refine* Γ *c* Γ′ *R a*
shows *hn-refine* Γ *c* (Γ′ ∨$_A$ Γ″) *R a*
⟨*proof*⟩

lemma *hn-refine-post-other*:
assumes *hn-refine* Γ *c* Γ″ *R a*
shows *hn-refine* Γ *c* (Γ′ ∨$_A$ Γ″) *R a*
⟨*proof*⟩

Return

lemma *hnr-RETURN-pass*:
hn-refine (*hn-ctxt R x p*) (*return p*) (*hn-invalid R x p*) *R* (*RETURN x*)
— Pass on a value from the heap as return value

⟨*proof*⟩

lemma *hnr-RETURN-pure*:
assumes (*c*,*a*)∈*R*
shows *hn-refine emp* (*return c*) *emp* (*pure R*) (*RETURN a*)
— Return pure value
⟨*proof*⟩

Assertion

lemma *hnr-FAIL*[*simp*, *intro*!]: *hn-refine* Γ *c* Γ′ *R FAIL*
⟨*proof*⟩

lemma *hnr-ASSERT*:
assumes Φ ⟹ *hn-refine* Γ *c* Γ′ *R c*′
shows *hn-refine* Γ *c* Γ′ *R* (*do* { *ASSERT* Φ; *c*′})
⟨*proof*⟩

Bind

lemma *bind-det-aux*: ⟦ *RETURN x* ≤ *m*; *RETURN y* ≤ *f x* ⟧ ⟹ *RETURN y* ≤ *m* ⤜ *f*
⟨*proof*⟩

lemma *hnr-bind*:
assumes *D1*: *hn-refine* Γ *m*′ Γ*1 Rh m*
assumes *D2*:
⋀*x x*′. *RETURN x* ≤ *m* ⟹ *hn-refine* (Γ*1* ∗ *hn-ctxt Rh x x*′) (*f*′ *x*′) (Γ*2 x x*′) *R* (*f x*)
assumes *IMP*: ⋀*x x*′. Γ*2 x x*′ ⟹$_t$ Γ′ ∗ *hn-ctxt Rx x x*′
shows *hn-refine* Γ (*m*′⤜*f*′) Γ′ *R* (*m*⤜*f*)
⟨*proof*⟩

Recursion

definition *hn-rel P m* ≡ λ*r*. ∃$_A$*x*. *P x r* ∗ ↑(*RETURN x* ≤ *m*)

lemma *hn-refine-alt*: *hn-refine Fpre c Fpost P m* ≡ *nofail m* ⟶
<*Fpre*> *c* <λ*r*. *hn-rel P m r* ∗ *Fpost*>$_t$
⟨*proof*⟩

lemma *wit-swap-forall*:
assumes *W*: <*P*> *c* <λ-. *true*>
assumes *T*: (∀ *x*. *A x* ⟶ <*P*> *c* <*Q x*>)
shows <*P*> *c* <λ*r*. ¬$_A$ (∃$_A$*x*. ↑(*A x*) ∗ ¬$_A$ *Q x r*)>
⟨*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*: ∀*f*∈*A*. *nofail* (*f x*) ⟶ <*P*> *c* <λ*r*. *hn-rel Ry* (*f x*) *r* ∗ *F*>
assumes *NF*: *nofail* (*INF f*∈*A*. *f x*)
shows <*P*> *c* <λ*r*. *hn-rel Ry* (*INF f*∈*A*. *f x*) *r* ∗ *F*>
⟨*proof*⟩

lemma *hn-admissible′*:
assumes *PREC*: *precise Ry*
assumes *E*: ∀*f*∈*A*. *nofail* (*f x*) ⟶ <*P*> *c* <λ*r*. *hn-rel Ry* (*f x*) *r* ∗ *F*>$_t$
assumes *NF*: *nofail* (*INF f*∈*A*. *f x*)
shows <*P*> *c* <λ*r*. *hn-rel Ry* (*INF f*∈*A*. *f x*) *r* ∗ *F*>$_t$
⟨*proof*⟩

lemma *hnr-RECT-old*:
assumes *S*: ⋀*cf af ax px*. ⟦
⋀*ax px*. *hn-refine* (*hn-ctxt Rx ax px* ∗ *F*) (*cf px*) (*F′ ax px*) *Ry* (*af ax*)⟧
⟹ *hn-refine* (*hn-ctxt Rx ax px* ∗ *F*) (*cB cf px*) (*F′ ax px*) *Ry* (*aB af ax*)
assumes *M*: (⋀*x*. *mono-Heap* (λ*f*. *cB f x*))
assumes *PREC*: *precise Ry*
shows *hn-refine*
(*hn-ctxt Rx ax px* ∗ *F*) (*heap.fixp-fun cB px*) (*F′ ax px*) *Ry* (*RECT aB ax*)
⟨*proof*⟩

lemma *hnr-RECT*:
assumes *S*: ⋀*cf af ax px*. ⟦
⋀*ax px*. *hn-refine* (*hn-ctxt Rx ax px* ∗ *F*) (*cf px*) (*F′ ax px*) *Ry* (*af ax*)⟧
⟹ *hn-refine* (*hn-ctxt Rx ax px* ∗ *F*) (*cB cf px*) (*F′ ax px*) *Ry* (*aB af ax*)
assumes *M*: (⋀*x*. *mono-Heap* (λ*f*. *cB f x*))
shows *hn-refine*
(*hn-ctxt Rx ax px* ∗ *F*) (*heap.fixp-fun cB px*) (*F′ ax px*) *Ry* (*RECT aB ax*)
⟨*proof*⟩

lemma *hnr-If*:
assumes *P*: Γ ⟹$_t$ Γ*1* ∗ *hn-val bool-rel a a′*
assumes *RT*: *a* ⟹ *hn-refine* (Γ*1* ∗ *hn-val bool-rel a a′*) *b′* Γ*2b R b*
assumes *RE*: ¬*a* ⟹ *hn-refine* (Γ*1* ∗ *hn-val bool-rel a a′*) *c′* Γ*2c R c*
assumes *IMP*: Γ*2b* ∨$_A$ Γ*2c* ⟹$_t$ Γ′
shows *hn-refine* Γ (*if a′ then b′ else c′*) Γ′ *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* ≡ *x*
lemma *SP-cong*[*cong*]: *SP x* ≡ *SP x* ⟨*proof*⟩
lemma *PR-CONST-cong*[*cong*]: *PR-CONST x* ≡ *PR-CONST x* ⟨*proof*⟩

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

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

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

lemma *monadify-simps*:
Refine-Basic.bind$(*RETURN*$*x*)$(λ$_2$*x*. *f x*) = *f x*
EVAL$*x* ≡ *RETURN*$*x*
⟨*proof*⟩

definition [*simp*]: *PASS* ≡ *RETURN*
— Pass on value, invalidating old one

lemma *remove-pass-simps*:
Refine-Basic.bind$(*PASS*$*x*)$(λ$_2$*x*. *f x*) ≡ *f x*
Refine-Basic.bind$*m*$(λ$_2$*x*. *PASS*$*x*) ≡ *m*
⟨*proof*⟩

definition *COPY* :: *'a* ⇒ *'a*
— Marks required copying of parameter
where [*simp*]: *COPY x* ≡ *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 ≡ λ$_2$*x*. *SP RETURN*$*x*
RECT ≡ λ$_2$*B x*. *SP RECT*$(λ$_2$*D x*. *B*$(λ$_2$*x*. *RCALL*$*D*$*x*)$*x*)$*x*
case-list ≡ λ$_2$*fn fc l*. *SP case-list*$*fn*$(λ$_2$*x xs*. *fc*$*x*$*xs*)$*l*
case-prod ≡ λ$_2$*fp p*. *SP case-prod*$(λ$_2$*a b*. *fp*$*a*$*b*)$*p*
case-option ≡ λ$_2$*fn fs ov*. *SP case-option*$*fn*$(λ$_2$*x*. *fs*$*x*)$*ov*
If ≡ λ$_2$*b t e*. *SP If*$*b*$*t*$*e*
Let ≡ λ$_2$*x f*. *SP Let*$*x*$(λ$_2$*x*. *f*$*x*)
⟨*proof*⟩

lemma *dflt-comb*[*sepref-monadify-comb*]:
⋀*B x*. *RECT*$*B*$*x* ≡ *Refine-Basic.bind*$(*EVAL*$*x*)$(λ$_2$*x*. *SP* (*RECT*$*B*$*x*))
⋀*D x*. *RCALL*$*D*$*x* ≡ *Refine-Basic.bind*$(*EVAL*$*x*)$(λ$_2$*x*. *SP* (*RCALL*$*D*$*x*))
⋀*fn fc l*. *case-list*$*fn*$*fc*$*l* ≡ *Refine-Basic.bind*$(*EVAL*$*l*)$(λ$_2$*l*. (*SP case-list*$*fn*$*fc*$*l*))

$\bigwedge fp\ p.\ case\text{-}prod\$fp\$p \equiv Refine\text{-}Basic.bind\$(EVAL\$p)\$(\lambda_2 p.\ (SP\ case\text{-}prod\$fp\$p))$
$\bigwedge fn\ fs\ ov.\ case\text{-}option\$fn\$fs\$ov$
$\equiv Refine\text{-}Basic.bind\$(EVAL\$ov)\$(\lambda_2 ov.\ (SP\ case\text{-}option\$fn\$fs\$ov))$
$\bigwedge b\ t\ e.\ If\$b\$t\$e \equiv Refine\text{-}Basic.bind\$(EVAL\$b)\$(\lambda_2 b.\ (SP\ If\$b\$t\$e))$
$\bigwedge x.\ RETURN\$x \equiv Refine\text{-}Basic.bind\$(EVAL\$x)\$(\lambda_2 x.\ SP\ (RETURN\$x))$
$\bigwedge x\ f.\ Let\$x\$f \equiv Refine\text{-}Basic.bind\$(EVAL\$x)\$(\lambda_2 x.\ (SP\ Let\$x\$f))$
⟨*proof*⟩

lemma *dflt-plain-comb*[*sepref-monadify-comb*]:
$EVAL\$(If\$b\$t\$e) \equiv Refine\text{-}Basic.bind\$(EVAL\$b)\$(\lambda_2 b.\ If\$b\$(EVAL\$t)\$(EVAL\$e))$
$EVAL\$(case\text{-}list\$fn\$(\lambda_2 x\ xs.\ fc\ x\ xs)\$l) \equiv$
$Refine\text{-}Basic.bind\$(EVAL\$l)\$(\lambda_2 l.\ case\text{-}list\$(EVAL\$fn)\$(\lambda_2 x\ xs.\ EVAL\$(fc\ x\ xs))\$l)$
$EVAL\$(case\text{-}prod\$(\lambda_2 a\ b.\ fp\ a\ b)\$p) \equiv$
$Refine\text{-}Basic.bind\$(EVAL\$p)\$(\lambda_2 p.\ case\text{-}prod\$(\lambda_2 a\ b.\ EVAL\$(fp\ a\ b))\$p)$
$EVAL\$(case\text{-}option\$fn\$(\lambda_2 x.\ fs\ x)\$ov) \equiv$
$Refine\text{-}Basic.bind\$(EVAL\$ov)\$(\lambda_2 ov.\ case\text{-}option\$(EVAL\$fn)\$(\lambda_2 x.\ EVAL\$(fs\ x))\$ov)$
$EVAL\ \$\ (Let\ \$\ v\ \$\ (\lambda_2 x.\ f\ x)) \equiv (\gg\!=)\ \$\ (EVAL\ \$\ v)\ \$\ (\lambda_2 x.\ EVAL\ \$\ (f\ x))$
⟨*proof*⟩

lemma *evalcomb-PR-CONST*[*sepref-monadify-comb*]:
$EVAL\$(PR\text{-}CONST\ x) \equiv SP\ (RETURN\$(PR\text{-}CONST\ x))$
⟨*proof*⟩

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 $PROP\ P \Longrightarrow PROP\ (CONSTRAINT\text{-}SLOT\ (Trueprop\ True)) \Longrightarrow PROP\ Q$
shows $PROP\ (CONSTRAINT\text{-}SLOT\ (PROP\ P)) \Longrightarrow PROP\ Q$
⟨*proof*⟩

lemma *insert-slot-rl2*:
assumes $PROP\ P \Longrightarrow PROP\ (CONSTRAINT\text{-}SLOT\ S) \Longrightarrow PROP\ Q$
shows $PROP\ (CONSTRAINT\text{-}SLOT\ (PROP\ S\ \&\&\&\ PROP\ P)) \Longrightarrow PROP\ Q$
⟨*proof*⟩

lemma *remove-slot*: $PROP\ (CONSTRAINT\text{-}SLOT\ (Trueprop\ True))$
⟨*proof*⟩

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

lemma *CONSTRAINT-D*:
assumes *CONSTRAINT* (*P*::*'a* => *bool*) *x*
shows *P x*
⟨*proof*⟩

lemma *CONSTRAINT-I*:
assumes *P x*
shows *CONSTRAINT* (*P*::*'a* => *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*⇒*bool*) ⇒ *'a* ⇒ *bool* **where** [*simp*]: *CN-FALSE P x* ≡ *False*
lemma *CN-FALSEI*: *CN-FALSE P x* ⟹ *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* ⇒ *'c* ⇒ *assn*) ⇒ (*'a* ⇒ *'c* ⇒ *assn*) ⇒ *'a* ⇒ *'c* ⇒ *assn*
where *mismatch-assn R1 R2 x y* ≡ *R1 x y* $\vee_A$ *R2 x y*

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

lemma *recover-pure-aux*: *CONSTRAINT is-pure R* ⟹ *hn-invalid R x y* ⟹$_t$ *hn-ctxt R x y*
⟨*proof*⟩

lemma *frame-thms*:
P ⟹$_t$ *P*
P⟹$_t$*P′* ⟹ *F*⟹$_t$*F′* ⟹ *F*∗*P* ⟹$_t$ *F′*∗*P′*
hn-ctxt R x y ⟹$_t$ *hn-invalid R x y*
hn-ctxt R x y ⟹$_t$ *hn-ctxt* (λ- -. *true*) *x y*
CONSTRAINT is-pure R ⟹ *hn-invalid R x y* ⟹$_t$ *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*⟹$_t$*P* ⟨*proof*⟩

lemma *frame-rem2*: *F* ⟹$_t$ *F′* ⟹ *F* ∗ *hn-ctxt A x y* ⟹$_t$ *F′* ∗ *hn-ctxt A x y*
⟨*proof*⟩

lemma *frame-rem3*: *F* ⟹$_t$ *F′* ⟹ *F* ∗ *hn-ctxt A x y* ⟹$_t$ *F′*
⟨*proof*⟩

lemma *frame-rem4*: *P* ⟹$_t$ *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* ∨$_A$ *C* ⟹$_A$ *E*; *B* ∨$_A$ *D* ⟹$_A$ *F* ⟧ ⟹ *A*∗*B* ∨$_A$ *C*∗*D* ⟹$_A$ *E*∗*F*
⟨*proof*⟩

lemma *entt-disj-star-mono*:
⟦ *A* ∨$_A$ *C* ⟹$_t$ *E*; *B* ∨$_A$ *D* ⟹$_t$ *F* ⟧ ⟹ *A*∗*B* ∨$_A$ *C*∗*D* ⟹$_t$ *E*∗*F*
⟨*proof*⟩

lemma *hn-merge1*:

$F \vee_A F \Longrightarrow_t F$
$[\![\textit{hn-ctxt R1 } x\ x' \vee_A \textit{hn-ctxt R2 } x\ x' \Longrightarrow_t \textit{hn-ctxt R } x\ x';\ Fl \vee_A Fr \Longrightarrow_t F\ ]\!]$
$\Longrightarrow Fl * \textit{hn-ctxt R1 } x\ x' \vee_A Fr * \textit{hn-ctxt R2 } x\ x' \Longrightarrow_t F * \textit{hn-ctxt R } x\ x'$
⟨*proof*⟩

lemma *hn-merge2*:
hn-invalid R x x' $\vee_A$ *hn-ctxt R x x'* $\Longrightarrow_t$ *hn-invalid R x x'*
hn-ctxt R x x' $\vee_A$ *hn-invalid R x x'* $\Longrightarrow_t$ *hn-invalid R x x'*
⟨*proof*⟩

lemma *invalid-assn-mono*: *hn-ctxt A x y* $\Longrightarrow_t$ *hn-ctxt B x y*
$\Longrightarrow$ *hn-invalid A x y* $\Longrightarrow_t$ *hn-invalid B x y*
⟨*proof*⟩

lemma *hn-merge3*:
$[\![$*NO-MATCH* (*hn-invalid XX*) *R2*; *hn-ctxt R1 x x'* $\vee_A$ *hn-ctxt R2 x x'* $\Longrightarrow_t$ *hn-ctxt Rm x x'*$]\!] \Longrightarrow$ *hn-invalid R1 x x'* $\vee_A$ *hn-ctxt R2 x x'* $\Longrightarrow_t$ *hn-invalid Rm x x'*
$[\![$*NO-MATCH* (*hn-invalid XX*) *R1*; *hn-ctxt R1 x x'* $\vee_A$ *hn-ctxt R2 x x'* $\Longrightarrow_t$ *hn-ctxt Rm x x'*$]\!] \Longrightarrow$ *hn-ctxt R1 x x'* $\vee_A$ *hn-invalid R2 x x'* $\Longrightarrow_t$ *hn-invalid Rm x x'*
⟨*proof*⟩

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

named-theorems *sepref-frame-merge-rules* ‹*Sepref*: *Additional merge rules*›

lemma *hn-merge-mismatch*: *hn-ctxt R1 x x'* $\vee_A$ *hn-ctxt R2 x x'* $\Longrightarrow_t$ *hn-mismatch R1 R2 x x'*
⟨*proof*⟩

lemma *is-merge*: $P1 \vee_A P2 \Longrightarrow_t P \Longrightarrow P1 \vee_A P2 \Longrightarrow_t P$ ⟨*proof*⟩

lemma *merge-mono*: $[\![A \Longrightarrow_t A';\ B \Longrightarrow_t B';\ A' \vee_A B' \Longrightarrow_t C]\!] \Longrightarrow A \vee_A B \Longrightarrow_t C$
⟨*proof*⟩

Apply forward rule on left or right side of merge

lemma *gen-merge-cons1*: $[\![A \Longrightarrow_t A';\ A' \vee_A B \Longrightarrow_t C]\!] \Longrightarrow A \vee_A B \Longrightarrow_t C$
⟨*proof*⟩

lemma *gen-merge-cons2*: $[\![B \Longrightarrow_t B';\ A \vee_A B' \Longrightarrow_t C]\!] \Longrightarrow A \vee_A B \Longrightarrow_t C$
⟨*proof*⟩

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

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

definition *RECOVER-PURE P Q* ≡ *P* ⟹$_t$ *Q*

lemma *recover-pure*:
RECOVER-PURE emp emp
⟦*RECOVER-PURE P2 Q2*; *RECOVER-PURE P1 Q1*⟧ ⟹ *RECOVER-PURE* (*P1*∗*P2*) (*Q1*∗*Q2*)
CONSTRAINT is-pure R ⟹ *RECOVER-PURE* (*hn-invalid R x y*) (*hn-ctxt R x y*)
RECOVER-PURE (*hn-ctxt R x y*) (*hn-ctxt R x y*)
⟨*proof*⟩

lemma *recover-pure-triv*:
RECOVER-PURE P P
⟨*proof*⟩

Weakening the postcondition by converting *invalid-assn* to λ- -. *true*

definition *WEAKEN-HNR-POST* Γ Γ′ Γ″ ≡ (∃ *h*. *h*⊨Γ) ⟶ (Γ″ ⟹$_t$ Γ′)

lemma *weaken-hnr-postI*:
assumes *WEAKEN-HNR-POST* Γ Γ″ Γ′
assumes *hn-refine* Γ *c* Γ′ *R a*
shows *hn-refine* Γ *c* Γ″ *R a*
⟨*proof*⟩

lemma *weaken-hnr-post-triv*: *WEAKEN-HNR-POST* Γ *P P*
⟨*proof*⟩

lemma *weaken-hnr-post*:
⟦*WEAKEN-HNR-POST* Γ *P P*′; *WEAKEN-HNR-POST* Γ′ *Q Q*′⟧ ⟹ *WEAKEN-HNR-POST* (Γ∗Γ′) (*P*∗*Q*) (*P*′∗*Q*′)
WEAKEN-HNR-POST (*hn-ctxt R x y*) (*hn-ctxt R x y*) (*hn-ctxt R x y*)
WEAKEN-HNR-POST (*hn-ctxt R x y*) (*hn-invalid R x y*) (*hn-ctxt* (λ- -. *true*) *x y*)
⟨*proof*⟩

lemma *reorder-enttI*:
assumes *A*∗*true* = *C*∗*true*
assumes *B*∗*true* = *D*∗*true*
shows (*A*⟹$_t$*B*) ≡ (*C*⟹$_t$*D*)
⟨*proof*⟩

lemma *merge-sat1*: (*A*∨$_A$*A*′ ⟹$_t$ *Am*) ⟹ (*A*∨$_A$*Am* ⟹$_t$ *Am*)
⟨*proof*⟩

lemma *merge-sat2*: $(A \vee_A A' \Longrightarrow_t Am) \Longrightarrow (Am \vee_A A' \Longrightarrow_t Am)$
⟨*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* :: $('a \Rightarrow - \Rightarrow assn) \Rightarrow 'b$ *itself* $\Rightarrow$ *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* $(R :: 'a \Rightarrow - \Rightarrow assn)$ *TYPE*$('a)$ ⟨*proof*⟩

1.5.2 Function Refinement with Precondition

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

$\Rightarrow ((\prime a \Rightarrow \prime b) \times (\prime c \Rightarrow \prime d))$ *set*
([-]$_f$ - $\rightarrow$ - [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 *freft* (- $\rightarrow_f$ - [60,60] 60) **where** $R \rightarrow_f S \equiv ([\lambda\text{-}.\ True]_f\ R \rightarrow S)$

lemma *rel2p-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*)))
⟨*proof*⟩

lemma *fref-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 $(f,g) \in [Q]_f\ R' \rightarrow S'$
⟨*proof*⟩

lemmas *fref-cons'* $=$ *fref-cons*[*OF - - order-refl order-refl*]

lemma *frefI*[*intro?*]:
assumes $\bigwedge x\ y.\ [\![P\ y;\ (x,y) \in R]\!] \Longrightarrow (f\ x,\ g\ y) \in S$
shows $(f,g) \in$ *fref P R S*
⟨*proof*⟩

lemma *fref-ncI*: $(f,g) \in R \rightarrow S \Longrightarrow (f,g) \in R \rightarrow_f S$
⟨*proof*⟩

lemma *frefD*:
assumes $(f,g) \in$ *fref P R S*
shows $[\![P\ y;\ (x,y) \in R]\!] \Longrightarrow (f\ x,\ g\ y) \in S$
⟨*proof*⟩

lemma *fref-ncD*: $(f,g) \in R \rightarrow_f S \Longrightarrow (f,g) \in R \rightarrow S$
⟨*proof*⟩

lemma *fref-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*)
⟨*proof*⟩

lemma *fref-compI'*:
$[\![\ (f,g) \in$ *fref P R1 R2*; $(g,h) \in$ *fref Q S1 S2* $]\!]$
$\Longrightarrow (f,h) \in$ *fref* ($\lambda x.\ Q\ x \wedge (\forall y.\ (y,x) \in S1 \longrightarrow P\ y)$) (*R1 O S1*) (*R2 O S2*)
⟨*proof*⟩

lemma *fref-unit-conv*:
($\lambda\text{-}.\ c$, $\lambda\text{-}.\ a$) $\in$ *fref P unit-rel S* $\longleftrightarrow$ ($P\ () \longrightarrow (c,a) \in S$)

⟨*proof*⟩

lemma *fref-uncurry-conv*:
(*uncurry c*, *uncurry a*) ∈ *fref P* (*R1*×$_r$*R2*) *S*
⟷ (∀ *x1 y1 x2 y2*. *P* (*y1*,*y2*) ⟶ (*x1*,*y1*)∈*R1* ⟶ (*x2*,*y2*)∈*R2* ⟶ (*c x1 x2*, *a y1 y2*) ∈ *S*)
⟨*proof*⟩

lemma *fref-mono*: ⟦ ⋀*x*. *P'* *x* ⟹ *P* *x*; *R'* ⊆ *R*; *S* ⊆ *S'* ⟧
⟹ *fref P R S* ⊆ *fref P' R' S'*
⟨*proof*⟩

lemma *fref-composeI*:
assumes *FR1*: (*f*,*g*)∈*fref P R1 R2*
assumes *FR2*: (*g*,*h*)∈*fref Q S1 S2*
assumes *C1*: ⋀*x*. *P'* *x* ⟹ *Q* *x*
assumes *C2*: ⋀*x y*. ⟦*P'* *x*; (*y*,*x*)∈*S1*⟧ ⟹ *P* *y*
assumes *R1*: *R'* ⊆ *R1* *O* *S1*
assumes *R2*: *R2* *O* *S2* ⊆ *S'*
assumes *FH*: *f'*=*f* *h'*=*h*
shows (*f'*,*h'*) ∈ *fref P' R' S'*
⟨*proof*⟩

lemma *fref-triv*: *A*⊆*Id* ⟹ (*f*,*f*)∈[*P*]$_f$ *A* → *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*
::
('*a* ⇒ *bool*)
⇒ (('*a* ⇒ '*ai* ⇒ *assn*) × ('*a* ⇒ '*ai* ⇒ *assn*))
⇒ ('*b* ⇒ '*bi* ⇒ *assn*)
⇒ (('*ai* ⇒ '*bi Heap*) × ('*a*⇒'*b nres*)) *set*
([-]$_a$ - → - [*0*,*60*,*60*] *60*)
where
[*P*]$_a$ *RS* → *T* ≡ { (*f*,*g*) . ∀ *c a*. *P a* ⟶ *hn-refine* (*fst RS a c*) (*f c*) (*snd RS a c*) *T* (*g a*)}

abbreviation *hfreft* (- →$_a$ - [*60*,*60*] *60*) **where** *RS* →$_a$ *T* ≡ ([λ-. *True*]$_a$ *RS* → *T*)

lemma *hfrefI*[*intro?*]:
assumes ⋀*c a*. *P a* ⟹ *hn-refine* (*fst RS a c*) (*f c*) (*snd RS a c*) *T* (*g a*)
shows (*f*,*g*)∈*hfref P RS T*
⟨*proof*⟩

lemma *hfrefD*:
assumes $(f,g)\in hfref\ P\ RS\ T$
shows $\bigwedge c\ a.\ P\ a \Longrightarrow hn\text{-}refine\ (fst\ RS\ a\ c)\ (f\ c)\ (snd\ RS\ a\ c)\ T\ (g\ a)$
⟨*proof*⟩

lemma *hfref-to-ASSERT-conv*:
$NO\text{-}MATCH\ (\lambda\text{-}.\ True)\ P \Longrightarrow (a,b)\in[P]_a\ R \to S \longleftrightarrow (a,\lambda x.\ ASSERT\ (P\ x) \gg b\ x) \in R \to_a S$
⟨*proof*⟩

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

abbreviation *hfkeep*
:: $('a \Rightarrow 'b \Rightarrow assn) \Rightarrow ('a \Rightarrow 'b \Rightarrow assn)\times('a \Rightarrow 'b \Rightarrow assn)$
$((-^k)\ [1000]\ 999)$
where $R^k \equiv hf\text{-}pres\ R\ True$
abbreviation *hfdrop*
:: $('a \Rightarrow 'b \Rightarrow assn) \Rightarrow ('a \Rightarrow 'b \Rightarrow assn)\times('a \Rightarrow 'b \Rightarrow assn)$
$((-^d)\ [1000]\ 999)$
where $R^d \equiv hf\text{-}pres\ R\ False$

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

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

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

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* ::
$(('a \Rightarrow 'b \Rightarrow assn)\times('a \Rightarrow 'b \Rightarrow assn))$
$\Rightarrow (('c \Rightarrow 'd \Rightarrow assn)\times('c \Rightarrow 'd \Rightarrow assn))$
$\Rightarrow ((('a\times'c) \Rightarrow ('b \times 'd) \Rightarrow assn) \times (('a\times'c) \Rightarrow ('b \times 'd) \Rightarrow assn))$
(**infixl** $*_a$ *65*)

where $RR *_a SS \equiv (prod\text{-}assn\ (fst\ RR)\ (fst\ SS),\ prod\text{-}assn\ (snd\ RR)\ (snd\ SS))$

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

Conversion from fref to hfref

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

Conversion from hfref to hnr

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

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

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

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

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

$(\Gamma' * \textit{hn-ctxt}\ B'\ b\ bi * \textit{hn-ctxt}\ A'\ a\ ai)$
R
$(f\ (a,b))$
)
⟨*proof*⟩

lemma *hnr-intro-dummy*:
$\forall x\ xi.\ P\ x \longrightarrow \textit{hn-refine}\ (\Gamma\ x\ xi)\ (c\ xi)\ (\Gamma'\ x\ xi)\ R\ (a\ x) \Longrightarrow \forall x\ xi.\ P\ x \longrightarrow \textit{hn-refine}\ (\textit{emp}*\Gamma\ x\ xi)\ (c\ xi)\ (\textit{emp}*\Gamma'\ x\ xi)\ R\ (a\ x)$
⟨*proof*⟩

lemma *hn-ctxt-ctxt-fix-conv*: $\textit{hn-ctxt}\ (\textit{hn-ctxt}\ R) = \textit{hn-ctxt}\ R$
⟨*proof*⟩

lemma *uncurry-APP*: $\textit{uncurry}\ f\$(a,b) = f\$a\$b$ ⟨*proof*⟩

lemma *norm-RETURN-o*:
$\bigwedge f.\ (\textit{RETURN}\ o\ f)\$x = (\textit{RETURN}\$(f\$x))$
$\bigwedge f.\ (\textit{RETURN}\ oo\ f)\$x\$y = (\textit{RETURN}\$(f\$x\$y))$
$\bigwedge f.\ (\textit{RETURN}\ ooo\ f)\$x\$y\$z = (\textit{RETURN}\$(f\$x\$y\$z))$
$\bigwedge f.\ (\lambda x.\ \textit{RETURN}\ ooo\ f\ x)\$x\$y\$z\$a = (\textit{RETURN}\$(f\$x\$y\$z\$a))$
$\bigwedge f.\ (\lambda x\ y.\ \textit{RETURN}\ ooo\ f\ x\ y)\$x\$y\$z\$a\$b = (\textit{RETURN}\$(f\$x\$y\$z\$a\$b))$
⟨*proof*⟩

lemma *norm-return-o*:
$\bigwedge f.\ (\textit{return}\ o\ f)\$x = (\textit{return}\$(f\$x))$
$\bigwedge f.\ (\textit{return}\ oo\ f)\$x\$y = (\textit{return}\$(f\$x\$y))$
$\bigwedge f.\ (\textit{return}\ ooo\ f)\$x\$y\$z = (\textit{return}\$(f\$x\$y\$z))$
$\bigwedge f.\ (\lambda x.\ \textit{return}\ ooo\ f\ x)\$x\$y\$z\$a = (\textit{return}\$(f\$x\$y\$z\$a))$
$\bigwedge f.\ (\lambda x\ y.\ \textit{return}\ ooo\ f\ x\ y)\$x\$y\$z\$a\$b = (\textit{return}\$(f\$x\$y\$z\$a\$b))$
⟨*proof*⟩

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

Conversion from hnr to hfref

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

abbreviation *id-assn* ≡ *pure Id*
abbreviation *unit-assn* ≡ *id-assn* :: *unit* ⇒ -

lemma *pure-unit-rel-eq-empty*: $\textit{unit-assn}\ x\ y = \textit{emp}$
⟨*proof*⟩

lemma *uc-hfprod-sel*:
$\textit{fst}\ (A *_a B)\ a\ c = (\textit{case}\ (a,c)\ \textit{of}\ ((a1,a2),(c1,c2)) \Rightarrow \textit{fst}\ A\ a1\ c1 * \textit{fst}\ B\ a2\ c2)$

snd (*A* $*_a$ *B*) *a c* = (*case* (*a*,*c*) *of* ((*a1*,*a2*),(*c1*,*c2*)) ⇒ *snd A a1 c1* ∗ *snd B a2 c2*)
⟨*proof*⟩

Conversion from relation to fref

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

definition *CURRY R* ≡ { (*f*,*g*). (*uncurry f*, *uncurry g*) ∈ *R* }

lemma *fref-param1*: *R*→*S* = *fref* (λ-. *True*) *R S*
⟨*proof*⟩

lemma *fref-nest*: *fref P1 R1* (*fref P2 R2 S*)
≡ *CURRY* (*fref* (λ(*a*,*b*). *P1 a* ∧ *P2 b*) (*R1*×$_r$*R2*) *S*)
⟨*proof*⟩

lemma *in-CURRY-conv*: (*f*,*g*) ∈ *CURRY R* ⟷ (*uncurry f*, *uncurry g*) ∈ *R*
⟨*proof*⟩

lemma *uncurry0-APP*[*simp*]: *uncurry0 c* $ *x* = *c* ⟨*proof*⟩

lemma *fref-param0I*: (*c*,*a*)∈*R* ⟹ (*uncurry0 c*, *uncurry0 a*) ∈ *fref* (λ-. *True*) *unit-rel R*
⟨*proof*⟩

Composition

definition *hr-comp* :: ('*b* ⇒ '*c* ⇒ *assn*) ⇒ ('*b* × '*a*) *set* ⇒ '*a* ⇒ '*c* ⇒ *assn*
— Compose refinement assertion with refinement relation
where *hr-comp R1 R2 a c* ≡ ∃$_A$*b*. *R1 b c* ∗ ↑((*b*,*a*)∈*R2*)

definition *hrp-comp*
:: ('*d* ⇒ '*b* ⇒ *assn*) × ('*d* ⇒ '*c* ⇒ *assn*)
⇒ ('*d* × '*a*) *set* ⇒ ('*a* ⇒ '*b* ⇒ *assn*) × ('*a* ⇒ '*c* ⇒ *assn*)
— Compose argument assertion-pair with refinement relation
where *hrp-comp RR*′ *S* ≡ (*hr-comp* (*fst RR*′) *S*, *hr-comp* (*snd RR*′) *S*)

lemma *hr-compI*: (*b*,*a*)∈*R2* ⟹ *R1 b c* ⟹$_A$ *hr-comp R1 R2 a c*
⟨*proof*⟩

lemma *hr-comp-Id1*[*simp*]: *hr-comp* (*pure Id*) *R* = *pure R*
⟨*proof*⟩

lemma *hr-comp-Id2*[*simp*]: *hr-comp R Id* = *R*
⟨*proof*⟩

lemma *hr-comp-emp*[*simp*]: *hr-comp* (λ*a c*. *emp*) *R a c* = ↑(∃ *b*. (*b*,*a*)∈*R*)

⟨*proof*⟩

lemma *hr-comp-prod-conv*[*simp*]:
hr-comp (*prod-assn Ra Rb*) (*Ra′* ×$_r$ *Rb′*)
= *prod-assn* (*hr-comp Ra Ra′*) (*hr-comp Rb Rb′*)
⟨*proof*⟩

lemma *hr-comp-pure*: *hr-comp* (*pure R*) *S* = *pure* (*R O S*)
⟨*proof*⟩

lemma *hr-comp-is-pure*[*safe-constraint-rules*]: *is-pure A* ⟹ *is-pure* (*hr-comp A B*)
⟨*proof*⟩

lemma *hr-comp-the-pure*: *is-pure A* ⟹ *the-pure* (*hr-comp A B*) = *the-pure A O B*
⟨*proof*⟩

lemma *rdomp-hrcomp-conv*: *rdomp* (*hr-comp A R*) *x* ⟷ (∃ *y*. *rdomp A y* ∧ (*y*,*x*)∈*R*)
⟨*proof*⟩

lemma *hn-rel-compI*:
⟦*nofail a*; (*b*,*a*)∈⟨*R2*⟩*nres-rel*⟧ ⟹ *hn-rel R1 b c* ⟹$_A$ *hn-rel* (*hr-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-comp* (*hr-comp R S*) *T* = *hr-comp R* (*S O T*)
⟨*proof*⟩

lemma *hnr-comp*:
assumes *R*: ⋀*b1 c1*. *P b1* ⟹ *hn-refine* (*R1 b1 c1* ∗ Γ) (*c c1*) (*R1p b1 c1* ∗ Γ′) *R* (*b b1*)
assumes *S*: ⋀*a1 b1*. ⟦*Q a1*; (*b1*,*a1*)∈*R1′*⟧ ⟹ (*b b1*,*a a1*)∈⟨*R′*⟩*nres-rel*
assumes *PQ*: ⋀*a1 b1*. ⟦*Q a1*; (*b1*,*a1*)∈*R1′*⟧ ⟹ *P b1*
assumes *Q*: *Q a1*
shows *hn-refine*
(*hr-comp R1 R1′ a1 c1* ∗ Γ)
(*c c1*)
(*hr-comp R1p R1′ a1 c1* ∗ Γ′)
(*hr-comp R R′*)
(*a a1*)

⟨*proof*⟩

lemma *hnr-comp1-aux*:
assumes *R*: ⋀*b1 c1*. *P b1* ⟹ *hn-refine* (*hn-ctxt R1 b1 c1*) (*c c1*) (*hn-ctxt R1p b1 c1*) *R* (*b*$*b1*)
assumes *S*: ⋀*a1 b1*. ⟦*Q a1*; (*b1*,*a1*)∈*R1*⟧ ⟹ (*b*$*b1*,*a*$*a1*)∈⟨*R*⟩*nres-rel*
assumes *PQ*: ⋀*a1 b1*. ⟦*Q a1*; (*b1*,*a1*)∈*R1*⟧ ⟹ *P b1*
assumes *Q*: *Q a1*
shows *hn-refine*
(*hr-comp R1 R1*′ *a1 c1*)
(*c c1*)
(*hr-comp R1p R1*′ *a1 c1*)
(*hr-comp R R*′)
(*a a1*)
⟨*proof*⟩

lemma *hfcomp*:
assumes *A*: (*f*,*g*) ∈ [*P*]$_a$ *RR*′ → *S*
assumes *B*: (*g*,*h*) ∈ [*Q*]$_f$ *T* → ⟨*U*⟩*nres-rel*
shows (*f*,*h*) ∈ [λ*a*. *Q a* ∧ (∀ *a*′. (*a*′,*a*)∈*T* ⟶ *P a*′)]$_a$
hrp-comp RR′ *T* → *hr-comp S U*
⟨*proof*⟩

lemma *hfref-weaken-pre-nofail*:
assumes (*f*,*g*) ∈ [*P*]$_a$ *R* → *S*
shows (*f*,*g*) ∈ [λ*x*. *nofail* (*g x*) ⟶ *P x*]$_a$ *R* → *S*
⟨*proof*⟩

lemma *hfref-cons*:
assumes (*f*,*g*) ∈ [*P*]$_a$ *R* → *S*
assumes ⋀*x*. *P*′ *x* ⟹ *P x*
assumes ⋀*x y*. *fst R*′ *x y* ⟹$_t$ *fst R x y*
assumes ⋀*x y*. *snd R x y* ⟹$_t$ *snd R*′ *x y*
assumes ⋀*x y*. *S x y* ⟹$_t$ *S*′ *x y*
shows (*f*,*g*) ∈ [*P*′]$_a$ *R*′ → *S*′
⟨*proof*⟩

Composition Automation

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

lemma *prod-hrp-comp*:
hrp-comp (*A* ∗$_a$ *B*) (*C* ×$_r$ *D*) = *hrp-comp A C* ∗$_a$ *hrp-comp B D*
⟨*proof*⟩

lemma *hrp-comp-keep*: *hrp-comp* (*A*^{k}) *B* = (*hr-comp A B*)^k
⟨*proof*⟩

lemma *hr-comp-invalid*: *hr-comp* (*invalid-assn R1*) *R2* = *invalid-assn* (*hr-comp R1 R2*)

⟨*proof*⟩

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

definition *hrp-imp* $RR\ RR' \equiv$
$\forall\, a\ b.\ (fst\ RR'\ a\ b \Longrightarrow_t fst\ RR\ a\ b) \wedge (snd\ RR\ a\ b \Longrightarrow_t snd\ RR'\ a\ b)$

lemma *hfref-imp*: *hrp-imp* $RR\ RR' \Longrightarrow [P]_a\ RR \to S \subseteq [P]_a\ RR' \to S$
⟨*proof*⟩

lemma *hrp-imp-refl*: *hrp-imp* $RR\ RR$
⟨*proof*⟩

lemma *hrp-imp-reflI*: $RR = RR' \Longrightarrow$ *hrp-imp* $RR\ RR'$
⟨*proof*⟩

lemma *hrp-comp-cong*: *hrp-imp* $A\ A' \Longrightarrow B{=}B' \Longrightarrow$ *hrp-imp* (*hrp-comp* $A\ B$) (*hrp-comp* $A'\ B'$)
⟨*proof*⟩

lemma *hrp-prod-cong*: *hrp-imp* $A\ A' \Longrightarrow$ *hrp-imp* $B\ B' \Longrightarrow$ *hrp-imp* $(A *_a B)$ $(A' *_a B')$
⟨*proof*⟩

lemma *hrp-imp-trans*: *hrp-imp* $A\ B \Longrightarrow$ *hrp-imp* $B\ C \Longrightarrow$ *hrp-imp* $A\ C$
⟨*proof*⟩

lemma *fcomp-norm-dflt-init*: $x{\in}[P]_a\ R \to T \Longrightarrow$ *hrp-imp* $R\ S \Longrightarrow x{\in}[P]_a\ S \to T$
⟨*proof*⟩

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 $\bigwedge x\ y.\ [\![P\ x;\ (y,x){\in}R;\ y{\in}Domain\ R;\ S'\ x\ ]\!] \Longrightarrow Q\ x\ y \equiv Q'\ x\ y$
shows *comp-PRE* $R\ P\ Q\ S \equiv$ *comp-PRE* $R'\ P'\ Q'\ S'$
⟨*proof*⟩

lemma *fref-compI-PRE*:
$[\![\ (f,g){\in}fref\ P\ R1\ R2;\ (g,h){\in}fref\ Q\ S1\ S2\ ]\!]$
$\Longrightarrow (f,h) \in fref\ (comp\text{-}PRE\ S1\ Q\ (\lambda\text{-}.\ P)\ (\lambda\text{-}.\ True))\ (R1\ O\ S1)\ (R2\ O\ S2)$

⟨*proof*⟩

lemma *PRE-D1*: $(Q\ x \wedge P\ x) \longrightarrow$ *comp-PRE S1 Q* $(\lambda x\ \text{-}.\ P\ x)\ S\ x$
⟨*proof*⟩

lemma *PRE-D2*: $(Q\ x \wedge (\forall y.\ (y,x) \in S1 \longrightarrow S\ x \longrightarrow P\ x\ y)) \longrightarrow$ *comp-PRE S1 Q P S x*
⟨*proof*⟩

lemma *fref-weaken-pre*:
assumes $\bigwedge x.\ P\ x \longrightarrow P'\ x$
assumes $(f,h) \in$ *fref* $P'\ R\ S$
shows $(f,h) \in$ *fref* $P\ R\ S$
⟨*proof*⟩

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

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

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

lemma *hfref-weaken-pre*:
assumes $\bigwedge x.\ P\ x \longrightarrow P'\ x$
assumes $(f,h) \in$ *hfref* $P'\ R\ S$
shows $(f,h) \in$ *hfref* $P\ R\ S$
⟨*proof*⟩

lemma *hfref-weaken-pre'*:
assumes $\bigwedge x.\ [\![P\ x;\ rdomp\ (fst\ R)\ x]\!] \Longrightarrow P'\ x$
assumes $(f,h) \in$ *hfref* $P'\ R\ S$
shows $(f,h) \in$ *hfref* $P\ R\ S$
⟨*proof*⟩

lemma *hfref-weaken-pre-nofail'*:
assumes $(f,g) \in [P]_a\ R \to S$
assumes $\bigwedge x.\ [\![nofail\ (g\ x);\ Q\ x]\!] \Longrightarrow P\ x$
shows $(f,g) \in [Q]_a\ R \to S$
⟨*proof*⟩

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

hrp-comp RR′ T → hr-comp S U
⟨*proof*⟩

lemma *hfref-compI-PRE*:
assumes *A*: (*f*,*g*) ∈ [*P*]$_a$ *RR′* → *S*
assumes *B*: (*g*,*h*) ∈ [*Q*]$_f$ *T* → ⟨*U*⟩*nres-rel*
shows (*f*,*h*) ∈ [*comp-PRE T Q* (λ*x y*. *P y*) (λ*x*. *nofail* (*h x*))]$_a$
hrp-comp RR′ T → hr-comp S U
⟨*proof*⟩

lemma *hfref-PRE-D1*:
assumes (*f*,*h*) ∈ *hfref* (*comp-PRE S1 Q* (λ*x* -. *P x*) *X*) *R S*
shows (*f*,*h*) ∈ *hfref* (λ*x*. *Q x* ∧ *P x*) *R S*
⟨*proof*⟩

lemma *hfref-PRE-D2*:
assumes (*f*,*h*) ∈ *hfref* (*comp-PRE S1 Q P X*) *R S*
shows (*f*,*h*) ∈ *hfref* (λ*x*. *Q x* ∧ (∀ *y*. (*y*,*x*)∈*S1* ⟶ *X x* ⟶ *P x y*)) *R S*
⟨*proof*⟩

lemma *hfref-PRE-D3*:
assumes (*f*,*h*) ∈ *hfref* (*comp-PRE S1 Q P X*) *R S*
shows (*f*,*h*) ∈ *hfref* (*comp-PRE S1 Q P X*) *R S*
⟨*proof*⟩

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* ‹*to-hnr converter*: *Postprocessing unfold rules*›

lemma *uncurry0-add-app-tag*: *uncurry0* (*RETURN c*) = *uncurry0* (*RETURN*$*c*)
⟨*proof*⟩

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* **for** *x*] *mult-1-right*[*of x*::*assn* **for** *x*]

named-theorems *to-hfref-post* ‹*to-hfref converter*: *Postprocessing unfold rules*›
lemma *prod-casesK*[*to-hfref-post*]: *case-prod* (λ- -. *k*) = (λ-. *k*) ⟨*proof*⟩
lemma *uncurry0-hfref-post*[*to-hfref-post*]: *hfref* (*uncurry0 True*) *R S* = *hfref* (λ-. *True*) *R S*
⟨*proof*⟩

Configuration for relation normalization after composition

named-theorems *fcomp-norm-unfold* ‹*fcomp−normalizer*: *Unfold theorems*›
named-theorems *fcomp-norm-simps* ‹*fcomp−normalizer*: *Simplification theorems*›
named-theorems *fcomp-norm-init fcomp−normalizer*: *Initialization rules*
named-theorems *fcomp-norm-trans fcomp−normalizer*: *Transitivity rules*
named-theorems *fcomp-norm-cong fcomp−normalizer*: *Congruence rules*
named-theorems *fcomp-norm-norm fcomp−normalizer*: *Normalization rules*
named-theorems *fcomp-norm-refl fcomp−normalizer*: *Reflexivity rules*

Default Setup

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

lemma [*fcomp-norm-simps*]: *CONSTRAINT is-pure P* ⟹ *pure* (*the-pure P*) = *P* ⟨*proof*⟩
lemmas [*fcomp-norm-simps*] = *True-implies-equals*

lemmas [*fcomp-norm-init*] = *fcomp-norm-dflt-init*
lemmas [*fcomp-norm-trans*] = *hrp-imp-trans*
lemmas [*fcomp-norm-cong*] = *hrp-comp-cong hrp-prod-cong*

lemmas [*fcomp-norm-refl*] = *refl hrp-imp-refl*

lemma *ensure-fref-nresI*: (*f*,*g*)∈[*P*]$_f$ *R*→*S* ⟹ (*RETURN o f*, *RETURN o g*)∈[*P*]$_f$ *R*→⟨*S*⟩*nres-rel*
⟨*proof*⟩

lemma *ensure-fref-nres-unfold*:
⋀*f*. *RETURN o* (*uncurry0 f*) = *uncurry0* (*RETURN f*)
⋀*f*. *RETURN o* (*uncurry f*) = *uncurry* (*RETURN oo f*)
⋀*f*. (*RETURN ooo uncurry*) *f* = *uncurry* (*RETURN ooo f*)
⟨*proof*⟩

Composed precondition normalizer

named-theorems *fcomp-prenorm-simps* ‹*fcomp precondition−normalizer*: *Simplification theorems*›

Support for preconditions of the form -∈*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 ⋀*x*. *PROTECT P x* ⟶ *P′ x*
assumes (*f*,*h*) ∈ *hfref P′ R S*

shows $(f,h) \in$ *hfref P R S*
⟨*proof*⟩

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

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

lemma *auto-weaken-pre-uncurry-step*:
assumes *PROTECT f a* $\equiv$ *f'*
shows *PROTECT* $(\lambda(x,y).$ *f x y*$)$ $(a,b) \equiv$ *f' b*
⟨*proof*⟩

lemma *auto-weaken-pre-uncurry-finish*:
PROTECT f x $\equiv$ *f x* ⟨*proof*⟩

lemma *auto-weaken-pre-uncurry-start*:
assumes *P* $\equiv$ *P'*
assumes *P'*$\longrightarrow$*Q*
shows *P*$\longrightarrow$*Q*
⟨*proof*⟩

lemma *auto-weaken-pre-comp-PRE-I*:
assumes *S x* $\Longrightarrow$ *P x*
assumes $\bigwedge y.$ $[\![(y,x)\in R;$ *P x*; *S x*$]\!] \Longrightarrow$ *Q x y*
shows *comp-PRE R P Q S x*
⟨*proof*⟩

lemma *auto-weaken-pre-to-imp-nf*:
$(A\longrightarrow B\longrightarrow C) = (A\wedge B \longrightarrow C)$
$((A\wedge B)\wedge C) = (A\wedge B\wedge C)$
⟨*proof*⟩

lemma *auto-weaken-pre-add-dummy-imp*:
P $\Longrightarrow$ *True* $\longrightarrow$ *P* ⟨*proof*⟩

Synthesis for hfref statements

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

lemma *hfsynth-ID-R-D*:
fixes *I* :: *'a itself*
assumes *hfsynth-ID-R R a*
assumes *intf-of-assn R I*
shows *a* $::_i$ *I*
⟨*proof*⟩

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

lemma *hfsynth-ID-R-uncurry-unfold*:
hfsynth-ID-R (*to-hnr-prod R S*) (*a*,*b*) $\equiv$ *hfsynth-ID-R R a* $\wedge$ *hfsynth-ID-R S b*
hfsynth-ID-R (*fst* (*hf-pres R k*)) $\equiv$ *hfsynth-ID-R R*
⟨*proof*⟩

⟨*ML*⟩

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* ⟨*proof*⟩
named-theorems *--itype-rewrite*

⟨*ML*⟩

end

1.6.2 Rewriting Inferred Interface Types

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

named-theorems-rev *map-type-eqs*

1.6.3 ML-Code

context begin

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

$\langle ML \rangle$

end

$\langle ML \rangle$

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*: $\bigwedge c.\ c\$x1\$x2\$x3 \equiv (\gg\!=)\$(EVAL\$x1)\$(\lambda_2 x1.\ (\gg\!=)\$(EVAL\$x2)\$(\lambda_2 x2.\ (\gg\!=)\$(EVAL\$x3)\$(\lambda_2 x3.\ SP\ (c\$x1\$x2\$x3))))$
$\langle proof \rangle$

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* $x\ m \equiv RETURN\ x \leq m$

Tag to keep track of preconditions in assertions

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

lemma *vassn-tagI*: $h \models \Gamma \Longrightarrow$ *vassn-tag* Γ
⟨*proof*⟩

lemma *vassn-dest*[*dest*!]:
vassn-tag $(\Gamma_1 * \Gamma_2) \Longrightarrow$ *vassn-tag* $\Gamma_1 \wedge$ *vassn-tag* Γ_2
vassn-tag (*hn-ctxt* $R\ a\ b$) $\Longrightarrow a \in$ *rdom* R
⟨*proof*⟩

lemma *entails-preI*:
assumes *vassn-tag* $A \Longrightarrow A \Longrightarrow_A B$
shows $A \Longrightarrow_A B$
⟨*proof*⟩

lemma *invalid-assn-const*:
invalid-assn $(\lambda\text{- -}.\ P)\ x\ y = \uparrow$(*vassn-tag* P) $*$ *true*
⟨*proof*⟩

lemma *vassn-tag-simps*[*simp*]:
vassn-tag emp
vassn-tag true
⟨*proof*⟩

definition *GEN-ALGO* $f\ \Phi \equiv \Phi\ f$
— Tag to synthesize f with property Φ.

lemma *is-GEN-ALGO*: *GEN-ALGO* $f\ \Phi \Longrightarrow$ *GEN-ALGO* $f\ \Phi$ ⟨*proof*⟩

Tag for side-condition solver to discharge by assumption

definition *RPREM* :: *bool* ⇒ *bool* **where** [*simp*]: *RPREM P* = *P*
lemma *RPREMI*: *P* ⟹ *RPREM P* ⟨*proof*⟩

lemma *trans-frame-rule*:
  assumes *RECOVER-PURE* Γ Γ′
  assumes *vassn-tag* Γ′ ⟹ *hn-refine* Γ′ *c* Γ″ *R a*
  shows *hn-refine* (*F*∗Γ) *c* (*F*∗Γ″) *R a*
  ⟨*proof*⟩
  applyF (*rule hn-refine-cons-pre*)
    focus ⟨*proof*⟩ **solved**

    apply1 (*rule hn-refine-preI*)
    apply1 (*rule assms*)
    applyS (*auto simp add*: *vassn-tag-def*)
  solved
  ⟨*proof*⟩

lemma *recover-pure-cons*: — Used for debugging
  assumes *RECOVER-PURE* Γ Γ′
  assumes *hn-refine* Γ′ *c* Γ″ *R a*
  shows *hn-refine* (Γ) *c* (Γ″) *R a*
  ⟨*proof*⟩
definition *CPR-TAG* :: *assn* ⇒ *assn* ⇒ *bool* **where** [*simp*]: *CPR-TAG y x* ≡ *True*
lemma *CPR-TAG-starI*:
  assumes *CPR-TAG P1 Q1*
  assumes *CPR-TAG P2 Q2*
  shows *CPR-TAG* (*P1*∗*P2*) (*Q1*∗*Q2*)
  ⟨*proof*⟩
lemma *CPR-tag-ctxtI*: *CPR-TAG* (*hn-ctxt R x xi*) (*hn-ctxt R′ x xi*) ⟨*proof*⟩
lemma *CPR-tag-fallbackI*: *CPR-TAG P Q* ⟨*proof*⟩

lemmas *CPR-TAG-rules* = *CPR-TAG-starI CPR-tag-ctxtI CPR-tag-fallbackI*

lemma *cons-pre-rule*: — Consequence rule to be applied if no direct operation rule matches
  assumes *CPR-TAG P P′*
  assumes *P* ⟹$_t$ *P′*
  assumes *hn-refine P′ c Q R m*
  shows *hn-refine P c Q R m*
  ⟨*proof*⟩

named-theorems-rev *sepref-gen-algo-rules* ‹*Sepref*: *Generic algorithm rules*›

⟨*ML*⟩

Basic Setup

lemma *hn-pass*[*sepref-fr-rules*]:
shows *hn-refine* (*hn-ctxt P x x'*) (*return x'*) (*hn-invalid P x x'*) *P* (*PASS*$*x*)
⟨*proof*⟩

lemma *hn-bind*[*sepref-comb-rules*]:
assumes *D1*: *hn-refine* Γ *m'* Γ*1 Rh m*
assumes *D2*:
⋀*x x'*. *bind-ref-tag x m* ⟹
hn-refine (Γ*1* ∗ *hn-ctxt Rh x x'*) (*f' x'*) (Γ*2 x x'*) *R* (*f x*)
assumes *IMP*: ⋀*x x'*. Γ*2 x x'* ⟹$_t$ Γ' ∗ *hn-ctxt Rx x x'*
shows *hn-refine* Γ (*m'*⤜*f'*) Γ' *R* (*Refine-Basic.bind*$*m*$(λ$_2$*x*. *f x*))
⟨*proof*⟩

lemma *hn-RECT'*[*sepref-comb-rules*]:
assumes *INDEP Ry INDEP Rx INDEP Rx'*
assumes *FR*: *P* ⟹$_t$ *hn-ctxt Rx ax px* ∗ *F*
assumes *S*: ⋀*cf af ax px*. ⟦
⋀*ax px*. *hn-refine* (*hn-ctxt Rx ax px* ∗ *F*) (*cf px*) (*hn-ctxt Rx' ax px* ∗ *F*) *Ry*
(*RCALL*$*af*$*ax*)⟧
⟹ *hn-refine* (*hn-ctxt Rx ax px* ∗ *F*) (*cB cf px*) (*F' ax px*) *Ry*
(*aB af ax*)
assumes *FR'*: ⋀*ax px*. *F' ax px* ⟹$_t$ *hn-ctxt Rx' ax px* ∗ *F*
assumes *M*: (⋀*x*. *mono-Heap* (λ*f*. *cB f x*))

shows *hn-refine*
(*P*) (*heap.fixp-fun cB px*) (*hn-ctxt Rx' ax px* ∗ *F*) *Ry*
(*RECT*$(λ$_2$*D x*. *aB D x*)$*ax*)
⟨*proof*⟩

lemma *hn-RCALL*[*sepref-comb-rules*]:
assumes *RPREM* (*hn-refine P' c Q' R* (*RCALL* $ *a* $ *b*))
and *P* ⟹$_t$ *F* ∗ *P'*
shows *hn-refine P c* (*F* ∗ *Q'*) *R* (*RCALL* $ *a* $ *b*)
⟨*proof*⟩

definition *monadic-WHILEIT I b f s* ≡ *do* {
RECT (λ*D s*. *do* {
ASSERT (*I s*);
bv ← *b s*;
if bv then do {
s ← *f s*;
D s
} *else do* {*RETURN s*}
}) *s*

```
}

definition heap-WHILET b f s ≡ do {
  heap.fixp-fun (λD s. do {
    bv ← b s;
    if bv then do {
      s ← f s;
      D s
    } else do {return s}
  }) s
}

lemma heap-WHILET-unfold[code]: heap-WHILET b f s =
  do {
    bv ← b s;
    if bv then do {
      s ← f s;
      heap-WHILET b f s
    } else
      return s
  }
  ⟨proof⟩



lemma WHILEIT-to-monadic: WHILEIT I b f s = monadic-WHILEIT I (λs.
RETURN (b s)) f s
  ⟨proof⟩

lemma WHILEIT-pat[def-pat-rules]:
  WHILEIT$I ≡ UNPROTECT (WHILEIT I)
  WHILET ≡ PR-CONST (WHILEIT (λ-. True))
  ⟨proof⟩

lemma id-WHILEIT[id-rules]:
  PR-CONST (WHILEIT I) ::ᵢ TYPE((′a ⇒ bool) ⇒ (′a ⇒ ′a nres) ⇒ ′a ⇒ ′a
nres)
  ⟨proof⟩

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

  PR-CONST (WHILEIT I) ≡ λ2b f s. SP (PR-CONST (WHILEIT I))$(λ2s.
b$s)$(λ2s. f$s)$s
  ⟨proof⟩

lemma WHILEIT-comb[sepref-monadify-comb]:
  PR-CONST (WHILEIT I)$(λ2x. b x)$f$s ≡
    Refine-Basic.bind$(EVAL$s)$(λ2s.
      SP (PR-CONST (monadic-WHILEIT I))$(λ2x. (EVAL$(b x)))$f$s
```

```
    )
  ⟨proof⟩

lemma hn-monadic-WHILE-aux:
  assumes FR: P ⟹_t Γ ∗ hn-ctxt Rs s' s
  assumes b-ref: ⋀s s'. I s' ⟹ hn-refine
    (Γ ∗ hn-ctxt Rs s' s)
    (b s)
    (Γb s' s)
    (pure bool-rel)
    (b' s')
  assumes b-fr: ⋀s' s. Γb s' s ⟹_t Γ ∗ hn-ctxt Rs s' s

  assumes f-ref: ⋀s' s. ⟦I s'⟧ ⟹ hn-refine
    (Γ ∗ hn-ctxt Rs s' s)
    (f s)
    (Γf s' s)
    Rs
    (f' s')
  assumes f-fr: ⋀s' s. Γf s' s ⟹_t Γ ∗ hn-ctxt (λ- -. true) s' s

  shows hn-refine (P) (heap-WHILET b f s) (Γ ∗ hn-invalid Rs s' s) Rs (monadic-WHILEIT
I b' f' s')
  ⟨proof⟩
  apply1 (rule hn-refine-cons-pre[OF FR])
  ⟨proof⟩
  focus (rule hn-refine-cons-pre[OF - hnr-RECT])
    applyS (subst mult-ac(2)[of Γ]; 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])
            ⟨proof⟩
          solved

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

          ⟨proof⟩
```

solved
applyF (*sep-auto*,*rule hn-refine-frame*)
applyS (*rule hnr-RETURN-pass*)

⟨*proof*⟩
solved

⟨*proof*⟩
solved

⟨*proof*⟩
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
⟨*proof*⟩
solved
⟨*proof*⟩

lemma *hn-monadic-WHILE-lin*[*sepref-comb-rules*]:
assumes *INDEP Rs*
assumes *FR*: $P \Longrightarrow_t \Gamma * \textit{hn-ctxt Rs } s' \ s$
assumes *b-ref*: $\bigwedge s\ s'.\ I\ s' \Longrightarrow \textit{hn-refine}$
($\Gamma * \textit{hn-ctxt Rs } s' \ s$)
($b\ s$)
($\Gamma b\ s' \ s$)
(*pure bool-rel*)
($b' \ s'$)
assumes *b-fr*: $\bigwedge s'\ s.\ \textit{TERM } (\textit{monadic-WHILEIT},''\textit{cond}'') \Longrightarrow \Gamma b\ s' \ s \Longrightarrow_t \Gamma * \textit{hn-ctxt Rs } s' \ s$

assumes *f-ref*: $\bigwedge s'\ s.\ I\ s' \Longrightarrow \textit{hn-refine}$
($\Gamma * \textit{hn-ctxt Rs } s' \ s$)
($f\ s$)
($\Gamma f\ s' \ s$)
Rs
($f' \ s'$)
assumes *f-fr*: $\bigwedge s'\ s.\ \textit{TERM } (\textit{monadic-WHILEIT},''\textit{body}'') \Longrightarrow \Gamma f\ s' \ s \Longrightarrow_t \Gamma * \textit{hn-ctxt } (\lambda\text{- -}.\ \textit{true})\ s' \ s$
shows *hn-refine*
P
(*heap-WHILET b f s*)
($\Gamma * \textit{hn-invalid Rs } s' \ s$)
Rs

(*PR-CONST* (*monadic-WHILEIT I*)$($\lambda_2 s'.\ b'\ s'$)$($\lambda_2 s'.\ f'\ s'$)$($s'$))
⟨*proof*⟩

lemma *monadic-WHILEIT-refine*[*refine*]:
assumes [*refine*]: $(s',s) \in R$
assumes [*refine*]: $\bigwedge s'\ s.\ [\![\ (s',s)\in R;\ I\ s\ ]\!] \Longrightarrow I'\ s'$
assumes [*refine*]: $\bigwedge s'\ s.\ [\![\ (s',s)\in R;\ I\ s;\ I'\ s'\ ]\!] \Longrightarrow b'\ s' \leq \Downarrow$*bool-rel* $(b\ s)$
assumes [*refine*]: $\bigwedge s'\ s.\ [\![\ (s',s)\in R;\ I\ s;\ I'\ s';$ *nofail* $(b\ s);$ *inres* $(b\ s)$ *True* $]\!]$ $\Longrightarrow f'\ s' \leq \Downarrow R\ (f\ s)$
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*]: $\bigwedge s'\ s.\ [\![\ (s',s)\in R;\ I\ s\ ]\!] \Longrightarrow I'\ s'$
assumes [*THEN order-trans*,*refine-vcg*]: $\bigwedge s'\ s.\ [\![\ (s',s)\in R;\ I\ s;\ I'\ s'\ ]\!] \Longrightarrow b'\ s' \leq$ *SPEC* $(\lambda r.\ r = b\ s)$
assumes [*refine*]: $\bigwedge s'\ s.\ [\![\ (s',s)\in R;\ I\ s;\ I'\ s';\ b\ s\ ]\!] \Longrightarrow f'\ s' \leq \Downarrow R\ (f\ s)$
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*]: $\bigwedge s'\ s.\ [\![\ (s',s)\in R\ ]\!] \Longrightarrow b'\ s' \leq$ *SPEC* $(\lambda r.\ r = b\ s)$
assumes [*refine*]: $\bigwedge s'\ s.\ [\![\ (s',s)\in R;\ b\ s\ ]\!] \Longrightarrow f'\ s' \leq \Downarrow R\ (f\ s)$
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$I \equiv$ *UNPROTECT* (*monadic-WHILEIT* I)
⟨*proof*⟩

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

lemma *monadic-WHILEIT-arities*[*sepref-monadify-arity*]:
PR-CONST (*monadic-WHILEIT* I) $\equiv \lambda_2 b\ f\ s.$ *SP* (*PR-CONST* (*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)bf$s \equiv$
Refine-Basic.bind$(*EVAL*$s$)$($\lambda_2 s.$
SP (*PR-CONST* (*monadic-WHILEIT* I))bf$s
)
⟨*proof*⟩

definition [*simp*]: *op-ASSERT-bind I m* ≡ *Refine-Basic.bind* (*ASSERT I*) (λ-. *m*)
lemma *pat-ASSERT-bind*[*def-pat-rules*]:
Refine-Basic.bind$(*ASSERT*$*I*)$(λ$_2$-. *m*) ≡ *UNPROTECT* (*op-ASSERT-bind I*)$*m*
⟨*proof*⟩

term *PR-CONST* (*op-ASSERT-bind I*)
lemma *id-op-ASSERT-bind*[*id-rules*]:
PR-CONST (*op-ASSERT-bind I*) ::$_i$ *TYPE*('*a nres* ⇒ '*a nres*)
⟨*proof*⟩

lemma *arity-ASSERT-bind*[*sepref-monadify-arity*]:
PR-CONST (*op-ASSERT-bind I*) ≡ λ$_2$*m*. *SP* (*PR-CONST* (*op-ASSERT-bind I*))$*m*
⟨*proof*⟩

lemma *hn-ASSERT-bind*[*sepref-comb-rules*]:
assumes *I* ⟹ *hn-refine* Γ *c* Γ' *R m*
shows *hn-refine* Γ *c* Γ' *R* (*PR-CONST* (*op-ASSERT-bind I*)$*m*)
⟨*proof*⟩

definition [*simp*]: *op-ASSUME-bind I m* ≡ *Refine-Basic.bind* (*ASSUME I*) (λ-. *m*)
lemma *pat-ASSUME-bind*[*def-pat-rules*]:
Refine-Basic.bind$(*ASSUME*$*I*)$(λ$_2$-. *m*) ≡ *UNPROTECT* (*op-ASSUME-bind I*)$*m*
⟨*proof*⟩

lemma *id-op-ASSUME-bind*[*id-rules*]:
PR-CONST (*op-ASSUME-bind I*) ::$_i$ *TYPE*('*a nres* ⇒ '*a nres*)
⟨*proof*⟩

lemma *arity-ASSUME-bind*[*sepref-monadify-arity*]:
PR-CONST (*op-ASSUME-bind I*) ≡ λ$_2$*m*. *SP* (*PR-CONST* (*op-ASSUME-bind I*))$*m*
⟨*proof*⟩

lemma *hn-ASSUME-bind*[*sepref-comb-rules*]:
assumes *vassn-tag* Γ ⟹ *I*
assumes *I* ⟹ *hn-refine* Γ *c* Γ' *R m*
shows *hn-refine* Γ *c* Γ' *R* (*PR-CONST* (*op-ASSUME-bind I*)$*m*)
⟨*proof*⟩

1.7.1 Import of Parametricity Theorems

lemma *pure-hn-refineI*:
assumes *Q* ⟶ (*c*,*a*)∈*R*

shows *hn-refine* (↑Q) (*return c*) (↑Q) (*pure R*) (*RETURN a*)
⟨*proof*⟩

lemma *pure-hn-refineI-no-asm*:
assumes (c,a)∈R
shows *hn-refine emp* (*return c*) *emp* (*pure R*) (*RETURN a*)
⟨*proof*⟩

lemma *import-param-0*:
(P⟹Q) ≡ *Trueprop* (*PROTECT P* ⟶ Q)
⟨*proof*⟩

lemma *import-param-1*:
(P⟹Q) ≡ *Trueprop* (P⟶Q)
(P⟶Q⟶R) ⟷ (P∧Q ⟶ R)
PROTECT (P ∧ Q) ≡ *PROTECT P* ∧ *PROTECT Q*
(P ∧ Q) ∧ R ≡ P ∧ Q ∧ R
(a,c)∈*Rel* ∧ *PROTECT P* ⟷ *PROTECT P* ∧ (a,c)∈*Rel*
⟨*proof*⟩

lemma *import-param-2*:
Trueprop (*PROTECT P* ∧ Q ⟶ R) ≡ (P ⟹ Q⟶R)
⟨*proof*⟩

lemma *import-param-3*:
↑(P ∧ Q) = ↑P∗↑Q
↑((c,a)∈R) = *hn-val R a c*
⟨*proof*⟩

named-theorems-rev *sepref-import-rewrite* ‹*Rewrite rules on importing parametricity theorems*›

lemma *to-import-frefD*:
assumes (f,g)∈*fref P R S*
shows ⟦*PROTECT* (*P y*); (x,y)∈R⟧ ⟹ (*f x*, *g y*)∈S
⟨*proof*⟩

lemma *add-PR-CONST*: (c,a)∈R ⟹ (c,*PR-CONST a*)∈R ⟨*proof*⟩

⟨*ML*⟩

1.7.2 Purity

definition *import-rel1 R* ≡ λA *c ci*. ↑(*is-pure A* ∧ (ci,c)∈⟨*the-pure A*⟩R)
definition *import-rel2 R* ≡ λA B *c ci*. ↑(*is-pure A* ∧ *is-pure B* ∧ (ci,c)∈⟨*the-pure A*, *the-pure B*⟩R)

lemma *import-rel1-pure-conv*: *import-rel1 R* (*pure A*) = *pure* (⟨A⟩R)
⟨*proof*⟩

lemma *import-rel2-pure-conv*: *import-rel2 R* (*pure A*) (*pure B*) = *pure* (⟨*A*,*B*⟩*R*)
⟨*proof*⟩

lemma *precise-pure*[*constraint-rules*]: *single-valued R* ⟹ *precise* (*pure R*)
⟨*proof*⟩

lemma *precise-pure-iff-sv*: *precise* (*pure R*) ⟷ *single-valued R*
⟨*proof*⟩

lemma *pure-precise-iff-sv*: ⟦*is-pure R*⟧
⟹ *precise R* ⟷ *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 [[*cd-patterns hn-refine - ?f - - -*]]

lemma *heap-fixp-codegen*:
assumes *DEF*: *f* ≡ *heap.fixp-fun cB*
assumes *M*: (⋀*x*. *mono-Heap* (λ*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* ⇒ ′*b* ⇒ *assn*) ⇒ (′*a* ⇒ ′*b* ⇒ *assn*)×(′*a* ⇒ ′*b* ⇒ *assn*)
((-[?]) [*1000*] *999*)

where $R^? \equiv$ *hf-pres R UNSPEC*

definition *SYNTH* :: $('a \Rightarrow 'r\ nres) \Rightarrow (('ai \Rightarrow 'ri\ Heap) \times ('a \Rightarrow 'r\ nres))\ set \Rightarrow bool$
where *SYNTH f R* $\equiv$ *True*

definition [*simp*]: *CP-UNCURRY - -* $\equiv$ *True*
definition [*simp*]: *INTRO-KD - -* $\equiv$ *True*
definition [*simp*]: *SPEC-RES-ASSN - -* $\equiv$ *True*

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

lemma [*synth-rules*]: ⟦*INTRO-KD R1 R1'*; *INTRO-KD R2 R2'*⟧ $\Longrightarrow$ *INTRO-KD* ($R1 *_a R2$) ($R1' *_a R2'$) ⟨*proof*⟩
lemma [*synth-rules*]: *INTRO-KD* ($R^?$) (*hf-pres R k*) ⟨*proof*⟩
lemma [*synth-rules*]: *INTRO-KD* (R^k) (R^k) ⟨*proof*⟩
lemma [*synth-rules*]: *INTRO-KD* (R^d) (R^d) ⟨*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'*⟧ $\Longrightarrow$ *SYNTH-TERM* (*SYNTH f* ($[P]_a$ $R \to S$)) ((*fi*,*SDUMMY*)∈*SDUMMY*,(*fi*,*f*)∈($[P]_a$ $R' \to S'$))
⟨*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* :: $('a \times 'b)\ set \Rightarrow 'c\ itself \Rightarrow bool$

where [*simp*]: *INTF-OF-REL R I* ≡ *True*

lemma *intf-of-relI*: *INTF-OF-REL* (*R*::(-×′*a*) *set*) *TYPE*(′*a*) ⟨*proof*⟩
declare *intf-of-relI*[*synth-rules*] — Declare as fallback rule

lemma [*synth-rules*]:
INTF-OF-REL unit-rel TYPE(*unit*)
INTF-OF-REL nat-rel TYPE(*nat*)
INTF-OF-REL int-rel TYPE(*int*)
INTF-OF-REL bool-rel TYPE(*bool*)

INTF-OF-REL R TYPE(′*a*) ⟹ *INTF-OF-REL* (⟨*R*⟩*option-rel*) *TYPE*(′*a option*)
INTF-OF-REL R TYPE(′*a*) ⟹ *INTF-OF-REL* (⟨*R*⟩*list-rel*) *TYPE*(′*a list*)
INTF-OF-REL R TYPE(′*a*) ⟹ *INTF-OF-REL* (⟨*R*⟩*nres-rel*) *TYPE*(′*a nres*)
⟦*INTF-OF-REL R TYPE*(′*a*); *INTF-OF-REL S TYPE*(′*b*)⟧ ⟹ *INTF-OF-REL* (*R*×$_r$*S*) *TYPE*(′*a*×′*b*)
⟦*INTF-OF-REL R TYPE*(′*a*); *INTF-OF-REL S TYPE*(′*b*)⟧ ⟹ *INTF-OF-REL* (⟨*R*,*S*⟩*sum-rel*) *TYPE*(′*a*+′*b*)
⟦*INTF-OF-REL R TYPE*(′*a*); *INTF-OF-REL S TYPE*(′*b*)⟧ ⟹ *INTF-OF-REL* (*R*→*S*) *TYPE*(′*a*⇒′*b*)
⟨*proof*⟩

lemma *synth-intf-of-relI*: *INTF-OF-REL R I* ⟹ *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*) ≫ *f x*

lemma *param-op-mop-iff*:
assumes (*Q*,*P*)∈*R*→*bool-rel*
shows
(*f*, *g*) ∈ [*P*]$_f$ *R* → ⟨*S*⟩*nres-rel*
⟷
(*mop Q f*, *mop P g*) ∈ *R* →$_f$ ⟨*S*⟩*nres-rel*

⟨*proof*⟩

lemma *param-mopI*:
assumes (*f*,*g*) ∈ [*P*]$_f$ *R* → ⟨*S*⟩*nres-rel*
assumes (*Q*,*P*) ∈ *R* → *bool-rel*
shows (*mop Q f*, *mop P g*) ∈ *R* →$_f$ ⟨*S*⟩*nres-rel*
⟨*proof*⟩

lemma *mop-spec-rl*: *P x* ⟹ *mop P f x* ≤ *f x* ⟨*proof*⟩

lemma *mop-spec-rl-from-def*:

assumes $f \equiv mop\ P\ g$
assumes $P\ x$
assumes $g\ x \leq z$
shows $f\ x \leq z$
⟨*proof*⟩

lemma *mop-leof-rl-from-def*:
assumes $f \equiv mop\ P\ g$
assumes $P\ x \Longrightarrow g\ x \leq_n z$
shows $f\ x \leq_n z$
⟨*proof*⟩

lemma *assert-true-bind-conv*: $ASSERT\ True \gg m = m$ ⟨*proof*⟩

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*: $[\![PROP\ P;\ CONSTRAINT\ is\text{-}pure\ A]\!] \Longrightarrow PROP\ P$ ⟨*proof*⟩
lemma *sepref-relpropI*: $P\ R = CONSTRAINT\ P\ R$ ⟨*proof*⟩

Purity

lemmas [*constraint-simps*] = *the-pure-pure*
definition [*constraint-abbrevs*]: $IS\text{-}PURE\ P\ R \equiv is\text{-}pure\ R \wedge P\ (the\text{-}pure\ R)$
lemma *IS-PURE-pureI*:
$P\ R \Longrightarrow IS\text{-}PURE\ P\ (pure\ R)$
⟨*proof*⟩

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

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

lemma *handle-purity1*:
$CONSTRAINT\ (IS\text{-}PURE\ \Phi)\ A \Longrightarrow CONSTRAINT\ \Phi\ (the\text{-}pure\ A)$
⟨*proof*⟩

lemma *handle-purity2*:
$CONSTRAINT\ (IS\text{-}PURE\ \Phi)\ A \Longrightarrow CONSTRAINT\ is\text{-}pure\ A$
⟨*proof*⟩

1.9.4 Composition

Preconditions

definition [*simp*]: *tcomp-pre Q T P* ≡ λ*a*. *Q a* ∧ (∀ *a'*. (*a'*, *a*) ∈ *T* ⟶ *P a'*)
definition *and-pre P1 P2* ≡ λ*x*. *P1 x* ∧ *P2 x*
definition *imp-pre P1 P2* ≡ λ*x*. *P1 x* ⟶ *P2 x*

lemma *and-pre-beta*: *PP* ⟶ *P x* ∧ *Q x* ⟹ *PP* ⟶ *and-pre P Q x* ⟨*proof*⟩
lemma *imp-pre-beta*: *PP* ⟶ *P x* ⟶ *Q x* ⟹ *PP* ⟶ *imp-pre P Q x* ⟨*proof*⟩

definition *IMP-PRE P1 P2* ≡ ∀ *x*. *P1 x* ⟶ *P2 x*
lemma *IMP-PRED*: *IMP-PRE P1 P2* ⟹ *P1 x* ⟹ *P2 x* ⟨*proof*⟩
lemma *IMP-PRE-refl*: *IMP-PRE P P* ⟨*proof*⟩

definition *IMP-PRE-CUSTOM* ≡ *IMP-PRE*
lemma *IMP-PRE-CUSTOMD*: *IMP-PRE-CUSTOM P1 P2* ⟹ *IMP-PRE P1 P2* ⟨*proof*⟩
lemma *IMP-PRE-CUSTOMI*: ⟦⋀*x*. *P1 x* ⟹ *P2 x*⟧ ⟹ *IMP-PRE-CUSTOM P1 P2*
⟨*proof*⟩

lemma *imp-and-triv-pre*: *IMP-PRE P* (*and-pre* (λ-. *True*) *P*)
⟨*proof*⟩

Premises

definition *ALL-LIST A* ≡ (∀ *x*∈*set A*. *x*)
definition *IMP-LIST A B* ≡ *ALL-LIST A* ⟶ *B*

lemma *to-IMP-LISTI*:
P ⟹ *IMP-LIST* [] *P*
⟨*proof*⟩

lemma *to-IMP-LIST*: (*P* ⟹ *IMP-LIST Ps Q*) ≡ *Trueprop* (*IMP-LIST* (*P*#*Ps*) *Q*)
⟨*proof*⟩

lemma *from-IMP-LIST*:
Trueprop (*IMP-LIST As B*) ≡ (*ALL-LIST As* ⟹ *B*)
(*ALL-LIST* [] ⟹ *B*) ≡ *Trueprop B*
(*ALL-LIST* (*A*#*As*) ⟹ *B*) ≡ (*A* ⟹ *ALL-LIST As* ⟹ *B*)
⟨*proof*⟩

lemma *IMP-LIST-trivial*: *IMP-LIST A B* ⟹ *IMP-LIST A B* ⟨*proof*⟩

Composition Rules

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

lemma *transform-pre-param*:
assumes *A*: *IMP-LIST Cns* $((f,\ h) \in [tcomp\text{-}pre\ Q\ T\ P]_a\ hrp\text{-}comp\ RR'\ T \to hr\text{-}comp\ S\ U)$
assumes *P*: *IMP-LIST Cns* $((P,P') \in T \to bool\text{-}rel)$
assumes *C*: *IMP-PRE* $PP'\ (and\text{-}pre\ P'\ Q)$
shows *IMP-LIST Cns* $((f,h) \in [PP']_a\ hrp\text{-}comp\ RR'\ T \to hr\text{-}comp\ S\ U)$
⟨*proof*⟩

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

lemma *hfref-op-to-mop*:
assumes *R*: $(impl,f) \in [Q]_a\ R \to S$
assumes *DEF*: $mf \equiv mop\ P\ f$
assumes *C*: *IMP-PRE* $PP'\ (imp\text{-}pre\ P\ Q)$
shows $(impl,mf) \in [PP']_a\ R \to S$
⟨*proof*⟩

lemma *hfref-mop-to-op*:
assumes *R*: $(impl,mf) \in [Q]_a\ R \to S$
assumes *DEF*: $mf \equiv mop\ P\ f$
assumes *C*: *IMP-PRE* $PP'\ (and\text{-}pre\ Q\ P)$
shows $(impl,f) \in [PP']_a\ R \to S$
⟨*proof*⟩

Precondition Simplification

lemma *IMP-PRE-eqI*:
assumes $\bigwedge x.\ P\ x \longrightarrow Q\ x$
assumes $CNV\ P\ P'$
shows *IMP-PRE* $P'\ Q$
⟨*proof*⟩

lemma *simp-and1*:
assumes $Q \Longrightarrow CNV\ P\ P'$
assumes $PP \longrightarrow P' \wedge Q$
shows $PP \longrightarrow P \wedge Q$
⟨*proof*⟩

lemma *simp-and2*:
assumes $P \Longrightarrow CNV\ Q\ Q'$

assumes $PP \longrightarrow P \wedge Q'$
shows $PP \longrightarrow P \wedge Q$
⟨*proof*⟩

lemma *triv-and1*: $Q \longrightarrow True \wedge Q$ ⟨*proof*⟩

lemma *simp-imp*:
assumes $P \Longrightarrow CNV\ Q\ Q'$
assumes $PP \longrightarrow Q'$
shows $PP \longrightarrow (P \longrightarrow Q)$
⟨*proof*⟩

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

lemma *CNV-prove*:
assumes P
shows $CNV\ P\ True$
⟨*proof*⟩

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

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

1.9.5 Protected Constants

lemma *add-PR-CONST-to-def*: $x \equiv y \Longrightarrow PR\text{-}CONST\ x \equiv y$ ⟨*proof*⟩

1.9.6 Rule Collections

named-theorems-rev *sepref-mop-def-thms* ‹*Sepref*: *mop − definition theorems*›

named-theorems-rev *sepref-fref-thms* ‹*Sepref*: *fref−theorems*›

named-theorems *sepref-relprops-transform* ‹*Sepref*: *Simp−rules to transform relator properties*›
named-theorems *sepref-relprops* ‹*Sepref*: *Simp−rules to add CONSTRAINT−tags to relator properties*›
named-theorems *sepref-relprops-simps* ‹*Sepref*: *Simp−rules to simplify relator properties*›

Default Setup

1.9.7 ML-Level Declarations

⟨*ML*⟩

1.9.8 Obsolete Manual Specification Helpers

lemma *vcg-of-RETURN-np*:
assumes $f \equiv RETURN\ r$
shows $SPEC\ (\lambda x.\ x{=}r) \leq m \Longrightarrow f \leq m$
and $SPEC\ (\lambda x.\ x{=}r) \leq_n m \Longrightarrow f \leq_n m$
⟨*proof*⟩

lemma *vcg-of-RETURN*:
assumes $f \equiv do\ \{\ ASSERT\ \Phi;\ RETURN\ r\ \}$
shows $[\![\Phi;\ SPEC\ (\lambda x.\ x{=}r) \leq m]\!] \Longrightarrow f \leq m$
and $[\![\Phi \Longrightarrow SPEC\ (\lambda x.\ x{=}r) \leq_n m]\!] \Longrightarrow f \leq_n m$
⟨*proof*⟩

lemma *vcg-of-SPEC*:
assumes $f \equiv do\ \{\ ASSERT\ pre;\ SPEC\ post\ \}$
shows $[\![pre;\ SPEC\ post \leq m]\!] \Longrightarrow f \leq m$
and $[\![pre \Longrightarrow SPEC\ post \leq_n m]\!] \Longrightarrow f \leq_n m$
⟨*proof*⟩

lemma *vcg-of-SPEC-np*:
assumes $f \equiv SPEC\ post$
shows $SPEC\ post \leq m \Longrightarrow f \leq m$
and $SPEC\ post \leq_n m \Longrightarrow f \leq_n m$
⟨*proof*⟩

lemma *mk-mop-rl1*:
assumes $\bigwedge x.\ mf\ x \equiv ASSERT\ (P\ x) \gg RETURN\ (f\ x)$
shows $(RETURN\ o\ f,\ mf) \in Id \to \langle Id\rangle nres\text{-}rel$
⟨*proof*⟩

lemma *mk-mop-rl2*:
assumes $\bigwedge x\ y.\ mf\ x\ y \equiv ASSERT\ (P\ x\ y) \gg RETURN\ (f\ x\ y)$
shows $(RETURN\ oo\ f,\ mf) \in Id \to Id \to \langle Id\rangle nres\text{-}rel$
⟨*proof*⟩

lemma *mk-mop-rl3*:
assumes $\bigwedge x\ y\ z.\ mf\ x\ y\ z \equiv ASSERT\ (P\ x\ y\ z) \gg RETURN\ (f\ x\ y\ z)$
shows $(RETURN\ ooo\ f,\ mf) \in Id \to Id \to Id \to \langle Id\rangle nres\text{-}rel$
⟨*proof*⟩

lemma *mk-mop-rl0-np*:
assumes $mf \equiv RETURN\ f$
shows $(RETURN\ f,\ mf) \in \langle Id\rangle nres\text{-}rel$
⟨*proof*⟩

lemma *mk-mop-rl1-np*:
assumes $\bigwedge x.\ mf\ x \equiv RETURN\ (f\ x)$
shows $(RETURN\ o\ f,\ mf) \in Id \to \langle Id\rangle nres\text{-}rel$
⟨*proof*⟩

lemma *mk-mop-rl2-np*:
assumes $\bigwedge x\ y.\ mf\ x\ y \equiv RETURN\ (f\ x\ y)$
shows $(RETURN\ oo\ f,\ mf) \in Id \to Id \to \langle Id\rangle nres\text{-}rel$
⟨*proof*⟩

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

lemma *mk-op-rl0-np*:
assumes $mf \equiv RETURN\ f$
shows $(uncurry0\ mf,\ uncurry0\ (RETURN\ f)) \in unit\text{-}rel \to_f \langle Id\rangle nres\text{-}rel$
⟨*proof*⟩

lemma *mk-op-rl1*:
assumes $\bigwedge x.\ mf\ x \equiv ASSERT\ (P\ x) \gg RETURN\ (f\ x)$
shows $(mf,\ RETURN\ o\ f) \in [P]_f\ Id \to \langle Id\rangle nres\text{-}rel$
⟨*proof*⟩

lemma *mk-op-rl1-np*:
assumes $\bigwedge x.\ mf\ x \equiv RETURN\ (f\ x)$
shows $(mf,\ (RETURN\ o\ f)) \in Id \to_f \langle Id\rangle nres\text{-}rel$
⟨*proof*⟩

lemma *mk-op-rl2*:
assumes $\bigwedge x\ y.\ mf\ x\ y \equiv ASSERT\ (P\ x\ y) \gg RETURN\ (f\ x\ y)$
shows $(uncurry\ mf,\ uncurry\ (RETURN\ oo\ f)) \in [uncurry\ P]_f\ Id \times_r Id \to \langle Id\rangle nres\text{-}rel$
⟨*proof*⟩

lemma *mk-op-rl2-np*:
assumes $\bigwedge x\ y.\ mf\ x\ y \equiv RETURN\ (f\ x\ y)$
shows $(uncurry\ mf,\ uncurry\ (RETURN\ oo\ f)) \in Id \times_r Id \to_f \langle Id\rangle nres\text{-}rel$
⟨*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 $(uncurry2\ mf, uncurry2\ (RETURN\ ooo\ f)) \in [uncurry2\ P]_f\ (Id\times_r Id)\times_r Id \to \langle Id\rangle nres\text{-}rel$
⟨*proof*⟩

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

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-refine* $\Gamma\ c\ \Gamma'\ R\ a$
assumes $\Gamma' \Longrightarrow_t \Gamma c'$
assumes $\bigwedge a\ c.$ *hn-ctxt* $R\ a\ c \Longrightarrow_t$ *hn-ctxt* $Rc\ a\ c$
shows *hn-refine* $\Gamma\ c\ \Gamma c'\ Rc\ a$
⟨*proof*⟩

lemma *ID-init*: $[\![ID\ a\ a'\ TYPE('T);$ *hn-refine* $\Gamma\ c\ \Gamma'\ R\ a']\!]$
$\Longrightarrow$ *hn-refine* $\Gamma\ c\ \Gamma'\ R\ a$ ⟨*proof*⟩

lemma *TRANS-init*: $[\![$ *hn-refine* $\Gamma\ c\ \Gamma'\ R\ a;\ CNV\ c\ c'\ ]\!]$
$\Longrightarrow$ *hn-refine* $\Gamma\ c'\ \Gamma'\ R\ a$
⟨*proof*⟩

lemma *infer-post-triv*: $P \Longrightarrow_t P$ ⟨*proof*⟩

⟨*ML*⟩

Default Optimizer Setup

lemma *return-bind-eq-let*: *do* { *x*←*return v*; *f x* } = *do* { *let x*=*v*; *f x* } ⟨*proof*⟩
lemmas [*sepref-opt-simps*] = *return-bind-eq-let bind-return bind-bind id-def*

We allow the synthesized function to contain tagged function applications. This is important to avoid higher-order unification problems when synthesizing generic algorithms, for example the to-list algorithm for foreach-loops.

lemmas [*sepref-opt-simps*] = *Autoref-Tagging.APP-def*

Revert case-pulling done by monadify

lemma *case-prod-return-opt*[*sepref-opt-simps*]:
case-prod (λ*a b*. *return* (*f a b*)) *p* = *return* (*case-prod f p*)
⟨*proof*⟩

lemma *case-option-return-opt*[*sepref-opt-simps*]:
case-option (*return fn*) (λ*s*. *return* (*fs s*)) *v* = *return* (*case-option fn fs v*)
⟨*proof*⟩

lemma *case-list-return*[*sepref-opt-simps*]:
case-list (*return fn*) (λ*x xs*. *return* (*fc x xs*)) *l* = *return* (*case-list fn fc l*)
⟨*proof*⟩

lemma *if-return*[*sepref-opt-simps*]:
If b (*return t*) (*return e*) = *return* (*If b t e*) ⟨*proof*⟩

In some cases, pushing in the returns is more convenient

lemma *case-prod-opt2*[*sepref-opt-simps2*]:
(λ*x*. *return* (*case x of* (*a*,*b*) ⇒ *f a b*))
= (λ(*a*,*b*). *return* (*f a b*))
⟨*proof*⟩

1.10.2 Debugging Methods

⟨*ML*⟩

1.10.3 Utilities

Manual hfref-proofs

⟨*ML*⟩

Copying of Parameters

lemma *fold-COPY*: *x* = *COPY x* ⟨*proof*⟩

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 $\Longrightarrow$ (*return*, *RETURN o COPY*) $\in R^k \rightarrow_a R$
⟨*proof*⟩

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*)
⟨*proof*⟩

Eliminating higher-order

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

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

Precision Proofs

We provide a method that tries to extract equalities from an assumption of the form - $\models$ *P1* $*$ ... $*$ *Pn* $\wedge_A$ *P1'* $*$ ... $*$ *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* Γ Γ' $\equiv$ *h* $\models$ Γ $*$ *true* $\wedge_A$ Γ' $*$ *true*
lemma *prec-specI*: *h* $\models$ Γ $\wedge_A$ Γ' $\Longrightarrow$ *prec-spec h* Γ Γ'
⟨*proof*⟩

lemma *prec-split1-aux*: *A*$*$*B*$*$*true* $\Longrightarrow_A$ *A*$*$*true*
⟨*proof*⟩

lemma *prec-split2-aux*: *A*$*$*B*$*$*true* $\Longrightarrow_A$ *B*$*$*true*
⟨*proof*⟩

lemma *prec-spec-splitE*:
assumes *prec-spec h* (*A*$*$*B*) (*C*$*$*D*)
obtains *prec-spec h A C prec-spec h B D*
⟨*proof*⟩

lemma *prec-specD*:
assumes *precise R*

assumes *prec-spec* h $(R\ a\ p)$ $(R\ a'\ p)$
shows $a{=}a'$
⟨*proof*⟩

⟨*ML*⟩

Combinator Rules

lemma *split-merge*: $[\![A \vee_A B \Longrightarrow_t X;\ X \vee_A C \Longrightarrow_t D]\!] \Longrightarrow (A \vee_A B \vee_A C \Longrightarrow_t D)$
⟨*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* ≡ *x*
context fixes *A* :: $'a \Rightarrow 'ai \Rightarrow assn$ **begin**
  sepref-register *PR-CONST* (*ASSN-ANNOT A*)
  lemma [*def-pat-rules*]: *ASSN-ANNOT$A* ≡ *UNPROTECT* (*ASSN-ANNOT A*)
⟨*proof*⟩
  lemma [*sepref-fr-rules*]: (*return o* (λ*x*. *x*), *RETURN o PR-CONST* (*ASSN-ANNOT A*)) ∈ $A^d \rightarrow_a A$
    ⟨*proof*⟩
end

lemma *annotate-assn*: *x* ≡ *ASSN-ANNOT A x* ⟨*proof*⟩

2.1.2 Shortcuts

abbreviation *nat-assn* ≡ (*id-assn*::*nat* ⇒ -)
abbreviation *int-assn* ≡ (*id-assn*::*int* ⇒ -)
abbreviation *bool-assn* ≡ (*id-assn*::*bool* ⇒ -)

2.1.3 Identity Relations

definition *IS-ID R* ≡ *R=Id*
definition *IS-BELOW-ID R* ≡ *R⊆Id*

lemma [*safe-constraint-rules*]:
IS-ID Id
IS-ID R1 ⟹ *IS-ID R2* ⟹ *IS-ID* (*R1* → *R2*)
IS-ID R ⟹ *IS-ID* (⟨*R*⟩*option-rel*)
IS-ID R ⟹ *IS-ID* (⟨*R*⟩*list-rel*)
IS-ID R1 ⟹ *IS-ID R2* ⟹ *IS-ID* (*R1* ×$_r$ *R2*)
IS-ID R1 ⟹ *IS-ID R2* ⟹ *IS-ID* (⟨*R1*,*R2*⟩*sum-rel*)
⟨*proof*⟩

lemma [*safe-constraint-rules*]:
IS-BELOW-ID Id
IS-BELOW-ID R ⟹ *IS-BELOW-ID* (⟨*R*⟩*option-rel*)
IS-BELOW-ID R1 ⟹ *IS-BELOW-ID R2* ⟹ *IS-BELOW-ID* (*R1* ×$_r$ *R2*)
IS-BELOW-ID R1 ⟹ *IS-BELOW-ID R2* ⟹ *IS-BELOW-ID* (⟨*R1*,*R2*⟩*sum-rel*)
⟨*proof*⟩

lemma *IS-BELOW-ID-fun-rel-aux*: *R1*⊇*Id* ⟹ *IS-BELOW-ID R2* ⟹ *IS-BELOW-ID* (*R1* → *R2*)
⟨*proof*⟩

corollary *IS-BELOW-ID-fun-rel*[*safe-constraint-rules*]:
IS-ID R1 ⟹ *IS-BELOW-ID R2* ⟹ *IS-BELOW-ID* (*R1* → *R2*)
⟨*proof*⟩

lemma *IS-BELOW-ID-list-rel*[*safe-constraint-rules*]:
IS-BELOW-ID R ⟹ *IS-BELOW-ID* (⟨*R*⟩*list-rel*)
⟨*proof*⟩

lemma *IS-ID-imp-BELOW-ID*[*constraint-rules*]:
IS-ID R ⟹ *IS-BELOW-ID R*
⟨*proof*⟩

2.1.4 Inverse Relation

lemma *inv-fun-rel-eq*[*simp*]: (*A*→*B*)$^{-1}$ = *A*$^{-1}$→*B*$^{-1}$
⟨*proof*⟩

lemma *inv-option-rel-eq*[*simp*]: (⟨*K*⟩*option-rel*)$^{-1}$ = ⟨*K*$^{-1}$⟩*option-rel*
⟨*proof*⟩

lemma *inv-prod-rel-eq*[*simp*]: (*P* ×$_r$ *Q*)$^{-1}$ = *P*$^{-1}$ ×$_r$ *Q*$^{-1}$
⟨*proof*⟩

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

⟨*proof*⟩

lemma *inv-list-rel-eq*[*simp*]: $(\langle R\rangle \textit{list-rel})^{-1} = \langle R^{-1}\rangle \textit{list-rel}$
⟨*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* $R \equiv$ *single-valued* (R^{-1})
definition *IS-LEFT-TOTAL* $R \equiv$ *Domain* $R =$ *UNIV*
definition *IS-RIGHT-TOTAL* $R \equiv$ *Range* $R =$ *UNIV*
abbreviation (*input*) *IS-RIGHT-UNIQUE* $\equiv$ *single-valued*

lemmas *IS-RIGHT-UNIQUED* = *single-valuedD*
lemma *IS-LEFT-UNIQUED*: $[\![$*IS-LEFT-UNIQUE* r; $(y, x) \in r$; $(z, x) \in r]\!] \Longrightarrow$ $y = z$
⟨*proof*⟩

lemma *prop2p*:
IS-LEFT-UNIQUE $R =$ *left-unique* (*rel2p* R)
IS-RIGHT-UNIQUE $R =$ *right-unique* (*rel2p* R)
right-unique (*rel2p* (R^{-1})) = *left-unique* (*rel2p* R)
IS-LEFT-TOTAL $R =$ *left-total* (*rel2p* R)
IS-RIGHT-TOTAL $R =$ *right-total* (*rel2p* R)
⟨*proof*⟩

lemma *p2prop*:
left-unique $P =$ *IS-LEFT-UNIQUE* (*p2rel* P)
right-unique $P =$ *IS-RIGHT-UNIQUE* (*p2rel* P)
left-total $P =$ *IS-LEFT-TOTAL* (*p2rel* P)
right-total $P =$ *IS-RIGHT-TOTAL* (*p2rel* P)
bi-unique $P \longleftrightarrow$ *left-unique* $P \wedge$ *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 $\Longrightarrow$ *IS-LEFT-UNIQUE R2* $\Longrightarrow$ *IS-LEFT-UNIQUE* (*R1*$\times_r$*R2*)
IS-LEFT-UNIQUE R1 $\Longrightarrow$ *IS-LEFT-UNIQUE R2* $\Longrightarrow$ *IS-LEFT-UNIQUE* ($\langle$*R1*,*R2*$\rangle$*sum-rel*)
IS-LEFT-UNIQUE R $\Longrightarrow$ *IS-LEFT-UNIQUE* ($\langle$*R*$\rangle$*option-rel*)
IS-LEFT-UNIQUE R $\Longrightarrow$ *IS-LEFT-UNIQUE* ($\langle$*R*$\rangle$*list-rel*)
$\langle$*proof*$\rangle$

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

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

lemma [*safe-constraint-rules*]:
IS-LEFT-TOTAL Id
IS-LEFT-TOTAL R1 $\Longrightarrow$ *IS-LEFT-TOTAL R2* $\Longrightarrow$ *IS-LEFT-TOTAL* (*R1*$\times_r$*R2*)
IS-LEFT-TOTAL R1 $\Longrightarrow$ *IS-LEFT-TOTAL R2* $\Longrightarrow$ *IS-LEFT-TOTAL* ($\langle$*R1*,*R2*$\rangle$*sum-rel*)
IS-LEFT-TOTAL R $\Longrightarrow$ *IS-LEFT-TOTAL* ($\langle$*R*$\rangle$*option-rel*)
$\langle$*proof*$\rangle$

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

lemma [*safe-constraint-rules*]:
IS-RIGHT-TOTAL Id
IS-RIGHT-TOTAL R1 $\Longrightarrow$ *IS-RIGHT-TOTAL R2* $\Longrightarrow$ *IS-RIGHT-TOTAL* (*R1*$\times_r$*R2*)
IS-RIGHT-TOTAL R1 $\Longrightarrow$ *IS-RIGHT-TOTAL R2* $\Longrightarrow$ *IS-RIGHT-TOTAL* ($\langle$*R1*,*R2*$\rangle$*sum-rel*)
IS-RIGHT-TOTAL R $\Longrightarrow$ *IS-RIGHT-TOTAL* ($\langle$*R*$\rangle$*option-rel*)
$\langle$*proof*$\rangle$

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

lemma [*constraint-simps*]:
IS-LEFT-TOTAL (R^{-1}) $\longleftrightarrow$ *IS-RIGHT-TOTAL R*
IS-RIGHT-TOTAL (R^{-1}) $\longleftrightarrow$ *IS-LEFT-TOTAL R*
IS-LEFT-UNIQUE (R^{-1}) $\longleftrightarrow$ *IS-RIGHT-UNIQUE R*
IS-RIGHT-UNIQUE (R^{-1}) $\longleftrightarrow$ *IS-LEFT-UNIQUE R*
$\langle$*proof*$\rangle$

lemma [*safe-constraint-rules*]:
IS-RIGHT-UNIQUE A $\Longrightarrow$ *IS-RIGHT-TOTAL B* $\Longrightarrow$ *IS-RIGHT-TOTAL* (*A*$\rightarrow$*B*)
IS-RIGHT-TOTAL A $\Longrightarrow$ *IS-RIGHT-UNIQUE B* $\Longrightarrow$ *IS-RIGHT-UNIQUE* (*A*$\rightarrow$*B*)
IS-LEFT-UNIQUE A $\Longrightarrow$ *IS-LEFT-TOTAL B* $\Longrightarrow$ *IS-LEFT-TOTAL* (*A*$\rightarrow$*B*)
IS-LEFT-TOTAL A $\Longrightarrow$ *IS-LEFT-UNIQUE B* $\Longrightarrow$ *IS-LEFT-UNIQUE* (*A*$\rightarrow$*B*)
$\langle$*proof*$\rangle$

lemma [*constraint-rules*]:
IS-BELOW-ID R $\Longrightarrow$ *IS-RIGHT-UNIQUE R*

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

thm *constraint-rules*

Additional Parametricity Lemmas

lemma *param-distinct*[*param*]: ⟦*IS-LEFT-UNIQUE A*; *IS-RIGHT-UNIQUE A*⟧
$\Longrightarrow$ (*distinct*, *distinct*) $\in$ ⟨*A*⟩*list-rel* $\rightarrow$ *bool-rel*
⟨*proof*⟩

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

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

definition *IS-PRES-EQ R* $\equiv$ ((=), (=))∈*R*$\rightarrow$*R*$\rightarrow$*bool-rel*
lemma [*constraint-rules*]: ⟦*single-valued R*; *single-valued* (R^{-1})⟧ $\Longrightarrow$ *IS-PRES-EQ R*
⟨*proof*⟩

2.1.6 Bounded Assertions

definition *b-rel R P* $\equiv$ *R* $\cap$ *UNIV*×*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 (λ-. *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 (λ-. *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-rel R P* $\longleftrightarrow$ $(x,y)\in R \wedge P\ y$
⟨*proof*⟩

lemma *b-assn-simp*[*simp*]: *b-assn A P x y* = $A\ x\ y * \uparrow(P\ x)$
⟨*proof*⟩

lemma *b-rel-Range*[*simp*]: *Range* (*b-rel R P*) = *Range R* $\cap$ *Collect P* ⟨*proof*⟩
lemma *b-assn-rdom*[*simp*]: *rdomp* (*b-assn R P*) *x* $\longleftrightarrow$ *rdomp R x* $\wedge$ *P x*
⟨*proof*⟩

lemma *b-rel-below-id*[*constraint-rules*]:
IS-BELOW-ID R $\Longrightarrow$ *IS-BELOW-ID* (*b-rel R P*)
⟨*proof*⟩

lemma *b-rel-left-unique*[*constraint-rules*]:
IS-LEFT-UNIQUE R $\Longrightarrow$ *IS-LEFT-UNIQUE* (*b-rel R P*)
⟨*proof*⟩

lemma *b-rel-right-unique*[*constraint-rules*]:
IS-RIGHT-UNIQUE R $\Longrightarrow$ *IS-RIGHT-UNIQUE* (*b-rel R P*)
⟨*proof*⟩
lemma *b-assn-is-pure*[*safe-constraint-rules*]:
is-pure A $\Longrightarrow$ *is-pure* (*b-assn A P*)
⟨*proof*⟩
lemma *b-assn-subtyping-match*[*sepref-frame-match-rules*]:
assumes *hn-ctxt* (*b-assn A P*) *x y* $\Longrightarrow_t$ *hn-ctxt A′ x y*
assumes ⟦*vassn-tag* (*hn-ctxt A x y*); *vassn-tag* (*hn-ctxt A′ x y*); *P x*⟧ $\Longrightarrow$ *P′ x*
shows *hn-ctxt* (*b-assn A P*) *x y* $\Longrightarrow_t$ *hn-ctxt* (*b-assn A′ P′*) *x y*
⟨*proof*⟩
lemma *b-assn-subtyping-match-eqA*[*sepref-frame-match-rules*]:
assumes ⟦*vassn-tag* (*hn-ctxt A x y*); *P x*⟧ $\Longrightarrow$ *P′ x*
shows *hn-ctxt* (*b-assn A P*) *x y* $\Longrightarrow_t$ *hn-ctxt* (*b-assn A P′*) *x y*
⟨*proof*⟩

lemma *b-assn-subtyping-match-tR*[*sepref-frame-match-rules*]:
assumes ⟦*P x*⟧ $\Longrightarrow$ *hn-ctxt A x y* $\Longrightarrow_t$ *hn-ctxt A′ x y*
shows *hn-ctxt* (*b-assn A P*) *x y* $\Longrightarrow_t$ *hn-ctxt A′ x y*
⟨*proof*⟩

lemma *b-assn-subtyping-match-tL*[*sepref-frame-match-rules*]:
assumes *hn-ctxt A x y* $\Longrightarrow_t$ *hn-ctxt A′ x y*
assumes ⟦*vassn-tag* (*hn-ctxt A x y*)⟧ $\Longrightarrow$ *P′ x*
shows *hn-ctxt A x y* $\Longrightarrow_t$ *hn-ctxt* (*b-assn A′ P′*) *x y*
⟨*proof*⟩

lemma *b-assn-subtyping-match-eqA-tR*[*sepref-frame-match-rules*]:
hn-ctxt (*b-assn A P*) *x y* $\Longrightarrow_t$ *hn-ctxt A x y*
⟨*proof*⟩

lemma *b-assn-subtyping-match-eqA-tL*[*sepref-frame-match-rules*]:
assumes ⟦*vassn-tag* (*hn-ctxt A x y*)⟧ $\Longrightarrow$ *P′ x*
shows *hn-ctxt A x y* $\Longrightarrow_t$ *hn-ctxt* (*b-assn A P′*) *x y*
⟨*proof*⟩
lemma *b-rel-subtyping-merge*[*sepref-frame-merge-rules*]:
assumes *hn-ctxt A x y* $\vee_A$ *hn-ctxt A′ x y* $\Longrightarrow_t$ *hn-ctxt Am x y*
shows *hn-ctxt* (*b-assn A P*) *x y* $\vee_A$ *hn-ctxt* (*b-assn A′ P′*) *x y* $\Longrightarrow_t$ *hn-ctxt* (*b-assn Am* (λ*x*. *P x* ∨ *P′ x*)) *x y*
⟨*proof*⟩
lemma *b-rel-subtyping-merge-eqA*[*sepref-frame-merge-rules*]:
shows *hn-ctxt* (*b-assn A P*) *x y* $\vee_A$ *hn-ctxt* (*b-assn A P′*) *x y* $\Longrightarrow_t$ *hn-ctxt* (*b-assn A* (λ*x*. *P x* ∨ *P′ x*)) *x y*
⟨*proof*⟩

lemma *b-rel-subtyping-merge-tL*[*sepref-frame-merge-rules*]:
assumes *hn-ctxt A x y* $\vee_A$ *hn-ctxt A′ x y* $\Longrightarrow_t$ *hn-ctxt Am x y*
shows *hn-ctxt A x y* $\vee_A$ *hn-ctxt* (*b-assn A′ P′*) *x y* $\Longrightarrow_t$ *hn-ctxt Am x y*
⟨*proof*⟩

lemma *b-rel-subtyping-merge-tR*[*sepref-frame-merge-rules*]:
assumes *hn-ctxt A x y* $\vee_A$ *hn-ctxt A′ x y* $\Longrightarrow_t$ *hn-ctxt Am x y*
shows *hn-ctxt* (*b-assn A P*) *x y* $\vee_A$ *hn-ctxt A′ x y* $\Longrightarrow_t$ *hn-ctxt Am x y*
⟨*proof*⟩

lemma *b-rel-subtyping-merge-eqA-tL*[*sepref-frame-merge-rules*]:
shows *hn-ctxt A x y* $\vee_A$ *hn-ctxt* (*b-assn A P′*) *x y* $\Longrightarrow_t$ *hn-ctxt A x y*
⟨*proof*⟩

lemma *b-rel-subtyping-merge-eqA-tR*[*sepref-frame-merge-rules*]:
shows *hn-ctxt* (*b-assn A P*) *x y* $\vee_A$ *hn-ctxt A x y* $\Longrightarrow_t$ *hn-ctxt A x y*
⟨*proof*⟩

lemma *b-assn-invalid-merge1*: *hn-invalid* (*b-assn A P*) *x y* $\vee_A$ *hn-invalid* (*b-assn A P′*) *x y*
$\Longrightarrow_t$ *hn-invalid* (*b-assn A* (λ*x*. *P x* ∨ *P′ x*)) *x y*
⟨*proof*⟩

lemma *b-assn-invalid-merge2*: *hn-invalid* (*b-assn A P*) *x y* $\vee_A$ *hn-invalid A x y*
$\Longrightarrow_t$ *hn-invalid A x y*
⟨*proof*⟩
lemma *b-assn-invalid-merge3*: *hn-invalid A x y* $\vee_A$ *hn-invalid* (*b-assn A P*) *x y*
$\Longrightarrow_t$ *hn-invalid A x y*
⟨*proof*⟩

lemma *b-assn-invalid-merge4*: *hn-invalid* (*b-assn A P*) *x y* $\vee_A$ *hn-ctxt* (*b-assn A P'*) *x y*
$\Longrightarrow_t$ *hn-invalid* (*b-assn A* ($\lambda x.$ *P x* $\vee$ *P' x*)) *x y*
⟨*proof*⟩
lemma *b-assn-invalid-merge5*: *hn-ctxt* (*b-assn A P'*) *x y* $\vee_A$ *hn-invalid* (*b-assn A P*) *x y*
$\Longrightarrow_t$ *hn-invalid* (*b-assn A* ($\lambda x.$ *P x* $\vee$ *P' x*)) *x y*
⟨*proof*⟩

lemma *b-assn-invalid-merge6*: *hn-invalid* (*b-assn A P*) *x y* $\vee_A$ *hn-ctxt A x y*
$\Longrightarrow_t$ *hn-invalid A x y*
⟨*proof*⟩
lemma *b-assn-invalid-merge7*: *hn-ctxt A x y* $\vee_A$ *hn-invalid* (*b-assn A P*) *x y*
$\Longrightarrow_t$ *hn-invalid A x y*
⟨*proof*⟩

lemma *b-assn-invalid-merge8*: *hn-ctxt* (*b-assn A P*) *x y* $\vee_A$ *hn-invalid A x y*
$\Longrightarrow_t$ *hn-invalid A x y*
⟨*proof*⟩
lemma *b-assn-invalid-merge9*: *hn-invalid A x y* $\vee_A$ *hn-ctxt* (*b-assn A P*) *x y*
$\Longrightarrow_t$ *hn-invalid A x y*
⟨*proof*⟩

lemmas *b-assn-invalid-merge*[*sepref-frame-merge-rules*] =
b-assn-invalid-merge1
b-assn-invalid-merge2
b-assn-invalid-merge3
b-assn-invalid-merge4
b-assn-invalid-merge5
b-assn-invalid-merge6
b-assn-invalid-merge7
b-assn-invalid-merge8
b-assn-invalid-merge9

abbreviation *nbn-rel* :: *nat* ⇒ (*nat* × *nat*) *set*
— Natural numbers with upper bound.
where *nbn-rel n* ≡ *b-rel nat-rel* (λx::*nat*. *x*<*n*)

abbreviation *nbn-assn* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *assn*
— Natural numbers with upper bound.
where *nbn-assn n* ≡ *b-assn nat-assn* (λx::*nat*. *x*<*n*)

2.1.7 Tool Setup

lemmas [*sepref-relprops*] =
sepref-relpropI[*of IS-LEFT-UNIQUE*]
sepref-relpropI[*of IS-RIGHT-UNIQUE*]
sepref-relpropI[*of IS-LEFT-TOTAL*]
sepref-relpropI[*of IS-RIGHT-TOTAL*]
sepref-relpropI[*of is-pure*]
sepref-relpropI[*of IS-PURE* Φ **for** Φ]
sepref-relpropI[*of IS-ID*]
sepref-relpropI[*of IS-BELOW-ID*]

lemma [*sepref-relprops-simps*]:
CONSTRAINT (*IS-PURE IS-ID*) *A* ⟹ *CONSTRAINT* (*IS-PURE IS-BELOW-ID*) *A*
CONSTRAINT (*IS-PURE IS-ID*) *A* ⟹ *CONSTRAINT* (*IS-PURE IS-LEFT-TOTAL*) *A*
CONSTRAINT (*IS-PURE IS-ID*) *A* ⟹ *CONSTRAINT* (*IS-PURE IS-RIGHT-TOTAL*) *A*
CONSTRAINT (*IS-PURE IS-BELOW-ID*) *A* ⟹ *CONSTRAINT* (*IS-PURE IS-LEFT-UNIQUE*) *A*
CONSTRAINT (*IS-PURE IS-BELOW-ID*) *A* ⟹ *CONSTRAINT* (*IS-PURE IS-RIGHT-UNIQUE*) *A*
⟨*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*: Γ ⟹$_t$ Γ*1* ∗ *hn-val bool-rel a a′*
assumes *RT*: *a* ⟹ *hn-refine* (Γ*1* ∗ *hn-val bool-rel a a′*) *b′* Γ*2b R b*
assumes *RE*: ¬*a* ⟹ *hn-refine* (Γ*1* ∗ *hn-val bool-rel a a′*) *c′* Γ*2c R c*
assumes *IMP*: *TERM If* ⟹ Γ*2b* ∨$_A$ Γ*2c* ⟹$_t$ Γ′
shows *hn-refine* Γ (*if a′ then b′ else c′*) Γ′ *R* (*If*$*a*$*b*$*c*)
⟨*proof*⟩

lemmas [*sepref-opt-simps*] = *if-True if-False*

lemma *hn-let*[*sepref-comb-rules*]:
assumes *P*: Γ ⟹$_t$ Γ*1* ∗ *hn-ctxt R v v′*
assumes *R*: ⋀*x x′*. *x*=*v* ⟹ *hn-refine* (Γ*1* ∗ *hn-ctxt R x x′*) (*f′ x′*) (Γ′ *x x′*) *R2* (*f x*)
assumes *F*: ⋀*x x′*. Γ′ *x x′* ⟹$_t$ Γ*2* ∗ *hn-ctxt R′ x x′*
shows

hn-refine Γ (*Let* v' f') (Γ2 ∗ *hn-ctxt* R' v v') *R2* (*Let*v(λ$_2$$x$. f x))
⟨*proof*⟩

2.1.9 Basic HOL types

lemma *hnr-default*[*sepref-import-param*]: (*default*,*default*)∈*Id* ⟨*proof*⟩

lemma *unit-hnr*[*sepref-import-param*]: ((),())∈*unit-rel* ⟨*proof*⟩

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* ⇒ *nat*)]
itypeI[*Pure.of numeral TYPE* (*num* ⇒ *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*)∈*nat-rel* → *nat-rel* → *nat-rel* ⟨*proof*⟩
lemma *param-max-nat*[*param*,*sepref-import-param*]: (*max*,*max*)∈*nat-rel* → *nat-rel* → *nat-rel* ⟨*proof*⟩

lemma *param-min-int*[*param*,*sepref-import-param*]: (*min*,*min*)∈*int-rel* → *int-rel* → *int-rel* ⟨*proof*⟩
lemma *param-max-int*[*param*,*sepref-import-param*]: (*max*,*max*)∈*int-rel* → *int-rel* → *int-rel* ⟨*proof*⟩

lemma *uminus-hnr*[*sepref-import-param*]: (*uminus*,*uminus*)∈*int-rel* → *int-rel* ⟨*proof*⟩

lemma *nat-param*[*param*,*sepref-import-param*]: (*nat*,*nat*) ∈ *int-rel* → *nat-rel* ⟨*proof*⟩
lemma *int-param*[*param*,*sepref-import-param*]: (*int*,*int*) ∈ *nat-rel* → *int-rel* ⟨*proof*⟩

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 ⟹ *precise P2* ⟹ *precise* (*prod-assn P1 P2*)
⟨*proof*⟩

lemma

precise P1 $\Longrightarrow$ *precise P2* $\Longrightarrow$ *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*) $\Longrightarrow_t$ *hn-ctxt A'* (*fst x*) (*fst y*)
assumes *hn-ctxt B* (*snd x*) (*snd y*) $\Longrightarrow_t$ *hn-ctxt B'* (*snd x*) (*snd y*)
shows *hn-ctxt* (*prod-assn A B*) *x y* $\Longrightarrow_t$ *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*) $\Longrightarrow_t$ *hn-ctxt Am* (*fst x*) (*fst y*)
assumes *hn-ctxt B* (*snd x*) (*snd y*) $\vee_A$ *hn-ctxt B'* (*snd x*) (*snd y*) $\Longrightarrow_t$ *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* $\Longrightarrow_t$ *hn-ctxt* (*prod-assn Am Bm*) *x y*
⟨*proof*⟩

lemma *entt-invalid-prod*: *hn-invalid* (*prod-assn A B*) *p p'* $\Longrightarrow_t$ *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* $\Longrightarrow$ *hn-ctxt* (*prod-assn A1 A2*) *x y* = *z*
⟨*proof*⟩

lemma *hn-case-prod'*[*sepref-prep-comb-rule*,*sepref-comb-rules*]:
assumes *FR*: Γ$\Longrightarrow_t$*hn-ctxt* (*prod-assn P1 P2*) *p' p* ∗ Γ*1*
assumes *Pair*: $\bigwedge$*a1 a2 a1' a2'*. ⟦*p'*=(*a1'*,*a2'*)⟧
$\Longrightarrow$ *hn-refine* (*hn-ctxt P1 a1' a1* ∗ *hn-ctxt P2 a2' a2* ∗ Γ*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* ∗ Γ*1'*) *R* (*f' a1' a2'*)
shows *hn-refine* Γ (*case-prod f p*) (*hn-ctxt* (*prod-assn P1' P2'*) *p' p* ∗ Γ*1'*) *R* (*case-prod*$(λ$_2$*a b*. *f' a b*)$*p'*) (**is** *?G* Γ)
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*: Γ⟹$_t$Γ*1* ∗ *hn-ctxt* (*prod-assn P1 P2*) *p' p*
assumes *R*: ⋀*a1 a2 a1' a2'*. ⟦*p'*=(*a1'*,*a2'*)⟧
⟹ *hn-refine* (Γ*1* ∗ *hn-ctxt P1 a1' a1* ∗ *hn-ctxt P2 a2' a2* ∗ *hn-invalid* (*prod-assn P1 P2*) *p' p*) (*f a1 a2*)
(Γ*h a1 a1' a2 a2'*) *R* (*f' a1' a2'*)
assumes *M*: ⋀*a1 a1' a2 a2'*. Γ*h a1 a1' a2 a2'*
⟹$_t$ Γ' ∗ *hn-ctxt P1' a1' a1* ∗ *hn-ctxt P2' a2' a2* ∗ *hn-ctxt Pxx p' p*
shows *hn-refine* Γ (*case-prod f p*) (Γ' ∗ *hn-ctxt* (*prod-assn P1' P2'*) *p' p*)
R (*case-prod*$(λ$_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*
(*hn-ctxt P1 x1 x1'* ∗ *hn-ctxt P2 x2 x2'*)
(*return* (*x1'*,*x2'*))
(*hn-invalid P1 x1 x1'* ∗ *hn-invalid P2 x2 x2'*)
(*prod-assn P1 P2*)
(*RETURN*$(*Pair*$*x1*$*x2*))
⟨*proof*⟩

lemma *fst-hnr*[*sepref-fr-rules*]: (*return o fst*,*RETURN o fst*) ∈ (*prod-assn A B*)^d →$_a$ *A*
⟨*proof*⟩
lemma *snd-hnr*[*sepref-fr-rules*]: (*return o snd*,*RETURN o snd*) ∈ (*prod-assn A B*)^d →$_a$ *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*) ⟹ *rdomp A a* ∧ *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* ≡ *hn-invalid* (*option-assn P*) *p p'*
assumes *FR*: Γ ⟹$_t$ *hn-ctxt* (*option-assn P*) *p p'* ∗ *F*
assumes *Rn*: *p*=*None* ⟹ *hn-refine* (*hn-ctxt* (*option-assn P*) *p p'* ∗ *F*) *f1'* (*hn-ctxt XX1 p p'* ∗ Γ*1'*) *R f1*
assumes *Rs*: ⋀*x x'*. ⟦ *p*=*Some x*; *p'*=*Some x'* ⟧ ⟹
hn-refine (*hn-ctxt P x x'* ∗ *INVE* ∗ *F*) (*f2' x'*) (*hn-ctxt P' x x'* ∗ *hn-ctxt XX2 p p'* ∗ Γ*2'*) *R* (*f2 x*)
assumes *MERGE1*: Γ*1'* ∨$_A$ Γ*2'* ⟹$_t$ Γ'
shows *hn-refine* Γ (*case-option f1' f2' p'*) (*hn-ctxt* (*option-assn P'*) *p p'* ∗ Γ') *R* (*case-option*$*f1*$(λ$_2$*x*. *f2 x*)$*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*)
applyS (*simp add*: *hn-ctxt-def*)
apply1 (*rule entt-trans*[*OF - MERGE1*])
applyS (*simp add*: *hn-ctxt-def*)
⟨*proof*⟩

lemma *hn-None*[*sepref-fr-rules*]:
hn-refine emp (*return None*) *emp* (*option-assn P*) (*RETURN*$*None*)
⟨*proof*⟩

lemma *hn-Some*[*sepref-fr-rules*]: *hn-refine*
(*hn-ctxt P v v'*)
(*return* (*Some v'*))
(*hn-invalid P v v'*)
(*option-assn P*)
(*RETURN*$(*Some*$*v*))
⟨*proof*⟩

definition *imp-option-eq eq a b* ≡ *case* (*a*,*b*) *of*
(*None*,*None*) ⇒ *return True*
| (*Some a*, *Some b*) ⇒ *eq a b*
| - ⇒ *return False*

lemma *option-assn-eq*[*sepref-comb-rules*]:
fixes *a b* :: *'a option*
assumes *F1*: Γ ⟹$_t$ *hn-ctxt* (*option-assn P*) *a a'* ∗ *hn-ctxt* (*option-assn P*) *b b'* ∗ Γ*1*

assumes *EQ*: $\bigwedge va\ va'\ vb\ vb'$. *hn-refine*
(*hn-ctxt P va va'* $*$ *hn-ctxt P vb vb'* $*$ $\Gamma 1$)
(*eq' va' vb'*)
(Γ' *va va' vb vb'*)
bool-assn
(*RETURN*$((=)\ \$va\$vb)$)
assumes *F2*:
$\bigwedge va\ va'\ vb\ vb'$.
$\Gamma'\ va\ va'\ vb\ vb' \Longrightarrow_t$ *hn-ctxt P va va'* $*$ *hn-ctxt P vb vb'* $*$ $\Gamma 1$
shows *hn-refine*
Γ
(*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*$((=)\ \$a\$b)$)
⟨*proof*⟩

lemma [*pat-rules*]:
(=) $\$a\$None \equiv$ *is-None*$\$a$
(=) $\$None\$a \equiv$ *is-None*$\$a$
⟨*proof*⟩

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*$(is\text{-}None\$a)$)
⟨*proof*⟩

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*$(the\$x)$)
⟨*proof*⟩

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*$(the\$x)$)

⟨*proof*⟩

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

lemma *list-assn-simps*[*simp*]:
hn-ctxt (*list-assn P*) [] *l*′ = (↑(*l*′=[]))
hn-ctxt (*list-assn P*) *l* [] = (↑(*l*=[]))
hn-ctxt (*list-assn P*) [] [] = *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*:
⟦⋀*x x'*. *P x x'*⟹$_A$*P' x x'*⟧ ⟹ *list-assn P l l'* ⟹$_A$ *list-assn P' l l'*
⟨*proof*⟩

lemma *list-assn-monot*:
⟦⋀*x x'*. *P x x'*⟹$_t$*P' x x'*⟧ ⟹ *list-assn P l l'* ⟹$_t$ *list-assn P' l l'*
⟨*proof*⟩

lemma *list-match-cong*[*sepref-frame-match-rules*]:
⟦⋀*x x'*. ⟦*x*∈*set l*; *x'*∈*set l'*⟧ ⟹ *hn-ctxt A x x'* ⟹$_t$ *hn-ctxt A' x x'* ⟧ ⟹ *hn-ctxt* (*list-assn A*) *l l'* ⟹$_t$ *hn-ctxt* (*list-assn A'*) *l l'*
⟨*proof*⟩

lemma *list-merge-cong*[*sepref-frame-merge-rules*]:
assumes ⋀*x x'*. ⟦*x*∈*set l*; *x'*∈*set l'*⟧ ⟹ *hn-ctxt A x x'* ∨$_A$ *hn-ctxt A' x x'* ⟹$_t$ *hn-ctxt Am x x'*
shows *hn-ctxt* (*list-assn A*) *l l'* ∨$_A$ *hn-ctxt* (*list-assn A'*) *l l'* ⟹$_t$ *hn-ctxt* (*list-assn Am*) *l l'*
⟨*proof*⟩

lemma *invalid-list-split*:
invalid-assn (*list-assn A*) (*x*#*xs*) (*y*#*ys*) ⟹$_t$ *invalid-assn A x y* ∗ *invalid-assn* (*list-assn A*) *xs ys*
⟨*proof*⟩

lemma *entt-invalid-list*: *hn-invalid* (*list-assn A*) *l l'* ⟹$_t$ *hn-ctxt* (*list-assn* (*invalid-assn A*)) *l 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*]: *hr-comp* (*list-assn A*) (⟨*B*⟩*list-rel*) = *list-assn* (*hr-comp A B*)
⟨*proof*⟩

lemma *hn-ctxt-eq*: *A x y* = *z* ⟹ *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* ≡ *hn-invalid* (*list-assn P*) *p p'*
assumes *FR*: Γ ⟹$_t$ *hn-ctxt* (*list-assn P*) *p p'* ∗ *F*
assumes *Rn*: *p*=[] ⟹ *hn-refine* (*hn-ctxt* (*list-assn P*) *p p'* ∗ *F*) *f1'* (*hn-ctxt XX1 p p'* ∗ Γ*1'*) *R f1*
assumes *Rs*: ⋀*x l x' l'*. ⟦ *p*=*x*#*l*; *p'*=*x'*#*l'* ⟧ ⟹
hn-refine (*hn-ctxt P x x'* ∗ *hn-ctxt* (*list-assn P*) *l l'* ∗ *INVE* ∗ *F*) (*f2' x' l'*) (*hn-ctxt P1' x x'* ∗ *hn-ctxt* (*list-assn P2'*) *l l'* ∗ *hn-ctxt XX2 p p'* ∗ Γ*2'*) *R* (*f2 x l*)
assumes *MERGE1*[*unfolded hn-ctxt-def*]: ⋀*x x'*. *hn-ctxt P1' x x'* ∨$_A$ *hn-ctxt P2' x x'* ⟹$_t$ *hn-ctxt P' x x'*
assumes *MERGE2*: Γ*1'* ∨$_A$ Γ*2'* ⟹$_t$ Γ'
shows *hn-refine* Γ (*case-list f1' f2' p'*) (*hn-ctxt* (*list-assn P'*) *p p'* ∗ Γ') *R* (*case-list*$*f1*$(λ$_2$*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*]:

hn-refine emp (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$((#) $x$xs))
⟨*proof*⟩

lemma *list-assn-aux-len*:
list-assn P l l' = list-assn P l l' ∗ ↑(length l = length l')
⟨*proof*⟩

lemma *list-assn-aux-eqlen-simp*:
vassn-tag (list-assn P l l') ⟹ length l' = length l
h ⊨ (list-assn P l l') ⟹ length l' = length l
⟨*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 = (∃ A b m'. R a b ∗ list-assn R l m' ∗ ↑(m=b#m'))
⟨*proof*⟩
lemma *list-assn-aux-cons-conv2*:
list-assn R l (b#m) = (∃ A a l'. R a b ∗ list-assn R l' m ∗ ↑(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 = (∃ A m1 m2. list-assn R l1 m1 ∗ list-assn R l2 m2 ∗ ↑(m=m1@m2))
⟨*proof*⟩
lemma *list-assn-aux-append-conv2*:
list-assn R l (m1@m2) = (∃ A l1 l2. list-assn R l1 m1 ∗ list-assn R l2 m2 ∗ ↑(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* ⇒ '*a* ⇒ *assn*) ⇒ ('*bi* ⇒ '*b* ⇒ *assn*) ⇒ ('*ai*+'*bi*) ⇒ ('*a*+'*b*)
⇒ *assn* **where**
  sum-assn A B (*Inl ai*) (*Inl a*) = *A ai a*
| *sum-assn A B* (*Inr bi*) (*Inr b*) = *B bi b*
| *sum-assn A B* - - = *false*

notation *sum-assn* (**infixr** +$_a$ *67*)

lemma *sum-assn-pure*[*safe-constraint-rules*]: ⟦*is-pure A*; *is-pure B*⟧ ⟹ *is-pure* (*sum-assn A B*)
  ⟨*proof*⟩

lemma *sum-assn-id*[*simp*]: *sum-assn id-assn id-assn* = *id-assn*
  ⟨*proof*⟩

lemma *sum-assn-pure-conv*[*simp*]: *sum-assn* (*pure A*) (*pure B*) = *pure* (⟨*A*,*B*⟩*sum-rel*)
  ⟨*proof*⟩

lemma *sum-match-cong*[*sepref-frame-match-rules*]:
  ⟦
    ⋀*x y*. ⟦*e* = *Inl x*; *e'*=*Inl y*⟧ ⟹ *hn-ctxt A x y* ⟹$_t$ *hn-ctxt A' x y*;
    ⋀*x y*. ⟦*e* = *Inr x*; *e'*=*Inr y*⟧ ⟹ *hn-ctxt B x y* ⟹$_t$ *hn-ctxt B' x y*
  ⟧ ⟹ *hn-ctxt* (*sum-assn A B*) *e e'* ⟹$_t$ *hn-ctxt* (*sum-assn A' B'*) *e e'*
  ⟨*proof*⟩

lemma *enum-merge-cong*[*sepref-frame-merge-rules*]:
  assumes ⋀*x y*. ⟦*e*=*Inl x*; *e'*=*Inl y*⟧ ⟹ *hn-ctxt A x y* ∨$_A$ *hn-ctxt A' x y* ⟹$_t$ *hn-ctxt Am x y*
  assumes ⋀*x y*. ⟦*e*=*Inr x*; *e'*=*Inr y*⟧ ⟹ *hn-ctxt B x y* ∨$_A$ *hn-ctxt B' x y* ⟹$_t$ *hn-ctxt Bm x y*
  shows *hn-ctxt* (*sum-assn A B*) *e e'* ∨$_A$ *hn-ctxt* (*sum-assn A' B'*) *e e'* ⟹$_t$ *hn-ctxt* (*sum-assn Am Bm*) *e e'*
  ⟨*proof*⟩

lemma *entt-invalid-sum*: *hn-invalid* (*sum-assn A B*) *e e'* ⟹$_t$ *hn-ctxt* (*sum-assn* (*invalid-assn A*) (*invalid-assn B*)) *e e'*
  ⟨*proof*⟩

lemmas *invalid-sum-merge*[*sepref-frame-merge-rules*] = *gen-merge-cons*[*OF entt-invalid-sum*]

sepref-register *Inr Inl*

lemma [*sepref-fr-rules*]: (*return o Inl*,*RETURN o Inl*) ∈ A^d →$_a$ *sum-assn A B*
  ⟨*proof*⟩
lemma [*sepref-fr-rules*]: (*return o Inr*,*RETURN o Inr*) ∈ B^d →$_a$ *sum-assn A B*
  ⟨*proof*⟩

sepref-register *case-sum*

In the monadify phase, this eta-expands to make visible all required arguments

lemma [*sepref-monadify-arity*]: *case-sum* ≡ $\lambda_2 f1\ f2\ x.$ *SP case-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-sum*\$*f1*\$*f2*\$*x* ≡ (≫=) \$(*EVAL*\$*x*)\$($\lambda_2 x.$ *SP case-sum*\$*f1*\$*f2*\$*x*) ⟨*proof*⟩

This enables translation of the case-distinction in a non-monadic context.

lemma [*sepref-monadify-comb*]: *EVAL*\$(*case-sum*\$($\lambda_2 x.$ *f1 x*)\$($\lambda_2 x.$ *f2 x*)\$*x*)
≡ (≫=) \$(*EVAL*\$*x*)\$($\lambda_2 x.$ *SP case-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-assn A B x y* = *z* ⟹ *hn-ctxt* (*sum-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* ≡ *hn-invalid* (*sum-assn A B*) *e e′*
assumes *FR*: Γ ⟹$_t$ *hn-ctxt* (*sum-assn A B*) *e e′* ∗ *F*
assumes *E1*: ⋀*x1 x1a*. ⟦*e* = *Inl x1*; *e′* = *Inl x1a*⟧ ⟹ *hn-refine* (*hn-ctxt A x1 x1a* ∗ *INVe* ∗ *F*) (*f1′ x1a*) (*hn-ctxt A′ x1 x1a* ∗ *hn-ctxt XX1 e e′* ∗ Γ*1′*) *R* (*f1 x1*)
assumes *E2*: ⋀*x2 x2a*. ⟦*e* = *Inr x2*; *e′* = *Inr x2a*⟧ ⟹ *hn-refine* (*hn-ctxt B x2 x2a* ∗ *INVe* ∗ *F*) (*f2′ x2a*) (*hn-ctxt B′ x2 x2a* ∗ *hn-ctxt XX2 e e′* ∗ Γ*2′*) *R* (*f2 x2*)
assumes *MERGE*[*unfolded hn-ctxt-def*]: Γ*1′* ∨$_A$ Γ*2′* ⟹$_t$ Γ′
shows *hn-refine* Γ (*case-sum f1′ f2′ e′*) (*hn-ctxt* (*sum-assn A′ B′*) *e e′* ∗ Γ′) *R* (*case-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*)) ∈ *Id*
⟨*proof*⟩

end

lemma *Literal-pat* [*def-pat-rules*]:
String.Literal $ *b0* $ *b1* $ *b2* $ *b3* $ *b4* $ *b5* $ *b6* $ *s* ≡
UNPROTECT (*String.Literal* $ *b0* $ *b1* $ *b2* $ *b3* $ *b4* $ *b5* $ *b6* $ *s*)
⟨*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 $\lambda x.\ RETURN\ (c\ x)$

definition *FOREACH-inv xs* Φ *s* ≡
case s of (*it*, σ) ⇒ ∃*xs'*. *xs* = *xs'* @ *it* ∧ Φ (*set it*) σ

definition *monadic-FOREACH R* Φ *S c f* σ*0* ≡ *do* {
ASSERT (*finite S*);
xs0 ← *it-to-sorted-list R S*;
(-,σ) ← *RECT* (λ*W* (*xs*,σ). *do* {
ASSERT (*FOREACH-inv xs0* Φ (*xs*,σ));
if xs≠[] *then do* {
b ← *c* σ;
if b then
FOREACH-body f (*xs*,σ) ≫= *W*
else
RETURN (*xs*,σ)
} *else RETURN* (*xs*,σ)
}) (*xs0*,σ*0*);
RETURN σ
}

lemma *FOREACH-oci-to-monadic*:
FOREACHoci R Φ S c f σ0 = monadic-FOREACH R Φ S (λσ. RETURN (c σ)) f σ0
⟨*proof*⟩

Next, we define a characterization w.r.t. *nfoldli*

definition *monadic-nfoldli l c f s ≡ RECT (λD (l,s). case l of*
[] ⇒ RETURN s
| x#ls ⇒ do {
b ← c s;
if b then do { s'←f x s; D (ls,s')} else RETURN s
}
) (l,s)

lemma *monadic-nfoldli-eq*:
monadic-nfoldli l c f s = (
case l of
[] ⇒ RETURN s
| x#ls ⇒ do {
b←c s;
if b then f x s ≫= monadic-nfoldli ls c f else RETURN s
}
)
⟨*proof*⟩

lemma *monadic-nfoldli-simp*[*simp*]:
monadic-nfoldli [] c f s = RETURN s
monadic-nfoldli (x#ls) c f s = do {
b←c s;
if b then f x s ≫= monadic-nfoldli ls c f else RETURN s
}
⟨*proof*⟩

lemma *nfoldli-to-monadic*:
nfoldli l c f = monadic-nfoldli l (λx. RETURN (c x)) f
⟨*proof*⟩

definition *nfoldli-alt l c f s ≡ RECT (λD (l,s). case l of*
[] ⇒ RETURN s
| x#ls ⇒ do {
let b = c s;
if b then do { s'←f x s; D (ls,s')} else RETURN s
}
) (l,s)

lemma *nfoldli-alt-eq*:
nfoldli-alt l c f s = (
case l of
[] ⇒ RETURN s

```
  | x#ls ⇒ do {let b=c s; if b then f x s ≫= nfoldli-alt ls c f else RETURN s}
)
⟨proof⟩

lemma nfoldli-alt-simp[simp]:
  nfoldli-alt [] c f s = RETURN s
  nfoldli-alt (x#ls) c f s = do {
    let b = c s;
    if b then f x s ≫= nfoldli-alt ls c f else RETURN s
  }
  ⟨proof⟩


lemma nfoldli-alt:
  (nfoldli::'a list ⇒ ('b ⇒ bool) ⇒ ('a ⇒ 'b ⇒ 'b nres) ⇒ 'b ⇒ 'b nres)
  = nfoldli-alt
⟨proof⟩

lemma monadic-nfoldli-rec:
  monadic-nfoldli x' c f σ
        ≤⇓Id (REC_T
          (λW (xs, σ).
              ASSERT (FOREACH-inv xs0 I (xs, σ)) ≫=
              (λ-. if xs = [] then RETURN (xs, σ)
                   else c σ ≫=
                        (λb. if b then FOREACH-body f (xs, σ) ≫= W
                             else RETURN (xs, σ))))
          (x', σ) ≫=
          (λ(-, y). RETURN y))
  ⟨proof⟩

lemma monadic-nfoldli-arities[sepref-monadify-arity]:
  monadic-nfoldli ≡ λ2s c f σ. SP (monadic-nfoldli)$s$(λ2x. c$x)$(λ2x σ. f$x$σ)$σ
  ⟨proof⟩

lemma monadic-nfoldli-comb[sepref-monadify-comb]:
  ⋀s c f σ. (monadic-nfoldli)$s$c$f$σ ≡
    Refine-Basic.bind$(EVAL$s)$(λ2s. Refine-Basic.bind$(EVAL$σ)$(λ2σ.
      SP (monadic-nfoldli)$s$c$f$σ
    ))
  ⟨proof⟩

lemma list-rel-congD:
  assumes A: (li,l)∈⟨S⟩list-rel
  shows (li,l)∈⟨S∩(set li×set l)⟩list-rel
⟨proof⟩

lemma monadic-nfoldli-refine[refine]:
  assumes L: (li, l) ∈ ⟨S⟩list-rel
```

and [*simp*]: $(si,\ s) \in R$
and *CR*[*refine*]: $\bigwedge si\ s.\ (si,s)\in R \Longrightarrow ci\ si \leq \Downarrow bool\text{-}rel\ (c\ s)$
and [*refine*]: $\bigwedge xi\ x\ si\ s.\ [\![\ (xi,x)\in S;\ x\in set\ l;\ (si,s)\in R;\ inres\ (c\ s)\ True\ ]\!] \Longrightarrow fi\ xi\ si \leq \Downarrow R\ (f\ x\ s)$
shows *monadic-nfoldli li ci fi si* $\leq \Downarrow$ *R* (*monadic-nfoldli l c f s*)

⟨*proof*⟩

lemma *monadic-FOREACH-itsl*:
fixes *R I tsl*
shows
do { $l \leftarrow$ *it-to-sorted-list R s*; *monadic-nfoldli l c f* σ }
$\leq$ *monadic-FOREACH R I s c f* σ
⟨*proof*⟩

lemma *FOREACHoci-itsl*:
fixes *R I tsl*
shows
do { $l \leftarrow$ *it-to-sorted-list R s*; *nfoldli l c f* σ }
$\leq$ *FOREACHoci R I s c f* σ
⟨*proof*⟩

lemma [*def-pat-rules*]:
FOREACHc $\equiv$ *PR-CONST* (*FOREACHoci* (λ- -. *True*) (λ- -. *True*))
FOREACHci\$*I* $\equiv$ *PR-CONST* (*FOREACHoci* (λ- -. *True*) *I*)
FOREACHi\$*I* $\equiv \lambda_2 s$. *PR-CONST* (*FOREACHoci* (λ- -. *True*) *I*)\$*s*\$($\lambda_2 x$. *True*)
FOREACH $\equiv$ *FOREACHi*\$(λ_2- -. *True*)
⟨*proof*⟩

term *FOREACHoci R I*
lemma *id-FOREACHoci*[*id-rules*]: *PR-CONST* (*FOREACHoci R I*) $::_i$
TYPE('*c set* $\Rightarrow$ ('*d* $\Rightarrow$ *bool*) $\Rightarrow$ ('*c* $\Rightarrow$ '*d* $\Rightarrow$ '*d nres*) $\Rightarrow$ '*d* $\Rightarrow$ '*d 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-CONST (*FOREACHoci R I*) $\equiv \lambda_2 s\ c\ f\ \sigma$. *SP* (*PR-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-CONST* (*FOREACHoci R I*))\$*s*\$($\lambda_2 x$. *c x*)\$*f*\$$\sigma \equiv$
Refine-Basic.bind\$(*EVAL*\$*s*)\$($\lambda_2 s$. *Refine-Basic.bind*\$(*EVAL*\$$\sigma$)\$($\lambda_2 \sigma$.
SP (*PR-CONST* (*monadic-FOREACH R I*))\$*s*\$($\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* ≡ *heap.fixp-fun* (λ*D* (*l*,*s*). *case l of*
  [] ⇒ *return s*
| *x*#*ls* ⇒ *do* {
    b←*c s*;
    if b then do { *s*′←*f x s*; *D* (*ls*,*s*′)} *else return s*
  }
) (*l*,*s*)

declare *imp-nfoldli-def*[*code del*]

lemma *imp-nfoldli-simps*[*simp*,*code*]:
  imp-nfoldli [] *c f s* = *return s*
  imp-nfoldli (*x*#*ls*) *c f s* = (*do* {
    b ← *c s*;
    if b then do {
      s′←*f x s*;
      imp-nfoldli ls c f s′
    } *else return s*
  })
  ⟨*proof*⟩

lemma *monadic-nfoldli-refine-aux*:
  assumes *c-ref*: ⋀*s s*′. *hn-refine*
    (Γ ∗ *hn-ctxt Rs s*′ *s*)
    (*c s*)
    (Γ ∗ *hn-ctxt Rs s*′ *s*)
    bool-assn
    (*c*′ *s*′)
  assumes *f-ref*: ⋀*x x*′ *s s*′. *hn-refine*
    (Γ ∗ *hn-ctxt Rl x*′ *x* ∗ *hn-ctxt Rs s*′ *s*)
    (*f x s*)
    (Γ ∗ *hn-invalid Rl x*′ *x* ∗ *hn-invalid Rs s*′ *s*) *Rs*
    (*f*′ *x*′ *s*′)

  shows *hn-refine*
    (Γ ∗ *hn-ctxt* (*list-assn Rl*) *l*′ *l* ∗ *hn-ctxt Rs s*′ *s*)
    (*imp-nfoldli l c f s*)
    (Γ ∗ *hn-invalid* (*list-assn Rl*) *l*′ *l* ∗ *hn-invalid Rs s*′ *s*) *Rs*
    (*monadic-nfoldli l*′ *c*′ *f*′ *s*′)
  applyF (*induct p*≡*Rl l*′ *l*
    arbitrary: *s s*′
    rule: *list-assn.induct*)

applyF *simp*
⟨*proof*⟩
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
⟨*proof*⟩

lemma *hn-monadic-nfoldli*:
assumes *FR*: $P \Longrightarrow_t \Gamma * \textit{hn-ctxt}\ (\textit{list-assn}\ Rl)\ l'\ l * \textit{hn-ctxt}\ Rs\ s'\ s$
assumes *c-ref*: $\bigwedge s\ s'.$ *hn-refine*
($\Gamma * \textit{hn-ctxt}\ Rs\ s'\ s$)
($c\ s$)
($\Gamma * \textit{hn-ctxt}\ Rs\ s'\ s$)
bool-assn
($c'\$s'$)
assumes *f-ref*: $\bigwedge x\ x'\ s\ s'.$ *hn-refine*
($\Gamma * \textit{hn-ctxt}\ Rl\ x'\ x * \textit{hn-ctxt}\ Rs\ s'\ s$)
($f\ x\ s$)
($\Gamma * \textit{hn-invalid}\ Rl\ x'\ x * \textit{hn-invalid}\ Rs\ s'\ s$) *Rs*
($f'\$x'\$s'$)
shows *hn-refine*
P
(*imp-nfoldli l c f s*)
($\Gamma * \textit{hn-invalid}\ (\textit{list-assn}\ Rl)\ l'\ l * \textit{hn-invalid}\ Rs\ s'\ s$)
Rs
(*monadic-nfoldli*l'\$$c'$\$$f'$\$$s'$)

⟨*proof*⟩

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* σ}

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* σ) σ)
⟨*proof*⟩

lemma *imp-nfoldli-mono*[*partial-function-mono*]:
assumes [*partial-function-mono*]: $\bigwedge x\ \sigma.$ *mono-Heap* ($\lambda fa.$ *f fa x* σ)
shows *mono-Heap* ($\lambda x.$ *imp-nfoldli l c* (*f x*) σ)
⟨*proof*⟩

lemma *imp-foreach-mono*[*partial-function-mono*]:
assumes [*partial-function-mono*]: $\bigwedge x\ \sigma.$ *mono-Heap* ($\lambda fa.$ *f fa x* σ)

shows *mono-Heap* (λx. *imp-foreach tsl l c* (*f x*) σ)
⟨*proof*⟩

lemmas [*sepref-opt-simps*] = *imp-foreach-def*

definition
IS-TO-SORTED-LIST Ω *Rs Rk tsl* ≡ (*tsl*,*it-to-sorted-list* Ω) ∈ (*Rs*)^k →$_a$ *list-assn Rk*

lemma *IS-TO-SORTED-LISTI*:
assumes (*tsl*,*PR-CONST* (*it-to-sorted-list* Ω)) ∈ (*Rs*)^k →$_a$ *list-assn Rk*
shows *IS-TO-SORTED-LIST* Ω *Rs Rk tsl*
⟨*proof*⟩

lemma *hn-monadic-FOREACH*[*sepref-comb-rules*]:
assumes *INDEP Rk INDEP Rs INDEP Rσ*
assumes *FR*: *P* ⟹$_t$ Γ ∗ *hn-ctxt Rs s' s* ∗ *hn-ctxt Rσ σ' σ*
assumes *STL*: *GEN-ALGO tsl* (*IS-TO-SORTED-LIST ordR Rs Rk*)
assumes *c-ref*: ⋀σ σ'. *hn-refine*
(Γ ∗ *hn-ctxt Rs s' s* ∗ *hn-ctxt Rσ σ' σ*)
(*c σ*)
(Γ*c σ' σ*)
bool-assn
(*c' σ'*)
assumes *C-FR*:
⋀σ' σ. *TERM monadic-FOREACH* ⟹
Γ*c σ' σ* ⟹$_t$ Γ ∗ *hn-ctxt Rs s' s* ∗ *hn-ctxt Rσ σ' σ*

assumes *f-ref*: ⋀*x' x σ' σ*. *hn-refine*
(Γ ∗ *hn-ctxt Rs s' s* ∗ *hn-ctxt Rk x' x* ∗ *hn-ctxt Rσ σ' σ*)
(*f x σ*)
(Γ*f x' x σ' σ*) *Rσ*
(*f' x' σ'*)
assumes *F-FR*: ⋀*x' x σ' σ*. *TERM monadic-FOREACH* ⟹ Γ*f x' x σ' σ* ⟹$_t$
Γ ∗ *hn-ctxt Rs s' s* ∗ *hn-ctxt Pfx x' x* ∗ *hn-ctxt Pfσ σ' σ*

shows *hn-refine*
P
(*imp-foreach tsl s c f σ*)
(Γ ∗ *hn-ctxt Rs s' s* ∗ *hn-invalid Rσ σ' σ*)
Rσ
((*PR-CONST* (*monadic-FOREACH ordR I*))
$*s'*$(λ$_2$σ'. *c' σ'*)$(λ$_2$x' σ'. *f' x' σ'*)$*σ'*
)
⟨*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* ⊆ *S*
shows *monadic-nfoldli l c* (λ*x s*. *ASSERT* (*x*∈*S*)≫*f x s*) *s* = *monadic-nfoldli l c f s*
⟨*proof*⟩

lemmas *monadic-nfoldli-assert* = *monadic-nfoldli-assert-aux*[*OF order-refl*]

lemma *nfoldli-arities*[*sepref-monadify-arity*]:
nfoldli ≡ λ$_2$*s c f σ*. *SP* (*nfoldli*)$*s*$(λ$_2$*x*. *c*$*x*)$(λ$_2$*x σ*. *f*$*x*$*σ*)$*σ*
⟨*proof*⟩

lemma *nfoldli-comb*[*sepref-monadify-comb*]:
⋀*s c f σ*. (*nfoldli*)$*s*$(λ$_2$*x*. *c x*)$*f*$*σ* ≡
Refine-Basic.bind$(*EVAL*$*s*)$(λ$_2$*s*. *Refine-Basic.bind*$(*EVAL*$*σ*)$(λ$_2$*σ*.
SP (*monadic-nfoldli*)$*s*$(λ$_2$*x*. (*EVAL*$(*c x*)))$*f*$*σ*
))
⟨*proof*⟩

lemma *monadic-nfoldli-refine-aux′*:
assumes *SS*: *set l′* ⊆ *S*
assumes *c-ref*: ⋀*s s′*. *hn-refine*
(Γ ∗ *hn-ctxt Rs s′ s*)
(*c s*)
(Γ ∗ *hn-ctxt Rs s′ s*)
bool-assn
(*c′ s′*)
assumes *f-ref*: ⋀*x x′ s s′*. ⟦*x′* ∈ *S*⟧ ⟹ *hn-refine*
(Γ ∗ *hn-ctxt Rl x′ x* ∗ *hn-ctxt Rs s′ s*)
(*f x s*)
(Γ ∗ *hn-ctxt Rl′ x′ x* ∗ *hn-invalid Rs s′ s*) *Rs*
(*f′ x′ s′*)

assumes *merge*[*sepref-frame-merge-rules*]: ⋀*x x′*. *hn-ctxt Rl′ x′ x* ∨$_A$ *hn-ctxt Rl x′ x* ⟹$_t$ *hn-ctxt Rl″ x′ x*
notes [*sepref-frame-merge-rules*] = *merge-sat2*[*OF merge*]

shows *hn-refine*
(Γ ∗ *hn-ctxt* (*list-assn Rl*) *l′ l* ∗ *hn-ctxt Rs s′ s*)
(*imp-nfoldli l c f s*)
(Γ ∗ *hn-ctxt* (*list-assn Rl″*) *l′ l* ∗ *hn-invalid Rs s′ s*) *Rs*
(*monadic-nfoldli l′ c′ f′ s′*)

apply1 (*subst monadic-nfoldli-assert-aux*[*OF SS*,*symmetric*])

applyF (*induct p*≡*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 no-tac*)) @{*context*} *1*›)

apply1 (*simp add*: *assn-assoc*)
applyS (*tactic* ‹*Sepref-Frame.merge-tac* (*K* (*K no-tac*)) @{*context*} *1*›)

⟨*proof*⟩
solved
⟨*proof*⟩

lemma *hn-monadic-nfoldli-rl'*[*sepref-comb-rules*]:
assumes *INDEP Rk INDEP Rσ*
assumes *FR*: $P \Longrightarrow_t \Gamma * \textit{hn-ctxt}\ (\textit{list-assn}\ Rk)\ s'\ s * \textit{hn-ctxt}\ R\sigma\ \sigma'\ \sigma$
assumes *c-ref*: $\bigwedge\sigma\ \sigma'.$ *hn-refine*
($\Gamma * \textit{hn-ctxt}\ R\sigma\ \sigma'\ \sigma$)
($c\ \sigma$)
($\Gamma c\ \sigma'\ \sigma$)
bool-assn
($c'\ \sigma'$)
assumes *C-FR*:
$\bigwedge\sigma'\ \sigma.$ *TERM monadic-nfoldli* $\Longrightarrow$
$\Gamma c\ \sigma'\ \sigma \Longrightarrow_t \Gamma * \textit{hn-ctxt}\ R\sigma\ \sigma'\ \sigma$

assumes *f-ref*: $\bigwedge x'\ x\ \sigma'\ \sigma.\ [\![x' \in \textit{set}\ s']\!] \Longrightarrow$ *hn-refine*

(Γ ∗ *hn-ctxt Rk x′ x* ∗ *hn-ctxt Rσ σ′ σ*)
(*f x σ*)
(Γ*f x′ x σ′ σ*) *Rσ*
(*f′ x′ σ′*)
assumes *F-FR*: ⋀*x′ x σ′ σ*. *TERM monadic-nfoldli* ⟹ Γ*f x′ x σ′ σ* ⟹$_t$
Γ ∗ *hn-ctxt Rk′ x′ x* ∗ *hn-ctxt Pfσ σ′ σ*

assumes *MERGE*: ⋀*x x′*. *hn-ctxt Rk′ x′ x* ∨$_A$ *hn-ctxt Rk x′ x* ⟹$_t$ *hn-ctxt Rk″ x′ x*

shows *hn-refine*
P
(*imp-nfoldli s c f σ*)
(Γ ∗ *hn-ctxt* (*list-assn Rk″*) *s′ s* ∗ *hn-invalid Rσ σ′ σ*)
Rσ
((*monadic-nfoldli*)
$*s′*$(λ$_2$*σ′*. *c′ σ′*)$(λ$_2$*x′ σ′*. *f′ x′ σ′*)$*σ′*
)
⟨*proof*⟩
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
⟨*proof*⟩
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* ‹*Sepref-Frame.frame-tac* (*K* (*K no-tac*)) @{*context*} *1*›)
solved

⟨*proof*⟩
solved

applyS (*tactic* ‹*Sepref-Frame.frame-tac* (*K* (*K no-tac*)) @{*context*} *1*›)
applyS (*tactic* ‹*Sepref-Frame.frame-tac* (*K* (*K no-tac*)) @{*context*} *1*›)
applyS (*tactic* ‹*Sepref-Frame.frame-tac* (*K* (*K no-tac*)) @{*context*} *1*›)
solved
⟨*proof*⟩

lemma *nfoldli-assert*:
assumes *set l* ⊆ *S*

shows *nfoldli l c* (λ *x s. ASSERT* (*x* $\in$ *S*) $\gg$ *f x s*) *s* = *nfoldli l c f s*
⟨*proof*⟩

lemmas *nfoldli-assert'* = *nfoldli-assert*[*OF order.refl*]

lemma *fold-eq-nfoldli*:
RETURN (*fold f l s*) = *nfoldli l* (λ-. *True*) (λ*x s. RETURN* (*f x s*)) *s*
⟨*proof*⟩

lemma *fold-eq-nfoldli-assert*:
RETURN (*fold f l s*) = *nfoldli l* (λ-. *True*) (λ*x s. ASSERT* (*x*∈*set l*) $\gg$ *RETURN* (*f x s*)) *s*
⟨*proof*⟩

lemma *fold-arity*[*sepref-monadify-arity*]: *fold* ≡ λ_2*f l s. SP fold*$(λ$_2$*x s. f*$x$$s*)$$l$$s*
⟨*proof*⟩

lemma *monadify-plain-fold*[*sepref-monadify-comb*]:
EVAL$(*fold*$(λ$_2$*x s. f x s*)$$l$$s*) ≡ ($\gg$=)$(*EVAL*$*l*)$(λ$_2$*l.* ($\gg$=)$(*EVAL*$*s*)$(λ$_2$*s. nfoldli*$$l$(λ$_2$-. *True*)$(λ$_2$*x s. EVAL*$(*f x s*))$$s*))
⟨*proof*⟩

lemma *monadify-plain-fold-old-rl*:
EVAL$(*fold*$(λ$_2$*x s. f x s*)$$l$$s*) ≡ ($\gg$=)$(*EVAL*$*l*)$(λ$_2$*l.* ($\gg$=)$(*EVAL*$*s*)$(λ$_2$*s. nfoldli*$$l$(λ$_2$-. *True*)$(λ$_2$*x s. PR-CONST* (*op-ASSERT-bind* (*x*∈*set l*))$(*EVAL*$(*f x s*)))$$s*))
⟨*proof*⟩

foldli

lemma *foldli-eq-nfoldli*:
RETURN (*foldli l c f s*) = *nfoldli l c* (λ*x s. RETURN* (*f x s*)) *s*
⟨*proof*⟩

lemma *foldli-arities*[*sepref-monadify-arity*]:
foldli ≡ λ_2*s c f* σ*. SP* (*foldli*)$$s$(λ$_2$*x. c*$$x$)$(λ$_2$*x* σ*. f*$$x$$\sigma$)$$\sigma*
⟨*proof*⟩

lemma *monadify-plain-foldli*[*sepref-monadify-comb*]:
EVAL$(*foldli*$$l$$c$(λ$_2$*x s. f x s*)$$s*) ≡
($\gg$=)$(*EVAL*$*l*)$
(λ$_2$*l.* ($\gg$=)$(*EVAL*$*s*)$
(λ$_2$*s. nfoldli*$$l$$c$(λ$_2$*x s.* (*EVAL*$(*f x s*)))$$s*))
⟨*proof*⟩

Deforestation

lemma *nfoldli-filter-deforestation*:
nfoldli (*filter P xs*) *c f s* = *nfoldli xs c* (λ*x s. if P x then f x s else RETURN s*) *s*

⟨*proof*⟩

lemma *extend-list-of-filtered-set*:
assumes [*simp*, *intro*!]: *finite S*
and *A*: *distinct xs' set xs' = {x ∈ S. P x}*
obtains *xs* **where** *xs' = filter P xs distinct xs set xs = S*
⟨*proof*⟩

lemma *FOREACHc-filter-deforestation*:
assumes *FIN*[*simp*, *intro*!]: *finite S*
shows (*FOREACHc* {x∈S. *P x*} *c f s*)
= FOREACHc S c (λ*x s. if P x then f x s else RETURN s*) *s*
⟨*proof*⟩
applyS *simp*
applyS (*simp add*: *nfoldli-filter-deforestation*)
⟨*proof*⟩

lemma *FOREACHc-filter-deforestation2*:
assumes [*simp*]: *distinct xs*
shows (*FOREACHc* (*set* (*filter P xs*)) *c f s*)
= FOREACHc (*set xs*) *c* (λ*x s. if P x then f x s else RETURN s*) *s*
⟨*proof*⟩

2.2.2 For Loops

partial-function (*heap*) *imp-for* :: *nat* ⇒ *nat* ⇒ (*'a* ⇒ *bool Heap*) ⇒ (*nat* ⇒ *'a* ⇒ *'a Heap*) ⇒ *'a* ⇒ *'a Heap* **where**
imp-for i u c f s = (if i ≥ u then return s else do {ctn <− c s; if ctn then f i s ⤜ *imp-for* (*i + 1*) *u c f else return s*})

declare *imp-for.simps*[*code*]

lemma [*simp*]:
i ≥ u ⟹ *imp-for i u c f s = return s*
i < u ⟹ *imp-for i u c f s = do {ctn <− c s; if ctn then f i s* ⤜ *imp-for* (*i + 1*) *u c f else return s*}
⟨*proof*⟩

lemma *imp-nfoldli-deforest*[*sepref-opt-simps*]:
imp-nfoldli [*l..<u*] *c = imp-for l u c*
⟨*proof*⟩

partial-function (*heap*) *imp-for'* :: *nat* ⇒ *nat* ⇒ (*nat* ⇒ *'a* ⇒ *'a Heap*) ⇒ *'a* ⇒ *'a Heap* **where**
imp-for' i u f s = (if i ≥ u then return s else f i s ⤜ *imp-for'* (*i + 1*) *u f*)

declare *imp-for'.simps*[*code*]

lemma [*simp*]:
  $i \geq u \Longrightarrow$ *imp-for'* $i\ u\ f\ s = return\ s$
  $i < u \Longrightarrow$ *imp-for'* $i\ u\ f\ s = f\ i\ s \gg\!= imp\text{-}for'\ (i + 1)\ u\ f$
⟨*proof*⟩

lemma *imp-for-imp-for'*[*sepref-opt-simps*]:
  imp-for $i\ u\ (\lambda\text{-}.\ return\ True) = imp\text{-}for'\ i\ u$
⟨*proof*⟩

partial-function (*heap*) *imp-for-down* :: *nat* ⇒ *nat* ⇒ (*'a* ⇒ *bool Heap*) ⇒ (*nat* ⇒ *'a* ⇒ *'a Heap*) ⇒ *'a* ⇒ *'a Heap* **where**
  imp-for-down l i c f s = do {
    let i = i − 1;
    ctn ← c s;
    if ctn then do {
      s ← f i s;
      if i>l then imp-for-down l i c f s else return s
    } else return s
  }

declare *imp-for-down.simps*[*code*]

lemma *imp-nfoldli-deforest-down*[*sepref-opt-simps*]:
  imp-nfoldli (rev [l..<u]) c =
    (λf s. if u≤l then return s else imp-for-down l u c f s)
⟨*proof*⟩

context begin

private fun *imp-for-down-induction-scheme* :: *nat* ⇒ *nat* ⇒ *unit* **where**
  imp-for-down-induction-scheme l i = (
    let i=i−1 in
    if i>l then
      imp-for-down-induction-scheme l i
    else ()
)

partial-function (*heap*) *imp-for-down'* :: *nat* ⇒ *nat* ⇒ (*nat* ⇒ *'a* ⇒ *'a Heap*) ⇒ *'a* ⇒ *'a Heap* **where**
  imp-for-down' l i f s = do {
    let i = i − 1;
    s ← 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 (λ-. *return True*) = *imp-for-down' l u*
  ⟨*proof*⟩

end

lemma *imp-for'-rule*:
assumes *LESS*: $l \leq u$
assumes *PRE*: $P \Longrightarrow_A I\ l\ s$
assumes *STEP*: $\bigwedge i\ s.\ [\![\ l \leq i;\ i < u\ ]\!] \Longrightarrow <I\ i\ s>\ f\ i\ s\ <I\ (i+1)>$
shows $<P>$ *imp-for'* $l\ u\ f\ s\ <I\ u>$
⟨*proof*⟩

This lemma is used to manually convert a fold to a loop over indices.

lemma *fold-idx-conv*: *fold f l s* = *fold* $(\lambda i.\ f\ (l!i))$ $[0..<length\ l]$ *s*
⟨*proof*⟩

end
theory *Default-Insts*
imports *Main*
begin

instantiation *nat* :: *default* **begin**
definition *default* = (*0*::*nat*)
instance ⟨*proof*⟩
end

instantiation *int* :: *default* **begin**
definition *default* = (*0*::*int*)
instance ⟨*proof*⟩
end

instantiation *bool* :: *default* **begin**
definition *default* = *False*
instance ⟨*proof*⟩
end

instantiation *prod* :: (*default*,*default*) *default* **begin**
definition *default* = (*default*,*default*)
instance ⟨*proof*⟩
end

instantiation *list* :: (*type*)*default* **begin**
definition *default* = []
instance ⟨*proof*⟩
end

instantiation *option* :: (*type*)*default* **begin**
definition *default* = *None*
instance ⟨*proof*⟩

end

instantiation *sum* :: (*default*,*type*)*default* **begin**
definition *default* = *Inl default*
instance ⟨*proof*⟩
end
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* ⇒ '*b*) ⇒ '*a* ⇒ '*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*→*B*) *f fi* ∗ *hn-val A x xi*)
(*return* (*pho-apply*$*fi*$*xi*))
(*hn-val* (*A*→*B*) *f fi* ∗ *hn-val A x xi*)
(*pure B*)
(*RETURN*$(*pho-apply*$*f*$*x*))
⟨*proof*⟩

end
theory *Sepref*
imports

Sepref-Tool
Sepref-HOL-Bindings

Sepref-Foreach
Sepref-Intf-Util
../Separation-Logic-Imperative-HOL/Examples/Default-Insts
Sepref-Improper
begin

end

Chapter 3

The Imperative Isabelle Collection Framework

The Imperative Isabelle Collection Framework provides efficient imperative implementations of collection data structures.

3.1 Set Interface

theory *IICF-Set*
imports *../../Sepref*
begin

3.1.1 Operations

definition [*simp*]: *op-set-is-empty s* ≡ *s*={}
lemma *op-set-is-empty-param*[*param*]: (*op-set-is-empty*,*op-set-is-empty*)∈⟨*A*⟩*set-rel* → *bool-rel* ⟨*proof*⟩

context
  notes [*simp*] = *IS-LEFT-UNIQUE-def*
begin

sepref-decl-op *set-empty*: {} :: ⟨*A*⟩*set-rel* ⟨*proof*⟩
sepref-decl-op (*no-def*) *set-is-empty*: *op-set-is-empty* :: ⟨*A*⟩*set-rel* → *bool-rel* ⟨*proof*⟩
sepref-decl-op *set-member*: (∈) :: *A* → ⟨*A*⟩*set-rel* → *bool-rel* **where** *IS-LEFT-UNIQUE A IS-RIGHT-UNIQUE A* ⟨*proof*⟩
sepref-decl-op *set-insert*: *Set.insert* :: *A* → ⟨*A*⟩*set-rel* → ⟨*A*⟩*set-rel* **where** *IS-RIGHT-UNIQUE A* ⟨*proof*⟩
sepref-decl-op *set-delete*: λ*x s*. *s* − {*x*} :: *A* → ⟨*A*⟩*set-rel* → ⟨*A*⟩*set-rel*
  where *IS-LEFT-UNIQUE A IS-RIGHT-UNIQUE A* ⟨*proof*⟩
sepref-decl-op *set-union*: (∪) :: ⟨*A*⟩*set-rel* → ⟨*A*⟩*set-rel* → ⟨*A*⟩*set-rel* ⟨*proof*⟩
sepref-decl-op *set-inter*: (∩) :: ⟨*A*⟩*set-rel* → ⟨*A*⟩*set-rel* → ⟨*A*⟩*set-rel* **where** *IS-LEFT-UNIQUE A IS-RIGHT-UNIQUE A* ⟨*proof*⟩

sepref-decl-op *set-diff*: $(-)$ ::- *set* $\Rightarrow$ - :: $\langle A\rangle$*set-rel* $\to$ $\langle A\rangle$*set-rel* $\to$ $\langle A\rangle$*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* $\to$ $\langle A\rangle$*set-rel* $\to$ *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* $\to$ $\langle A\rangle$*set-rel* $\to$ *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* $\to$ *K* $\langle$*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* :: *'ax set*

  lemma *fold-custom-empty*:
    $\{\}$ = *op-custom-empty*
    op-set-empty = *op-custom-empty*
    mop-set-empty = *RETURN op-custom-empty*
    $\langle$*proof*$\rangle$
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* ≡ *hr-comp* (*list-assn A*) (*br set* (λ-. *True*))
lemmas [*fcomp-norm-unfold*] = *lso-assn-def*[*symmetric*]
lemma *lso-is-pure*[*safe-constraint-rules*]: *is-pure A* ⟹ *is-pure* (*lso-assn A*)
⟨*proof*⟩

lemma *lso-empty-aref*: (*uncurry0* (*RETURN* []), *uncurry0* (*RETURN op-set-empty*))

∈ *unit-rel* →*f* ⟨*br set* (λ-. *True*)⟩*nres-rel*
⟨*proof*⟩

lemma *lso-ins-aref*: (*uncurry* (*RETURN oo* ((#))), *uncurry* (*RETURN oo op-set-insert*))
∈ *Id* ×*r br set* (λ-. *True*) →*f* ⟨*br set* (λ-. *True*)⟩*nres-rel*
⟨*proof*⟩

sepref-decl-impl (*no-register*) *lso-empty*: *hn-Nil*[*to-hfref*] **uses** *lso-empty-aref* ⟨*proof*⟩
definition [*simp*]: *op-lso-empty* ≡ *op-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-Cons*[*to-hfref*] **uses** *lso-ins-aref* ⟨*proof*⟩

thm *hn-Cons*[*FCOMP lso-ins-aref*]

definition [*simp*]: *op-lso-bex P S* ≡ ∃ *x*∈*S*. *P x*
lemma *fold-lso-bex*: *Bex* ≡ λ*s P*. *op-lso-bex P s* ⟨*proof*⟩

definition [*simp*]: *mop-lso-bex P S* ≡ *ASSERT* (∀ *x*∈*S*. ∃ *y*. *P x* = *RETURN y*) ≫ *RETURN* (∃ *x*∈*S*. *P x* = *RETURN True*)

lemma *op-mop-lso-bex*: *RETURN* (*op-lso-bex P S*) = *mop-lso-bex* (*RETURN o P*) *S* ⟨*proof*⟩

sepref-register *op-lso-bex*
lemma *lso-bex-arity*[*sepref-monadify-arity*]:
op-lso-bex ≡ λ$2*P s*. *SP op-lso-bex*$(λ$2*x*. *P*$*x*)$*s* ⟨*proof*⟩
lemma *op-lso-bex-monadify*[*sepref-monadify-comb*]:

EVAL$(*op-lso-bex*$(λ$_2$*x*. *P x*)$*s*) ≡ (≫=) $(*EVAL*$*s*)$(λ$_2$*s*. *mop-lso-bex*$(λ$_2$*x*. *EVAL* $ *P x*)$*s*) ⟨*proof*⟩

definition *lso-abex P l* ≡ *nfoldli l* (*Not*) (λ*x* -. *P x*) *False*
lemma *lso-abex-to-set*: *lso-abex P l* ≤ *mop-lso-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* ⇒ *bool nres*
fixes *li* :: *'c list* **and** *l* :: *'a list*
fixes *A* :: *'a* ⇒ *'c* ⇒ *assn*
fixes *F* :: *assn*

assumes *Prl*: ⋀*x xi*. ⟦*x*∈*set l*⟧ ⟹ *hn-refine* (*F* ∗ *hn-ctxt A x xi*) (*Pi xi*) (*F* ∗ *hn-ctxt A x xi*) *bool-assn* (*P x*)
begin
sepref-register *l*
sepref-register *P*

lemma [*sepref-comb-rules*]:
assumes Γ ⟹$_t$ *F'* ∗ *F* ∗ *hn-ctxt A x xi*
assumes *x*∈*set l*
shows *hn-refine* Γ (*Pi xi*) (*F'* ∗ *F* ∗ *hn-ctxt A x xi*) *bool-assn* (*P*$*x*)
⟨*proof*⟩

schematic-goal *lso-bex-impl*:
hn-refine (*hn-ctxt* (*list-assn A*) *l li* ∗ *F*) (*?c*) (*F* ∗ *hn-ctxt* (*list-assn A*) *l li*) *bool-assn* (*lso-abex P l*)
⟨*proof*⟩
end
concrete-definition *lso-bex-impl* **uses** *lso-bex-impl-loc.lso-bex-impl*

lemma *hn-lso-bex*[*sepref-prep-comb-rule*,*sepref-comb-rules*]:
assumes *FR*: Γ ⟹$_t$ *hn-ctxt* (*lso-assn A*) *s li* ∗ *F*
assumes *Prl*: ⋀*x xi*. ⟦*x*∈*s*⟧ ⟹ *hn-refine* (*F* ∗ *hn-ctxt A x xi*) (*Pi xi*) (*F* ∗ *hn-ctxt A x xi*) *bool-assn* (*P x*)
notes [*simp del*] = *mop-lso-bex-def*
shows *hn-refine* Γ (*lso-bex-impl Pi li*) (*F* ∗ *hn-ctxt* (*lso-assn A*) *s li*) *bool-assn* (*mop-lso-bex*$(λ$_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 ⟦ *rel-mset A m n*; *A x y* ⟧ ⟹ *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*
⟨*proof*⟩

3.3.2 Parametricity Setup

definition [*to-relAPP*]: *mset-rel A* ≡ *p2rel* (*rel-mset* (*rel2p A*))

lemma *rel2p-mset*[*rel2p*]: *rel2p* (⟨*A*⟩*mset-rel*) = *rel-mset* (*rel2p A*)
⟨*proof*⟩

lemma *p2re-mset*[*p2rel*]: *p2rel* (*rel-mset A*) = ⟨*p2rel A*⟩*mset-rel*

⟨*proof*⟩

lemma *mset-rel-empty*[*simp*]:
(a,{#})∈⟨A⟩*mset-rel* ⟷ a={#}
({#},b)∈⟨A⟩*mset-rel* ⟷ b={#}
⟨*proof*⟩

lemma *param-mset-empty*[*param*]: ({#},{#}) ∈ ⟨A⟩*mset-rel*
⟨*proof*⟩

lemma *param-mset-Plus*[*param*]: ((+),(+))∈⟨A⟩*mset-rel* → ⟨A⟩*mset-rel* → ⟨A⟩*mset-rel*

⟨*proof*⟩

lemma *param-mset-add*[*param*]: (*add-mset*, *add-mset*) ∈ A → ⟨A⟩*mset-rel* → ⟨A⟩*mset-rel*
⟨*proof*⟩

lemma *param-mset-minus*[*param*]: ⟦*single-valued* A; *single-valued* (A^{-1})⟧
⟹ ((−), (−)) ∈ ⟨A⟩*mset-rel* → ⟨A⟩*mset-rel* → ⟨A⟩*mset-rel*
⟨*proof*⟩

lemma *param-count*[*param*]: ⟦*single-valued* A; *single-valued* (A^{-1})⟧ ⟹ (*count*,*count*)∈⟨A⟩*mset-rel*
→ A → *nat-rel*
⟨*proof*⟩

lemma *param-set-mset*[*param*]:
shows (*set-mset*, *set-mset*) ∈ ⟨A⟩*mset-rel* → ⟨A⟩*set-rel*
⟨*proof*⟩

definition [*simp*]: *mset-is-empty* m ≡ m = {#}

lemma *mset-is-empty-param*[*param*]: (*mset-is-empty*,*mset-is-empty*) ∈ ⟨A⟩*mset-rel*
→ *bool-rel*
⟨*proof*⟩

3.3.3 Operations

sepref-decl-op *mset-empty*: {#} :: ⟨A⟩*mset-rel* ⟨*proof*⟩

sepref-decl-op *mset-is-empty*: λm. m={#} :: ⟨A⟩*mset-rel* → *bool-rel*
⟨*proof*⟩

sepref-decl-op *mset-insert*: *add-mset* :: A → ⟨A⟩*mset-rel* → ⟨A⟩*mset-rel* ⟨*proof*⟩

sepref-decl-op *mset-delete*: $\lambda x\ m.\ m - \{\#x\#\} :: A \to \langle A\rangle$*mset-rel* $\to \langle A\rangle$*mset-rel*
where *single-valued A single-valued* (A^{-1}) ⟨*proof*⟩

sepref-decl-op *mset-plus*: (+)::- *multiset* ⇒ - :: $\langle A\rangle$*mset-rel* $\to \langle A\rangle$*mset-rel* $\to$ $\langle A\rangle$*mset-rel* ⟨*proof*⟩
sepref-decl-op *mset-minus*: (−)::- *multiset* ⇒ - :: $\langle A\rangle$*mset-rel* $\to \langle A\rangle$*mset-rel* $\to \langle A\rangle$*mset-rel*
where *single-valued A single-valued* (A^{-1}) ⟨*proof*⟩

sepref-decl-op *mset-contains*: $(\in\#) :: A \to \langle A\rangle$*mset-rel* $\to$ *bool-rel*
where *single-valued A single-valued* (A^{-1}) ⟨*proof*⟩

sepref-decl-op *mset-count*: $\lambda x\ y.$ *count y x* :: $A \to \langle A\rangle$*mset-rel* $\to$ *nat-rel*
where *single-valued A single-valued* (A^{-1}) ⟨*proof*⟩

sepref-decl-op *mset-pick*: $\lambda m.$ *SPEC* $(\lambda(x,m').\ m = \{\#x\#\} + m') ::$
$[\lambda m.\ m\neq\{\#\}]_f\ \langle A\rangle$*mset-rel* $\to A \times_r \langle A\rangle$*mset-rel*
⟨*proof*⟩
apply1 (*rule ccontr*; *clarsimp*)
applyS (*metis msed-rel-invL rel2p-def rel2p-mset union-ac*(*2*))
applyS *parametricity*
⟨*proof*⟩

3.3.4 Patterns

lemma [*def-pat-rules*]:
$\{\#\} \equiv$ *op-mset-empty*
add-mset ≡ *op-mset-insert*
(=) b{#} ≡ *op-mset-is-empty*$b
(=) ${#}$b ≡ *op-mset-is-empty*$b
(+) ab ≡ *op-mset-plus*ab
(−) ab ≡ *op-mset-minus*ab
⟨*proof*⟩

lemma [*def-pat-rules*]:
(+) b(*add-mset*x{#}) ≡ *op-mset-insert*xb
(+) $(*add-mset*$x${#})$b ≡ *op-mset-insert*xb
(−) b(*add-mset*x{#}) ≡ *op-mset-delete*xb
(<) 0(*count*ax) ≡ *op-mset-contains*xa
(∈) 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-assn*^{k} →$_a$ *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* (λ*v*.
v ∈# *b*
∧ (∀ *v'*∈*set-mset b*. *prio v* ≤ *prio v'*))

end

lemma *param-mop-prio-pop-min*[*param*]:
assumes [*param*]: (*prio'*,*prio*) ∈ *A* → *B*
assumes [*param*]: ((≤),(≤)) ∈ *B* → *B* → *bool-rel*
shows (*mop-prio-pop-min prio'*,*mop-prio-pop-min prio*) ∈ ⟨*A*⟩*mset-rel* → ⟨*A* ×$_r$ ⟨*A*⟩*mset-rel*⟩*nres-rel*

⟨*proof*⟩

lemma *param-mop-prio-peek-min*[*param*]:
assumes [*param*]: (*prio'*,*prio*) ∈ *A* → *B*
assumes [*param*]: ((≤),(≤)) ∈ *B* → *B* → *bool-rel*
shows (*mop-prio-peek-min prio'*,*mop-prio-peek-min prio*) ∈ ⟨*A*⟩*mset-rel* → ⟨*A*⟩*nres-rel*
⟨*proof*⟩

context fixes *prio* :: *'a* ⇒ *'b*::*linorder* **and** *A* :: (*'a*×*'a*) *set* **begin**
sepref-decl-op (*no-def*,*no-mop*) *prio-pop-min*:
PR-CONST (*mop-prio-pop-min prio*) :: ⟨*A*⟩*mset-rel* →$_f$ ⟨*A* ×$_r$ ⟨*A*⟩*mset-rel*⟩*nres-rel*
where *IS-BELOW-ID A*
⟨*proof*⟩

sepref-decl-op (*no-def*,*no-mop*) *prio-peek-min*:
PR-CONST (*mop-prio-peek-min prio*) :: ⟨*A*⟩*mset-rel* →$_f$ ⟨*A*⟩*nres-rel*
where *IS-BELOW-ID A*
⟨*proof*⟩
end

3.4.2 Patterns

lemma [*def-pat-rules*]:
mop-prio-pop-min$*prio* ≡ *UNPROTECT* (*mop-prio-pop-min prio*)
mop-prio-peek-min$*prio* ≡ *UNPROTECT* (*mop-prio-peek-min prio*)
⟨*proof*⟩

end

3.5 Multisets by Lists

theory *IICF-List-Mset*
imports *../Intf/IICF-Multiset*
begin

3.5.1 Abstract Operations

definition *list-mset-rel* ≡ *br mset* (λ-. *True*)

lemma *lms-empty-aref*: ([],*op-mset-empty*) ∈ *list-mset-rel*
⟨*proof*⟩

lemma *lms-is-empty-aref*: (*is-Nil*,*op-mset-is-empty*) ∈ *list-mset-rel* → *bool-rel*

⟨*proof*⟩

lemma *lms-insert-aref*: ((#), *op-mset-insert*) ∈ *Id* → *list-mset-rel* → *list-mset-rel*
⟨*proof*⟩

lemma *lms-union-aref*: ((@), *op-mset-plus*) ∈ *list-mset-rel* → *list-mset-rel* → *list-mset-rel*
⟨*proof*⟩

lemma *lms-pick-aref*: (λ*x*#*l* ⇒ *RETURN* (*x*,*l*), *mop-mset-pick*) ∈ *list-mset-rel* → ⟨*Id* ×$_r$ *list-mset-rel*⟩*nres-rel*
⟨*proof*⟩
apply1 (*refine-vcg nres-relI fun-relI*)
apply1 (*clarsimp simp*: *in-br-conv neq-Nil-conv*)
apply1 (*refine-vcg RETURN-SPEC-refine*)
applyS (*clarsimp simp*: *in-br-conv algebra-simps*)
⟨*proof*⟩

definition *list-contains x l* ≡ *list-ex* ((=) *x*) *l*
lemma *lms-contains-aref*: (*list-contains*, *op-mset-contains*) ∈ *Id* → *list-mset-rel* → *bool-rel*
⟨*proof*⟩

fun *list-remove1* :: *'a* ⇒ *'a list* ⇒ *'a list* **where**
list-remove1 x [] = []
| *list-remove1 x* (*y*#*ys*) = (*if x*=*y then ys else y*#*list-remove1 x ys*)

lemma *mset-list-remove1*[*simp*]: *mset* (*list-remove1 x l*) = *mset l* − {#*x*#}
⟨*proof*⟩
applyS *simp*
⟨*proof*⟩

lemma *lms-remove-aref*: (*list-remove1*, *op-mset-delete*) ∈ *Id* → *list-mset-rel* → *list-mset-rel*
⟨*proof*⟩

fun *list-count* :: *'a* ⇒ *'a list* ⇒ *nat* **where**
list-count - [] = *0*
| *list-count x* (*y*#*ys*) = (*if x*=*y then 1* + *list-count x ys else list-count x ys*)

lemma *mset-list-count*[*simp*]: *list-count x ys* = *count* (*mset ys*) *x*
⟨*proof*⟩

lemma *lms-count-aref*: (*list-count*, *op-mset-count*) ∈ *Id* → *list-mset-rel* → *nat-rel*

⟨*proof*⟩

definition *list-remove-all* :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**

list-remove-all xs ys ≡ fold list-remove1 ys xs
lemma *list-remove-all-mset*[*simp*]: *mset* (*list-remove-all xs ys*) = *mset xs* − *mset ys*
⟨*proof*⟩

lemma *lms-minus-aref*: (*list-remove-all*,*op-mset-minus*) ∈ *list-mset-rel* → *list-mset-rel* → *list-mset-rel*
⟨*proof*⟩

3.5.2 Declaration of Implementations

definition *list-mset-assn A* ≡ *pure* (*list-mset-rel O* ⟨*the-pure A*⟩*mset-rel*)
declare *list-mset-assn-def*[*symmetric*,*fcomp-norm-unfold*]
lemma [*safe-constraint-rules*]: *is-pure* (*list-mset-assn A*) ⟨*proof*⟩

sepref-decl-impl (*no-register*) *lms-empty*: *lms-empty-aref*[*sepref-param*] ⟨*proof*⟩

definition [*simp*]: *op-list-mset-empty* ≡ *op-mset-empty*
lemma *lms-fold-custom-empty*:
{#} = *op-list-mset-empty*
op-mset-empty = *op-list-mset-empty*
⟨*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′*:
(*λx*#*l* ⇒ *return* (*x*,*l*), *mop-mset-pick*) ∈ (*pure list-mset-rel*)^k →$_a$ *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 *lmso-assn A* ≡ *hr-comp* (*list-assn A*) (*br mset* (*λ*-. *True*))
lemmas [*fcomp-norm-unfold*] = *lmso-assn-def*[*symmetric*]

lemma *lmso-is-pure*[*safe-constraint-rules*]: *is-pure A* $\Longrightarrow$ *is-pure* (*lmso-assn A*)
⟨*proof*⟩

lemma *lmso-empty-aref*: (*uncurry0* (*RETURN* []), *uncurry0* (*RETURN op-mset-empty*)) ∈ *unit-rel* $\rightarrow_f$ ⟨*br mset* (λ-. *True*)⟩*nres-rel*
⟨*proof*⟩

lemma *lmso-is-empty-aref*: (*RETURN o List.null*, *RETURN o op-mset-is-empty*) ∈ *br mset* (λ-. *True*) $\rightarrow_f$ ⟨*bool-rel*⟩*nres-rel*
⟨*proof*⟩

lemma *lmso-insert-aref*: (*uncurry* (*RETURN oo* (#)), *uncurry* (*RETURN oo op-mset-insert*)) ∈ (*Id* $\times_r$ *br mset* (λ-. *True*)) $\rightarrow_f$ ⟨*br mset* (λ-. *True*)⟩*nres-rel*
⟨*proof*⟩

definition [*simp*]: *hd-tl l* ≡ (*hd l*, *tl l*)

lemma *hd-tl-opt*[*sepref-opt-simps*]: *hd-tl l* = (*case l of* (*x*#*xs*) ⇒ (*x*,*xs*) | - ⇒ *CODE-ABORT* (λ-. (*hd l*, *tl l*)))
⟨*proof*⟩

lemma *lmso-pick-aref*: (*RETURN o hd-tl*,*op-mset-pick*) ∈ [λ*m*. *m*≠{#}]$_f$ *br mset* (λ-. *True*) → ⟨*Id* $\times_r$ *br mset* (λ-. *True*)⟩*nres-rel*
⟨*proof*⟩

lemma *hd-tl-hnr*: (*return o hd-tl*,*RETURN o hd-tl*) ∈ [λ*l*. ¬*is-Nil l*]$_a$ (*list-assn A*)^d → *prod-assn A* (*list-assn A*)
⟨*proof*⟩

sepref-decl-impl (*no-register*) *lmso-empty*: *hn-Nil*[*to-hfref*] **uses** *lmso-empty-aref*
⟨*proof*⟩
definition [*simp*]: *op-lmso-empty* ≡ *op-mset-empty*
sepref-register *op-lmso-empty*
lemma *lmso-fold-custom-empty*:
{#} = *op-lmso-empty*
op-mset-empty = *op-lmso-empty*
mop-mset-empty = *RETURN op-lmso-empty*
⟨*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 exloit *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* := *l*!*j*, *j*:=*l*!*i*]
lemma *swap-nth*[*simp*]: ⟦*i* < *length l*; *j*<*length l*; *k*<*length l*⟧ ⟹
swap l i j!*k* = (
if k=*i then l*!*j*
else if k=*j then l*!*i*
else l!*k*
)
⟨*proof*⟩

lemma *swap-set*[*simp*]: ⟦ *i* < *length l*; *j*<*length l* ⟧ ⟹ *set* (*swap l i j*) = *set l*
⟨*proof*⟩

lemma *swap-multiset*[*simp*]: ⟦ *i* < *length l*; *j*<*length l* ⟧ ⟹ *mset* (*swap l i j*) = *mset l*
⟨*proof*⟩

lemma *swap-length*[*simp*]: *length* (*swap l i j*) = *length l*
⟨*proof*⟩

lemma *swap-same*[*simp*]: *swap l i i* = *l*
⟨*proof*⟩

lemma *distinct-swap*[*simp*]:
⟦*i*<*length l*; *j*<*length l*⟧ ⟹ *distinct* (*swap l i j*) = *distinct l*
⟨*proof*⟩

lemma *map-swap*: ⟦*i*<*length l*; *j*<*length l*⟧
⟹ *map f* (*swap l i j*) = *swap* (*map f l*) *i j*
⟨*proof*⟩

lemma *swap-param*[*param*]: ⟦ *i*<*length l*; *j*<*length l*; (*l'*,*l*)∈⟨*A*⟩*list-rel*; (*i'*,*i*)∈*nat-rel*; (*j'*,*j*)∈*nat-rel*⟧
⟹ (*swap l' i' j'*, *swap l i j*)∈⟨*A*⟩*list-rel*
⟨*proof*⟩

lemma *swap-param-fref*: (*uncurry2 swap*,*uncurry2 swap*) ∈
[λ((*l*,*i*),*j*). *i*<*length l* ∧ *j*<*length l*]$_f$ (⟨*A*⟩*list-rel* ×$_r$ *nat-rel*) ×$_r$ *nat-rel* → ⟨*A*⟩*list-rel*
⟨*proof*⟩

lemma *param-list-null*[*param*]: (*List.null*,*List.null*) ∈ ⟨*A*⟩*list-rel* → *bool-rel*
⟨*proof*⟩

3.5.4 Operations

sepref-decl-op *list-empty*: [] :: ⟨*A*⟩*list-rel* ⟨*proof*⟩
context notes [*simp*] = *eq-Nil-null* **begin**
sepref-decl-op *list-is-empty*: λ*l*. *l*=[] :: ⟨*A*⟩*list-rel* →$_f$ *bool-rel* ⟨*proof*⟩
end
sepref-decl-op *list-replicate*: *replicate* :: *nat-rel* → *A* → ⟨*A*⟩*list-rel* ⟨*proof*⟩
definition *op-list-copy* :: *'a list* ⇒ *'a list* **where** [*simp*]: *op-list-copy l* ≡ *l*
sepref-decl-op (*no-def*) *list-copy*: *op-list-copy* :: ⟨*A*⟩*list-rel* → ⟨*A*⟩*list-rel* ⟨*proof*⟩
sepref-decl-op *list-prepend*: (#) :: *A* → ⟨*A*⟩*list-rel* → ⟨*A*⟩*list-rel* ⟨*proof*⟩
sepref-decl-op *list-append*: λ*xs x*. *xs*@[*x*] :: ⟨*A*⟩*list-rel* → *A* → ⟨*A*⟩*list-rel* ⟨*proof*⟩
sepref-decl-op *list-concat*: (@) :: ⟨*A*⟩*list-rel* → ⟨*A*⟩*list-rel* → ⟨*A*⟩*list-rel* ⟨*proof*⟩
sepref-decl-op *list-length*: *length* :: ⟨*A*⟩*list-rel* → *nat-rel* ⟨*proof*⟩
sepref-decl-op *list-get*: *nth* :: [λ(*l*,*i*). *i*<*length l*]$_f$ ⟨*A*⟩*list-rel* ×$_r$ *nat-rel* → *A*
⟨*proof*⟩
sepref-decl-op *list-set*: *list-update* :: [λ((*l*,*i*),-). *i*<*length l*]$_f$ (⟨*A*⟩*list-rel* ×$_r$ *nat-rel*)
×$_r$ *A* → ⟨*A*⟩*list-rel* ⟨*proof*⟩
context notes [*simp*] = *eq-Nil-null* **begin**
sepref-decl-op *list-hd*: *hd* :: [λ*l*. *l*≠[]]$_f$ ⟨*A*⟩*list-rel* → *A* ⟨*proof*⟩
sepref-decl-op *list-tl*: *tl* :: [λ*l*. *l*≠[]]$_f$ ⟨*A*⟩*list-rel* → ⟨*A*⟩*list-rel* ⟨*proof*⟩
sepref-decl-op *list-last*: *last* :: [λ*l*. *l*≠[]]$_f$ ⟨*A*⟩*list-rel* → *A* ⟨*proof*⟩
sepref-decl-op *list-butlast*: *butlast* :: [λ*l*. *l*≠[]]$_f$ ⟨*A*⟩*list-rel* → ⟨*A*⟩*list-rel* ⟨*proof*⟩

end
sepref-decl-op *list-contains*: $\lambda x\ l.\ x \in set\ l :: A \to \langle A \rangle list\text{-}rel \to bool\text{-}rel$
where *single-valued A single-valued* (A^{-1}) ⟨*proof*⟩
sepref-decl-op *list-swap*: *swap* :: $[\lambda((l,i),j).\ i<length\ l \wedge j<length\ l]_f$ $(\langle A \rangle list\text{-}rel \times_r nat\text{-}rel) \times_r nat\text{-}rel \to \langle A \rangle list\text{-}rel$ ⟨*proof*⟩
sepref-decl-op *list-rotate1*: *rotate1* :: $\langle A \rangle list\text{-}rel \to \langle A \rangle list\text{-}rel$ ⟨*proof*⟩
sepref-decl-op *list-rev*: *rev* :: $\langle A \rangle list\text{-}rel \to \langle A \rangle list\text{-}rel$ ⟨*proof*⟩
sepref-decl-op *list-index*: *index* :: $\langle A \rangle list\text{-}rel \to A \to nat\text{-}rel$
where *single-valued A single-valued* (A^{-1}) ⟨*proof*⟩

3.5.5 Patterns

lemma [*def-pat-rules*]:
$[] \equiv op\text{-}list\text{-}empty$
$(=)\ \$l\$[] \equiv op\text{-}list\text{-}is\text{-}empty\$l$
$(=)\ \$[]\$l \equiv op\text{-}list\text{-}is\text{-}empty\$l$
$replicate\$n\$v \equiv op\text{-}list\text{-}replicate\$n\$v$
$Cons\$x\$xs \equiv op\text{-}list\text{-}prepend\$x\$xs$
$(@)\ \$xs\$(Cons\$x\$[]) \equiv op\text{-}list\text{-}append\$xs\$x$
$(@)\ \$xs\$ys \equiv op\text{-}list\text{-}concat\$xs\$ys$
$op\text{-}list\text{-}concat\$xs\$(Cons\$x\$[]) \equiv op\text{-}list\text{-}append\$xs\$x$
$length\$xs \equiv op\text{-}list\text{-}length\$xs$
$nth\$l\$i \equiv op\text{-}list\text{-}get\$l\$i$
$list\text{-}update\$l\$i\$x \equiv op\text{-}list\text{-}set\$l\$i\$x$
$hd\$l \equiv op\text{-}list\text{-}hd\$l$
$hd\$l \equiv op\text{-}list\text{-}hd\$l$
$tl\$l \equiv op\text{-}list\text{-}tl\$l$
$tl\$l \equiv op\text{-}list\text{-}tl\$l$
$last\$l \equiv op\text{-}list\text{-}last\$l$
$butlast\$l \equiv op\text{-}list\text{-}butlast\$l$
$(\in)\ \$x\$(set\$l) \equiv op\text{-}list\text{-}contains\$x\$l$
$swap\$l\$i\$j \equiv op\text{-}list\text{-}swap\$l\$i\$j$
$rotate1\$l \equiv op\text{-}list\text{-}rotate1\$l$
$rev\$l \equiv op\text{-}list\text{-}rev\$l$
$index\$l\$x \equiv op\text{-}list\text{-}index\$l\$x$
⟨*proof*⟩

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*]: $(x',x) \in \langle A \rangle list\text{-}rel \Longrightarrow x' \neq [] \longleftrightarrow x \neq []$ ⟨*proof*⟩
lemma *list-rel-pres-length*[*fcomp-prenorm-simps*]: $(x',x) \in \langle A \rangle list\text{-}rel \Longrightarrow length\ x' = length\ x$ ⟨*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*))) ∈ $unit\text{-}assn^k \to_a rel$
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 heap* = *'a list*

locale *heapstruct* =
fixes *prio* :: *'a* ⇒ *'b*::*linorder*
begin
definition *valid* :: *'a heap* ⇒ *nat* ⇒ *bool*

where *valid h i* ≡ *i*>*0* ∧ *i*≤*length h*

abbreviation α :: *'a heap* ⇒ *'a multiset* **where** α ≡ *mset*

lemma *valid-empty*[*simp*]: ¬*valid* [] *i* ⟨*proof*⟩
lemma *valid0*[*simp*]: ¬*valid h 0* ⟨*proof*⟩
lemma *valid-glen*[*simp*]: *i*>*length h* ⟹ ¬*valid h i* ⟨*proof*⟩

lemma *valid-len*[*simp*]: *h*≠[] ⟹ *valid h* (*length h*) ⟨*proof*⟩

lemma *validI*: *0*<*i* ⟹ *i*≤*length h* ⟹ *valid h i*
⟨*proof*⟩

definition *val-of* :: *'a heap* ⇒ *nat* ⇒ *'a* **where** *val-of l i* ≡ *l*!(*i*−*1*)
abbreviation *prio-of* :: *'a heap* ⇒ *nat* ⇒ *'b* **where**
prio-of l i ≡ *prio* (*val-of l i*)

Navigating the tree

definition *parent* :: *nat* ⇒ *nat* **where** *parent i* ≡ *i div 2*
definition *left* :: *nat* ⇒ *nat* **where** *left i* ≡ *2*∗*i*
definition *right* :: *nat* ⇒ *nat* **where** *right i* ≡ *2*∗*i* + *1*

abbreviation *has-parent h i* ≡ *valid h* (*parent i*)
abbreviation *has-left h i* ≡ *valid h* (*left i*)
abbreviation *has-right h i* ≡ *valid h* (*right i*)

abbreviation *vparent h i* == *val-of h* (*parent i*)
abbreviation *vleft h i* == *val-of h* (*left i*)
abbreviation *vright h i* == *val-of h* (*right i*)

abbreviation *pparent h i* == *prio-of h* (*parent i*)
abbreviation *pleft h i* == *prio-of h* (*left i*)
abbreviation *pright h i* == *prio-of h* (*right i*)

lemma *parent-left-id*[*simp*]: *parent* (*left i*) = *i*
⟨*proof*⟩

lemma *parent-right-id*[*simp*]: *parent* (*right i*) = *i*
⟨*proof*⟩

lemma *child-of-parentD*:
has-parent l i ⟹ *left* (*parent i*) = *i* ∨ *right* (*parent i*) = *i*
⟨*proof*⟩

lemma *rc-imp-lc*: ⟦*valid h i*; *has-right h i*⟧ ⟹ *has-left h i*
⟨*proof*⟩

lemma *plr-corner-cases*[*simp*]:
assumes *0<i*
shows
i≠parent i
i≠left i
i≠right i
parent i ≠ i
left i ≠ i
right i ≠ i
⟨*proof*⟩

lemma *i-eq-parent-conv*[*simp*]: *i=parent i ⟷ i=0*
⟨*proof*⟩

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 ⇒ bool*
where *heap-invar l*
≡ ∀ i. valid l i ⟶ has-parent l i ⟶ pparent l i ≤ prio-of l i

definition *heap-rel1 ≡ br α heap-invar*

lemma *heap-invar-empty*[*simp*]: *heap-invar []*
⟨*proof*⟩

function *heap-induction-scheme* :: *nat ⇒ unit* **where**
heap-induction-scheme i = (
if i>1 then heap-induction-scheme (parent i) else ())
⟨*proof*⟩

termination
⟨*proof*⟩

lemma
heap-parent-le: ⟦*heap-invar l*; *valid l i*; *has-parent l i*⟧
⟹ pparent l i ≤ prio-of l i
⟨*proof*⟩

lemma *heap-min-prop*:
assumes *H*: *heap-invar h*
assumes *V*: *valid h i*
shows *prio-of h (Suc 0) ≤ prio-of h i*
⟨*proof*⟩

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* ≡
(*has-left h i* ⟶ *p* ≤ *pleft h i*)
∧ (*has-right h i* ⟶ *p* ≤ *pright h i*)

definition *heap-invar′ h* ≡ ∀ *i*. *valid h i* ⟶ *children-ge h* (*prio-of h i*) *i*

lemma *heap-eq-heap′*:
shows *heap-invar h* ⟷ *heap-invar′ h*
⟨*proof*⟩

3.6.2 Basic Operations

The basic operations are the only operations that directly modify the underlying data structure.

Val-Of

abbreviation (*input*) *val-of-pre l i* ≡ *valid l i*
definition *val-of-op* :: *′a heap* ⇒ *nat* ⇒ *′a nres*
where *val-of-op l i* ≡ *ASSERT* (*i*>*0*) ≫ *mop-list-get l* (*i*−*1*)
lemma *val-of-correct*[*refine-vcg*]:
val-of-pre l i ⟹ *val-of-op l i* ≤ *SPEC* (λ*r*. *r* = *val-of l i*)
⟨*proof*⟩

abbreviation (*input*) *prio-of-pre* ≡ *val-of-pre*
definition *prio-of-op l i* ≡ *do* {*v* ← *val-of-op l i*; *RETURN* (*prio v*)}
lemma *prio-of-op-correct*[*refine-vcg*]:
prio-of-pre l i ⟹ *prio-of-op l i* ≤ *SPEC* (λ*r*. *r* = *prio-of l i*)
⟨*proof*⟩

Update

abbreviation *update-pre h i v* ≡ *valid h i*
definition *update* :: *′a heap* ⇒ *nat* ⇒ *′a* ⇒ *′a heap*
where *update h i v* ≡ *h*[*i* − *1* := *v*]
definition *update-op* :: *′a heap* ⇒ *nat* ⇒ *′a* ⇒ *′a heap nres*
where *update-op h i v* ≡ *ASSERT* (*i*>*0*) ≫ *mop-list-set h* (*i*−*1*) *v*
lemma *update-correct*[*refine-vcg*]:
update-pre h i v ⟹ *update-op h i v* ≤ *SPEC*(λ*r*. *r* = *update h i v*)
⟨*proof*⟩

lemma *update-valid*[*simp*]: *valid* (*update h i v*) *j* ⟷ *valid h j*
⟨*proof*⟩

lemma *val-of-update*[*simp*]: ⟦*update-pre h i v*; *valid h j*⟧ ⟹ *val-of* (*update h i v*) *j* = (
if i=*j then v else val-of h j*)
⟨*proof*⟩

lemma *length-update*[*simp*]: *length* (*update l i v*) = *length l*
⟨*proof*⟩

Exchange

Exchange two elements

definition *exch* :: *'a heap* ⇒ *nat* ⇒ *nat* ⇒ *'a heap* **where**
exch l i j ≡ *swap l* (*i* − *1*) (*j* − *1*)
abbreviation *exch-pre l i j* ≡ *valid l i* ∧ *valid l j*

definition *exch-op* :: *'a list* ⇒ *nat* ⇒ *nat* ⇒ *'a list nres*
where *exch-op l i j* ≡ *do* {
ASSERT (*i*>*0* ∧ *j*>*0*);
l ← *mop-list-swap l* (*i* − *1*) (*j* − *1*);
RETURN l
}

lemma *exch-op-alt*: *exch-op l i j* = *do* {
vi ← *val-of-op l i*;
vj ← *val-of-op l j*;
l ← *update-op l i vj*;
l ← *update-op l j vi*;
RETURN l }
⟨*proof*⟩

lemma *exch-op-correct*[*refine-vcg*]:
exch-pre l i j ⟹ *exch-op l i j* ≤ *SPEC* (λ*r*. *r* = *exch l i j*)
⟨*proof*⟩

lemma *valid-exch*[*simp*]: *valid* (*exch l i j*) *k* = *valid l k*
⟨*proof*⟩

lemma *val-of-exch*[*simp*]: ⟦*valid l i*; *valid l j*; *valid l k*⟧ ⟹
val-of (*exch l i j*) *k* = (
if k=*i then val-of l j*
else if k=*j then val-of l i*
else val-of l k
)
⟨*proof*⟩

lemma *exch-eq*[*simp*]: *exch h i i* = *h*
⟨*proof*⟩

lemma *α-exch*[*simp*]: ⟦*valid l i*; *valid l j*⟧
⟹ *α* (*exch l i j*) = *α l*
⟨*proof*⟩

lemma *length-exch*[*simp*]: *length* (*exch l i j*) = *length l*
⟨*proof*⟩

Butlast

Remove last element

abbreviation *butlast-pre* $l \equiv l \neq []$
definition *butlast-op* :: *'a heap* ⇒ *'a heap nres*
where *butlast-op* $l \equiv$ *mop-list-butlast* l
lemma *butlast-op-correct*[*refine-vcg*]:
$\langle proof \rangle$

lemma *valid-butlast-conv*[*simp*]: *valid* (*butlast* h) $i \longleftrightarrow$ *valid* h $i \wedge i <$ *length* h
$\langle proof \rangle$

lemma *valid-butlast*: *valid* (*butlast* h) $i \Longrightarrow$ *valid* h i
$\langle proof \rangle$

lemma *val-of-butlast*[*simp*]: ⟦*valid* h i; i<*length* h⟧
$\Longrightarrow$ *val-of* (*butlast* h) i = *val-of* h i
$\langle proof \rangle$

lemma *val-of-butlast'*[*simp*]:
valid (*butlast* h) $i \Longrightarrow$ *val-of* (*butlast* h) i = *val-of* h i
$\langle proof \rangle$

lemma α-*butlast*[*simp*]: ⟦ *length* $h \neq 0$ ⟧
$\Longrightarrow \alpha$ (*butlast* h) = α h − {# *val-of* h (*length* h)#}
$\langle proof \rangle$

lemma *heap-invar-butlast*[*simp*]: *heap-invar* $h \Longrightarrow$ *heap-invar* (*butlast* h)
$\langle proof \rangle$

Append

definition *append-op* :: *'a heap* ⇒ *'a* ⇒ *'a heap nres*
where *append-op* l $v \equiv$ *mop-list-append* l v
lemma *append-op-correct*[*refine-vcg*]:
append-op l $v \leq$ *SPEC* ($\lambda r.$ r = l@[v])
$\langle proof \rangle$

lemma *valid-append*[*simp*]: *valid* (l@[v]) $i \longleftrightarrow$ *valid* l $i \vee i$ = *length* l + *1*
$\langle proof \rangle$

lemma *val-of-append*[*simp*]: *valid* (l@[v]) $i \Longrightarrow$
val-of (l@[v]) i = (*if valid* l i *then val-of* l i *else* v)
$\langle proof \rangle$

lemma α-*append*[*simp*]: α (l@[v]) = α 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 heap* ⇒ *nat* ⇒ *bool*
where *swim-invar h i* ≡
valid h i
∧ (∀ *j*. *valid h j* ∧ *has-parent h j* ∧ *j*≠*i* ⟶ *pparent h j* ≤ *prio-of h j*)
∧ (*has-parent h i* ⟶
(∀ *j*. *valid h j* ∧ *has-parent h j* ∧ *parent j* = *i*
⟶ *pparent h i* ≤ *prio-of h j*))

Move up an element that is too small, until it fits

definition *swim-op* :: *'a heap* ⇒ *nat* ⇒ *'a heap nres* **where**
swim-op h i ≡ *do* {
RECT (λ*swim* (*h*,*i*). *do* {
ASSERT (*valid h i* ∧ *swim-invar h i*);
if has-parent h i then do {
ppi ← *prio-of-op h* (*parent i*);
pi ← *prio-of-op h i*;
if (¬*ppi* ≤ *pi*) *then do* {
h ← *exch-op h i* (*parent i*);
swim (*h*, *parent i*)
} *else*
RETURN h
} *else*
RETURN h
}) (*h*,*i*)
}

lemma *swim-invar-valid*: *swim-invar h i* ⟹ *valid h i*
⟨*proof*⟩

lemma *swim-invar-exit1*: ¬*has-parent h i* ⟹ *swim-invar h i* ⟹ *heap-invar h*
⟨*proof*⟩

lemma *swim-invar-exit2*: *pparent h i* ≤ *prio-of h i* ⟹ *swim-invar h i* ⟹ *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* ≤ *prio-of h i*
shows *swim-invar* (*update h i v*) *i*
⟨*proof*⟩

lemma *swim-op-correct*[*refine-vcg*]:
⟦*swim-invar h i*⟧ ⟹
swim-op h i ≤ *SPEC* (λ*h'*. *α h'* = *α h* ∧ *heap-invar h'* ∧ *length h'* = *length h*)
⟨*proof*⟩

Sink

Move down an element that is too big, until it fits in

```
definition sink-op :: 'a heap ⇒ nat ⇒ 'a heap nres where
  sink-op h i ≡ do {
    RECT (λsink (h,i). do {
      ASSERT (valid h i);
      if has-right h i then do {
        ASSERT (has-left h i);
        lp ← prio-of-op h (left i);
        rp ← prio-of-op h (right i);
        p ← prio-of-op h i;
        if (lp < p ∧ rp ≥ lp) then do {
          h ← exch-op h i (left i);
          sink (h,left i)
        } else if (rp<lp ∧ rp < p) then do {
          h ← exch-op h i (right i);
          sink (h,right i)
        } else
          RETURN h
      } else if (has-left h i) then do {
        lp ← prio-of-op h (left i);
        p ← prio-of-op h i;
        if (lp < p) then do {
          h ← exch-op h i (left i);
          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* ≡
  valid l i
∧ (∀ *j*. *valid l j* ∧ *j*≠*i* ⟶ *children-ge l* (*prio-of l j*) *j*)
∧ (*has-parent l i* ⟶ *children-ge l* (*pparent l i*) *i*)

lemma *sink-invar-valid*: *sink-invar l i* ⟹ *valid l i*
  ⟨*proof*⟩

lemma *sink-invar-exit*: ⟦*sink-invar l i*; *children-ge l* (*prio-of l i*) *i*⟧
  ⟹ *heap-invar′ l*
  ⟨*proof*⟩

lemma *sink-aux1*: ¬ (*2*∗*i* ≤ *length h*) ⟹ ¬*has-left h i* ∧ ¬*has-right h i*
  ⟨*proof*⟩

lemma *sink-invar-pres1*:
  assumes *sink-invar h i*
  assumes *has-left h i has-right h i*
  assumes *prio-of h i* ≥ *pleft h i*
  assumes *pleft h i* ≥ *pright h i*
  shows *sink-invar* (*exch h i* (*right i*)) (*right i*)
  ⟨*proof*⟩

lemma *sink-invar-pres2*:
  assumes *sink-invar h i*
  assumes *has-left h i has-right h i*
  assumes *prio-of h i* ≥ *pleft h i*
  assumes *pleft h i* ≤ *pright h i*
  shows *sink-invar* (*exch h i* (*left i*)) (*left i*)
  ⟨*proof*⟩

lemma *sink-invar-pres3*:
  assumes *sink-invar h i*
  assumes *has-left h i has-right h i*
  assumes *prio-of h i* ≥ *pright h i*
  assumes *pleft h i* ≤ *pright h i*
  shows *sink-invar* (*exch h i* (*left i*)) (*left i*)
  ⟨*proof*⟩

lemma *sink-invar-pres4*:
  assumes *sink-invar h i*

assumes *has-left h i has-right h i*
assumes *prio-of h i ≥ pright h i*
assumes *pleft h i ≥ pright h i*
shows *sink-invar* (*exch h i* (*right i*)) (*right i*)
⟨*proof*⟩

lemma *sink-invar-pres5*:
assumes *sink-invar h i*
assumes *has-left h i* ¬*has-right h i*
assumes *prio-of h i ≥ pleft h i*
shows *sink-invar* (*exch h i* (*left i*)) (*left i*)
⟨*proof*⟩

lemmas *sink-invar-pres* =
sink-invar-pres1
sink-invar-pres2
sink-invar-pres3
sink-invar-pres4
sink-invar-pres5

lemma *sink-invar-incr*:
assumes *INV*: *heap-invar h*
assumes *V*: *valid h i*
assumes *INCR*: *prio v ≥ prio-of h i*
shows *sink-invar* (*update h i v*) *i*
⟨*proof*⟩

lemma *sink-op-correct*[*refine-vcg*]:
⟦*sink-invar h i*⟧ ⟹
sink-op h i ≤ SPEC (λ*h'*. *α h' = α h ∧ heap-invar h' ∧ length h' = length h*)
⟨*proof*⟩

lemma *sink-op-swim-rule*:
swim-invar h i ⟹ *sink-op h i ≤ SPEC* (λ*h'*. *h'=h*)
⟨*proof*⟩

definition *sink-op-opt*
— Sink operation as presented in Sedgewick et al. Algs4 reference implementation
where
sink-op-opt h k ≡ RECT (λ*D* (*h*,*k*). *do* {
ASSERT (*k>0 ∧ k≤length h*);
let len = length h;
if (*2∗k ≤ len*) *then do* {
let j = 2∗k;
pj ← *prio-of-op h j*;

```
    j ← (
      if j<len then do {
        psj ← prio-of-op h (Suc j);
        if pj>psj then RETURN (j+1) else RETURN j
      } else RETURN j);

    pj ← prio-of-op h j;
    pk ← prio-of-op h k;
    if (pk > pj) then do {
      h ← 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 ≡ do {
  h ← sink-op h i;
  h ← 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*: ⟦*heap-invar h*; *valid h i*⟧ ⟹ *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* ≤ *SPEC* (λ*h'*.
heap-invar h' ∧ *α h'* = *α (update h i v)* ∧ *length h'* = *length h*)
⟨*proof*⟩

3.6.4 Operations

Empty

abbreviation (*input*) *empty* :: *'a heap* — The empty heap
where *empty* ≡ []
definition *empty-op* :: *'a heap nres*
where *empty-op* ≡ *mop-list-empty*
lemma *empty-op-correct*[*refine-vcg*]:
empty-op ≤ *SPEC* (λ*r*. α *r* = {#} ∧ *heap-invar r*)
⟨*proof*⟩

Emptiness check

definition *is-empty-op* :: *'a heap* ⇒ *bool nres* — Check for emptiness
where *is-empty-op h* ≡ *do* {*ASSERT* (*heap-invar h*); *let l*=*length h*; *RETURN* (*l*=*0*)}
lemma *is-empty-op-correct*[*refine-vcg*]:
heap-invar h ⟹ *is-empty-op h* ≤ *SPEC* (λ*r*. *r*⟷α *h* = {#})
⟨*proof*⟩

Insert

definition *insert-op* :: *'a* ⇒ *'a heap* ⇒ *'a heap nres* — Insert element
where *insert-op v h* ≡ *do* {
ASSERT (*heap-invar h*);
h ← *append-op h v*;
let l = *length h*;
h ← *swim-op h l*;
RETURN h
}

lemma *swim-invar-insert*: *heap-invar l* ⟹ *swim-invar* (*l*@[*x*]) (*Suc* (*length l*))
⟨*proof*⟩

lemma
(*insert-op*,*RETURN oo op-mset-insert*) ∈ *Id* → *heap-rel1* → ⟨*heap-rel1*⟩*nres-rel*
⟨*proof*⟩

lemma *insert-op-correct*:
heap-invar h ⟹ *insert-op v h* ≤ *SPEC* (λ*h'*. *heap-invar h'* ∧ α *h'* = α *h* + {#*v*#})
⟨*proof*⟩
lemmas [*refine-vcg*] = *insert-op-correct*

Pop minimum element

definition *pop-min-op* :: *'a heap* ⇒ (*'a* × *'a heap*) *nres* **where**
pop-min-op h ≡ *do* {
ASSERT (*heap-invar h*);
ASSERT (*valid h 1*);

```
    m ← val-of-op h 1;
    let l = length h;
    h ← exch-op h 1 l;
    h ← butlast-op h;

    if (l≠1) then do {
      h ← sink-op h 1;
      RETURN (m,h)
    } else RETURN (m,h)
  }
```

lemma *left-not-one*[*simp*]: *left j ≠ Suc 0*
⟨*proof*⟩

lemma *right-one-conv*[*simp*]: *right j = Suc 0 ⟷ j=0*
⟨*proof*⟩

lemma *parent-one-conv*[*simp*]: *parent (Suc 0) = 0*
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *in-set-conv-val*: *v ∈ set h ⟷ (∃ i. valid h i ∧ v = val-of h i)*
⟨*proof*⟩

lemma *pop-min-op-correct*:
assumes *heap-invar h α h ≠ {#}*
shows *pop-min-op h ≤ SPEC (λ(v,h′). heap-invar h′ ∧*
v ∈# α h ∧ α h′ = α h − {#v#} ∧ (∀ v′∈set-mset (α h). prio v ≤ prio v′))
⟨*proof*⟩

lemmas [*refine-vcg*] = *pop-min-op-correct*

Peek minimum element

definition *peek-min-op* :: *′a heap ⇒ ′a nres* **where**
peek-min-op h ≡ do {
ASSERT (heap-invar h);
ASSERT (valid h 1);
val-of-op h 1
}

lemma *peek-min-op-correct*:
assumes *heap-invar h α h* ≠ {#}
shows *peek-min-op h* ≤ *SPEC* (λ*v*.
v ∈# *α h* ∧ (∀ *v′*∈*set-mset* (*α h*). *prio v* ≤ *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*)∈⟨*heap-rel1*⟩*nres-rel*
⟨*proof*⟩

lemma *is-empty-op-refine*: (*is-empty-op*,*RETURN o op-mset-is-empty*) ∈ *heap-rel1* → ⟨*bool-rel*⟩*nres-rel*
⟨*proof*⟩

lemma *insert-op-refine*: (*insert-op*,*RETURN oo op-mset-insert*) ∈ *Id* → *heap-rel1* → ⟨*heap-rel1*⟩*nres-rel*
⟨*proof*⟩

lemma *pop-min-op-refine*:
(*pop-min-op*, *PR-CONST* (*mop-prio-pop-min prio*)) ∈ *heap-rel1* → ⟨*Id* ×*r* *heap-rel1*⟩*nres-rel*
⟨*proof*⟩

lemma *peek-min-op-refine*:
(*peek-min-op*, *PR-CONST* (*mop-prio-peek-min prio*)) ∈ *heap-rel1* → ⟨*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*<*length l*
assumes ⋀*l1 x l2*. ⟦*l*=*l1*@*x*#*l2*; *i* = *length l1* ⟧ ⟹ *P* (*l1*@*x*#*l2*)
shows *P l*
⟨*proof*⟩ **lemma** *list-set-entails-aux*:
shows *list-assn A l li* ∗ *A x xi* ⟹*A* *list-assn A* (*l*[*i* := *x*]) (*li*[*i* := *xi*]) ∗ *true*
⟨*proof*⟩ **lemma** *list-set-hd-tl-aux*:
a ≠ [] ⟹ *list-assn R a c* ⟹*A* *R* (*hd a*) (*hd c*) ∗ *true*

$a \neq [] \Longrightarrow$ *list-assn R a c* $\Longrightarrow_A$ *list-assn R* (*tl a*) (*tl c*) $*$ *true*
⟨*proof*⟩ **lemma** *list-set-last-butlast-aux*:
$a \neq [] \Longrightarrow$ *list-assn R a c* $\Longrightarrow_A$ *R* (*last a*) (*last c*) $*$ *true*
$a \neq [] \Longrightarrow$ *list-assn R a c* $\Longrightarrow_A$ *list-assn R* (*butlast a*) (*butlast c*) $*$ *true*
⟨*proof*⟩ **lemma** *swap-decomp-simp*[*simp*]:
swap (*l1* @ *x* # *c21'* @ *xa* # *l2a*) (*length l1*) (*Suc* (*length l1* + *length c21'*)) = *l1*@*xa*#*c21'*@*x*#*l2a*
swap (*l1* @ *x* # *c21'* @ *xa* # *l2a*) (*Suc* (*length l1* + *length c21'*)) (*length l1*) = *l1*@*xa*#*c21'*@*x*#*l2a*
⟨*proof*⟩ **lemma** *list-swap-aux*: ⟦*i* < *length l*; *j* < *length l*⟧ $\Longrightarrow$ *list-assn A l li* $\Longrightarrow_A$ *list-assn A* (*swap l i j*) (*swap li i j*) $*$ *true*
⟨*proof*⟩ **lemma** *list-rotate1-aux*: *list-assn A a c* $\Longrightarrow_A$ *list-assn A* (*rotate1 a*) (*rotate1 c*) $*$ *true*
⟨*proof*⟩ **lemma** *list-rev-aux*: *list-assn A a c* $\Longrightarrow_A$ *list-assn A* (*rev a*) (*rev c*) $*$ *true*
⟨*proof*⟩

lemma *mod-starE*:
assumes *h* $\models$ *A*$*$*B*
obtains *h1 h2* **where** *h1*$\models$*A h2*$\models$*B*
⟨*proof*⟩ **lemma** *CONSTRAINT-is-pureE*:
assumes *CONSTRAINT is-pure A*
obtains *R* **where** *A*=*pure R*
⟨*proof*⟩ **method** *solve-dbg* =
((*elim CONSTRAINT-is-pureE*; (*simp only*: *list-assn-pure-conv the-pure-pure*)*?*)*?*;
sep-auto
simp: *pure-def hn-ctxt-def invalid-assn-def list-assn-aux-eqlen-simp*
intro!: *hn-refineI*[*THEN hn-refine-preI*] *hfrefI*
elim!: *mod-starE*
intro: *list-set-entails-aux list-set-hd-tl-aux list-set-last-butlast-aux*
list-swap-aux list-rotate1-aux list-rev-aux
;
((*rule entails-preI*; *sep-auto simp*: *list-assn-aux-eqlen-simp* | (*parametricity*; *simp*; *fail*))*?*)
)

private method *solve* = *solve-dbg*; *fail*

lemma *HOL-list-empty-hnr-aux*: (*uncurry0* (*return op-list-empty*), *uncurry0* (*RETURN op-list-empty*)) $\in$ *unit-assn*^{k} $\rightarrow_a$ (*list-assn A*) ⟨*proof*⟩
lemma *HOL-list-is-empty-hnr*[*sepref-fr-rules*]: (*return* $\circ$ *op-list-is-empty*, *RETURN* $\circ$ *op-list-is-empty*) $\in$ (*list-assn A*)^k $\rightarrow_a$ *bool-assn* ⟨*proof*⟩
lemma *HOL-list-prepend-hnr*[*sepref-fr-rules*]: (*uncurry* (*return* $\circ\circ$ *op-list-prepend*), *uncurry* (*RETURN* $\circ\circ$ *op-list-prepend*)) $\in$ A^d $*_a$ (*list-assn A*)^d $\rightarrow_a$ *list-assn A* ⟨*proof*⟩
lemma *HOL-list-append-hnr*[*sepref-fr-rules*]: (*uncurry* (*return* $\circ\circ$ *op-list-append*), *uncurry* (*RETURN* $\circ\circ$ *op-list-append*)) $\in$ (*list-assn A*)^d $*_a$ A^d $\rightarrow_a$ *list-assn A*

⟨*proof*⟩
lemma *HOL-list-concat-hnr*[*sepref-fr-rules*]: (*uncurry* (*return* ∘∘ *op-list-concat*), *uncurry* (*RETURN* ∘∘ *op-list-concat*)) ∈ (*list-assn A*)^d *$_a$ (*list-assn A*)^d →$_a$ *list-assn A* ⟨*proof*⟩
lemma *HOL-list-length-hnr*[*sepref-fr-rules*]: (*return* ∘ *op-list-length*, *RETURN* ∘ *op-list-length*) ∈ (*list-assn A*)^k →$_a$ *nat-assn* ⟨*proof*⟩
lemma *HOL-list-set-hnr*[*sepref-fr-rules*]: (*uncurry2* (*return* ∘∘∘ *op-list-set*), *uncurry2* (*RETURN* ∘∘∘ *op-list-set*)) ∈ (*list-assn A*)^d *$_a$ *nat-assn*^{k} *$_a$ *A*^{d} →$_a$ *list-assn A* ⟨*proof*⟩
lemma *HOL-list-hd-hnr*[*sepref-fr-rules*]: (*return* ∘ *op-list-hd*, *RETURN* ∘ *op-list-hd*) ∈ [λ*y*. *y* ≠ []]$_a$ (*list-assn R*)^d → *R* ⟨*proof*⟩
lemma *HOL-list-tl-hnr*[*sepref-fr-rules*]: (*return* ∘ *op-list-tl*, *RETURN* ∘ *op-list-tl*) ∈ [λ*y*. *y* ≠ []]$_a$ (*list-assn A*)^d → *list-assn A* ⟨*proof*⟩
lemma *HOL-list-last-hnr*[*sepref-fr-rules*]: (*return* ∘ *op-list-last*, *RETURN* ∘ *op-list-last*) ∈ [λ*y*. *y* ≠ []]$_a$ (*list-assn R*)^d → *R* ⟨*proof*⟩
lemma *HOL-list-butlast-hnr*[*sepref-fr-rules*]: (*return* ∘ *op-list-butlast*, *RETURN* ∘ *op-list-butlast*) ∈ [λ*y*. *y* ≠ []]$_a$ (*list-assn A*)^d → *list-assn A* ⟨*proof*⟩
lemma *HOL-list-swap-hnr*[*sepref-fr-rules*]: (*uncurry2* (*return* ∘∘∘ *op-list-swap*), *uncurry2* (*RETURN* ∘∘∘ *op-list-swap*))
∈ [λ((*a*, *b*), *ba*). *b* < *length a* ∧ *ba* < *length a*]$_a$ (*list-assn A*)^d *$_a$ *nat-assn*^{k} *$_a$ *nat-assn*^{k} → *list-assn A* ⟨*proof*⟩
lemma *HOL-list-rotate1-hnr*[*sepref-fr-rules*]: (*return* ∘ *op-list-rotate1*, *RETURN* ∘ *op-list-rotate1*) ∈ (*list-assn A*)^d →$_a$ *list-assn A* ⟨*proof*⟩
lemma *HOL-list-rev-hnr*[*sepref-fr-rules*]: (*return* ∘ *op-list-rev*, *RETURN* ∘ *op-list-rev*) ∈ (*list-assn A*)^d →$_a$ *list-assn A* ⟨*proof*⟩

lemma *HOL-list-replicate-hnr*[*sepref-fr-rules*]: *CONSTRAINT is-pure A* ⟹ (*uncurry* (*return* ∘∘ *op-list-replicate*), *uncurry* (*RETURN* ∘∘ *op-list-replicate*)) ∈ *nat-assn*^{k} *$_a$ *A*^{k} →$_a$ *list-assn A* ⟨*proof*⟩
lemma *HOL-list-get-hnr*[*sepref-fr-rules*]: *CONSTRAINT is-pure A* ⟹ (*uncurry* (*return* ∘∘ *op-list-get*), *uncurry* (*RETURN* ∘∘ *op-list-get*)) ∈ [λ(*a*, *b*). *b* < *length a*]$_a$ (*list-assn A*)^k *$_a$ *nat-assn*^{k} → *A* ⟨*proof*⟩ **lemma** *bool-by-paramE*: ⟦ *a*; (*a*,*b*)∈*Id* ⟧ ⟹ *b* ⟨*proof*⟩ **lemma** *bool-by-paramE′*: ⟦ *a*; (*b*,*a*)∈*Id* ⟧ ⟹ *b* ⟨*proof*⟩

lemma *HOL-list-contains-hnr*[*sepref-fr-rules*]: ⟦*CONSTRAINT is-pure A*; *single-valued* (*the-pure A*); *single-valued* ((*the-pure A*)$^{-1}$)⟧
⟹ (*uncurry* (*return* ∘∘ *op-list-contains*), *uncurry* (*RETURN* ∘∘ *op-list-contains*)) ∈ *A*^{k} *$_a$ (*list-assn A*)^k →$_a$ *bool-assn*
⟨*proof*⟩

lemmas *HOL-list-empty-hnr-mop* = *HOL-list-empty-hnr-aux*[*FCOMP mk-mop-rl0-np*[*OF mop-list-empty-alt*]]
lemmas *HOL-list-is-empty-hnr-mop*[*sepref-fr-rules*] = *HOL-list-is-empty-hnr*[*FCOMP mk-mop-rl1-np*[*OF mop-list-is-empty-alt*]]
lemmas *HOL-list-prepend-hnr-mop*[*sepref-fr-rules*] = *HOL-list-prepend-hnr*[*FCOMP mk-mop-rl2-np*[*OF mop-list-prepend-alt*]]
lemmas *HOL-list-append-hnr-mop*[*sepref-fr-rules*] = *HOL-list-append-hnr*[*FCOMP mk-mop-rl2-np*[*OF mop-list-append-alt*]]

lemmas *HOL-list-concat-hnr-mop*[*sepref-fr-rules*] = *HOL-list-concat-hnr*[*FCOMP mk-mop-rl2-np*[*OF mop-list-concat-alt*]]
lemmas *HOL-list-length-hnr-mop*[*sepref-fr-rules*] = *HOL-list-length-hnr*[*FCOMP mk-mop-rl1-np*[*OF mop-list-length-alt*]]
lemmas *HOL-list-set-hnr-mop*[*sepref-fr-rules*] = *HOL-list-set-hnr*[*FCOMP mk-mop-rl3*[*OF mop-list-set-alt*]]
lemmas *HOL-list-hd-hnr-mop*[*sepref-fr-rules*] = *HOL-list-hd-hnr*[*FCOMP mk-mop-rl1*[*OF mop-list-hd-alt*]]
lemmas *HOL-list-tl-hnr-mop*[*sepref-fr-rules*] = *HOL-list-tl-hnr*[*FCOMP mk-mop-rl1*[*OF mop-list-tl-alt*]]
lemmas *HOL-list-last-hnr-mop*[*sepref-fr-rules*] = *HOL-list-last-hnr*[*FCOMP mk-mop-rl1*[*OF mop-list-last-alt*]]
lemmas *HOL-list-butlast-hnr-mop*[*sepref-fr-rules*] = *HOL-list-butlast-hnr*[*FCOMP mk-mop-rl1*[*OF mop-list-butlast-alt*]]
lemmas *HOL-list-swap-hnr-mop*[*sepref-fr-rules*] = *HOL-list-swap-hnr*[*FCOMP mk-mop-rl3*[*OF mop-list-swap-alt*]]
lemmas *HOL-list-rotate1-hnr-mop*[*sepref-fr-rules*] = *HOL-list-rotate1-hnr*[*FCOMP mk-mop-rl1-np*[*OF mop-list-rotate1-alt*]]
lemmas *HOL-list-rev-hnr-mop*[*sepref-fr-rules*] = *HOL-list-rev-hnr*[*FCOMP mk-mop-rl1-np*[*OF mop-list-rev-alt*]]
lemmas *HOL-list-replicate-hnr-mop*[*sepref-fr-rules*] = *HOL-list-replicate-hnr*[*FCOMP mk-mop-rl2-np*[*OF mop-list-replicate-alt*]]
lemmas *HOL-list-get-hnr-mop*[*sepref-fr-rules*] = *HOL-list-get-hnr*[*FCOMP mk-mop-rl2*[*OF mop-list-get-alt*]]
lemmas *HOL-list-contains-hnr-mop*[*sepref-fr-rules*] = *HOL-list-contains-hnr*[*FCOMP mk-mop-rl2-np*[*OF mop-list-contains-alt*]]

lemmas *HOL-list-empty-hnr* = *HOL-list-empty-hnr-aux HOL-list-empty-hnr-mop*

end

definition [*simp*]: *op-HOL-list-empty* ≡ *op-list-empty*
interpretation *HOL-list*: *list-custom-empty list-assn A return* [] *op-HOL-list-empty*
⟨*proof*⟩

schematic-goal
notes [*sepref-fr-rules*] = *HOL-list-empty-hnr*
shows
hn-refine (*emp*) (*?c*::*?'c Heap*) *?Γ' ?R* (*do* {
x ← *RETURN* [*1*,*2*,*3*::*nat*];
let x2 = *op-list-append x 5*;
ASSERT (*length x* = *4*);
let x = *op-list-swap x 1 2*;
x ← *mop-list-swap x 1 2*;
RETURN (*x*@*x*)
})
⟨*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* ≡ λ(*a*,*n*). ∃$_A$*l'*. *a* ↦$_a$ *l'* ∗ ↑(*n* ≤ *length l'* ∧ *l* = *take n l'* ∧ *length l'*>*0*)

lemma *is-array-list-prec*[*safe-constraint-rules*]: *precise is-array-list*
⟨*proof*⟩

definition *initial-capacity* ≡ *16*::*nat*
definition *minimum-capacity* ≡ *16*::*nat*

definition *arl-empty* ≡ *do* {
a ← *Array.new initial-capacity default*;
return (*a*,*0*)
}

definition *arl-empty-sz init-cap* ≡ *do* {
a ← *Array.new* (*max init-cap minimum-capacity*) *default*;
return (*a*,*0*)
}

definition *arl-append* ≡ λ(*a*,*n*) *x*. *do* {
len ← *Array.len a*;

if n<*len then do* {
a ← *Array.upd n x a*;
return (*a*,*n*+*1*)
} *else do* {
let newcap = *2* ∗ *len*;
a ← *array-grow a newcap default*;
a ← *Array.upd n x a*;
return (*a*,*n*+*1*)
}
}

definition *arl-copy* ≡ λ(*a*,*n*). *do* {
a ← *array-copy a*;
return (*a*,*n*)
}

definition *arl-length* :: *'a*::*heap array-list* ⇒ *nat Heap* **where**
arl-length ≡ λ(*a*,*n*). *return* (*n*)

definition *arl-is-empty* :: *'a::heap array-list* ⇒ *bool Heap* **where**
arl-is-empty ≡ λ(*a*,*n*). *return* (*n*=*0*)

definition *arl-last* :: *'a::heap array-list* ⇒ *'a Heap* **where**
arl-last ≡ λ(*a*,*n*). *do* {
Array.nth a (*n* − *1*)
}

definition *arl-butlast* :: *'a::heap array-list* ⇒ *'a array-list Heap* **where**
arl-butlast ≡ λ(*a*,*n*). *do* {
let n = *n* − *1*;
len ← *Array.len a*;
if (*n*∗*4* < *len* ∧ *n*∗*2*≥*minimum-capacity*) *then do* {
a ← *array-shrink a* (*n*∗*2*);
return (*a*,*n*)
} *else*
return (*a*,*n*)
}

definition *arl-get* :: *'a::heap array-list* ⇒ *nat* ⇒ *'a Heap* **where**
arl-get ≡ λ(*a*,*n*) *i*. *Array.nth a i*

definition *arl-set* :: *'a::heap array-list* ⇒ *nat* ⇒ *'a* ⇒ *'a array-list Heap* **where**
arl-set ≡ λ(*a*,*n*) *i x*. *do* { *a* ← *Array.upd i x a*; *return* (*a*,*n*)}

lemma *arl-empty-rule*[*sep-heap-rules*]: < *emp* > *arl-empty* <*is-array-list* []>
⟨*proof*⟩

lemma *arl-empty-sz-rule*[*sep-heap-rules*]: < *emp* > *arl-empty-sz N* <*is-array-list* []>
⟨*proof*⟩

lemma *arl-copy-rule*[*sep-heap-rules*]: < *is-array-list l a* > *arl-copy a* <λ*r*. *is-array-list l a* ∗ *is-array-list l r*>
⟨*proof*⟩

lemma *arl-append-rule*[*sep-heap-rules*]:
< *is-array-list l a* >
arl-append a x
<λ*a*. *is-array-list* (*l*@[*x*]) *a* >$_t$
⟨*proof*⟩

lemma *arl-length-rule*[*sep-heap-rules*]:
<*is-array-list l a*>
arl-length a
<λ*r*. *is-array-list l a* ∗ ↑(*r*=*length l*)>
⟨*proof*⟩

lemma *arl-is-empty-rule*[*sep-heap-rules*]:
<*is-array-list l a*>
arl-is-empty a
<λ*r*. *is-array-list l a* ∗ ↑(*r*⟷(*l*=[]))>
⟨*proof*⟩

lemma *arl-last-rule*[*sep-heap-rules*]:
l≠[] ⟹
<*is-array-list l a*>
arl-last a
<λ*r*. *is-array-list l a* ∗ ↑(*r*=*last l*)>
⟨*proof*⟩

lemma *arl-butlast-rule*[*sep-heap-rules*]:
l≠[] ⟹
<*is-array-list l a*>
arl-butlast a
<*is-array-list* (*butlast l*)>$_t$
⟨*proof*⟩

lemma *arl-get-rule*[*sep-heap-rules*]:
i<*length l* ⟹
<*is-array-list l a*>
arl-get a i
<λ*r*. *is-array-list l a* ∗ ↑(*r*=*l*!*i*)>
⟨*proof*⟩

lemma *arl-set-rule*[*sep-heap-rules*]:
i<*length l* ⟹
<*is-array-list l a*>
arl-set a i x
<*is-array-list* (*l*[*i*:=*x*])>
⟨*proof*⟩

definition *arl-assn A* ≡ *hr-comp is-array-list* (⟨*the-pure A*⟩*list-rel*)
lemmas [*safe-constraint-rules*] = *CN-FALSEI*[*of is-pure arl-assn A* **for** *A*]

lemma *arl-assn-comp*: *is-pure A* ⟹ *hr-comp* (*arl-assn A*) (⟨*B*⟩*list-rel*) = *arl-assn* (*hr-comp A B*)
⟨*proof*⟩

lemma *arl-assn-comp'*: *hr-comp* (*arl-assn id-assn*) (⟨*B*⟩*list-rel*) = *arl-assn* (*pure B*)
⟨*proof*⟩

context
notes [*fcomp-norm-unfold*] = *arl-assn-def*[*symmetric*] *arl-assn-comp'*

notes [*intro*!] = *hfrefI hn-refineI*[*THEN hn-refine-preI*]
notes [*simp*] = *pure-def hn-ctxt-def invalid-assn-def*
begin

lemma *arl-empty-hnr-aux*: (*uncurry0 arl-empty*,*uncurry0* (*RETURN op-list-empty*)) ∈ *unit-assn*^{k} →$_a$ *is-array-list*
⟨*proof*⟩
sepref-decl-impl (*no-register*) *arl-empty*: *arl-empty-hnr-aux* ⟨*proof*⟩

lemma *arl-empty-sz-hnr-aux*: (*uncurry0* (*arl-empty-sz N*),*uncurry0* (*RETURN op-list-empty*)) ∈ *unit-assn*^{k} →$_a$ *is-array-list*
⟨*proof*⟩
sepref-decl-impl (*no-register*) *arl-empty-sz*: *arl-empty-sz-hnr-aux* ⟨*proof*⟩

definition *op-arl-empty* ≡ *op-list-empty*
definition *op-arl-empty-sz* (*N*::*nat*) ≡ *op-list-empty*

lemma *arl-copy-hnr-aux*: (*arl-copy*,*RETURN o op-list-copy*) ∈ *is-array-list*^{k} →$_a$ *is-array-list*
⟨*proof*⟩
sepref-decl-impl *arl-copy*: *arl-copy-hnr-aux* ⟨*proof*⟩

lemma *arl-append-hnr-aux*: (*uncurry arl-append*,*uncurry* (*RETURN oo op-list-append*)) ∈ (*is-array-list*^{d} ∗$_a$ *id-assn*^{k}) →$_a$ *is-array-list*
⟨*proof*⟩
sepref-decl-impl *arl-append*: *arl-append-hnr-aux* ⟨*proof*⟩

lemma *arl-length-hnr-aux*: (*arl-length*,*RETURN o op-list-length*) ∈ *is-array-list*^{k} →$_a$ *nat-assn*
⟨*proof*⟩
sepref-decl-impl *arl-length*: *arl-length-hnr-aux* ⟨*proof*⟩

lemma *arl-is-empty-hnr-aux*: (*arl-is-empty*,*RETURN o op-list-is-empty*) ∈ *is-array-list*^{k} →$_a$ *bool-assn*
⟨*proof*⟩
sepref-decl-impl *arl-is-empty*: *arl-is-empty-hnr-aux* ⟨*proof*⟩

lemma *arl-last-hnr-aux*: (*arl-last*,*RETURN o op-list-last*) ∈ [*pre-list-last*]$_a$ *is-array-list*^{k} → *id-assn*
⟨*proof*⟩
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-list*^{d} → *is-array-list*
⟨*proof*⟩
sepref-decl-impl *arl-butlast*: *arl-butlast-hnr-aux* ⟨*proof*⟩

lemma *arl-get-hnr-aux*: (*uncurry arl-get*,*uncurry* (*RETURN oo op-list-get*)) ∈

$[\lambda(l,i).\ i<\textit{length}\ l]_a\ (\textit{is-array-list}^k\ *_a\ \textit{nat-assn}^k) \to \textit{id-assn}$
⟨*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*))
$\in [\lambda((l,i),\text{-}).\ i<\textit{length}\ l]_a\ (\textit{is-array-list}^d\ *_a\ \textit{nat-assn}^k\ *_a\ \textit{id-assn}^k) \to \textit{is-array-list}$
⟨*proof*⟩
sepref-decl-impl *arl-set*: *arl-set-hnr-aux* ⟨*proof*⟩

sepref-definition *arl-swap* **is** *uncurry2 mop-list-swap* :: $((\textit{arl-assn id-assn})^d\ *_a\ \textit{nat-assn}^k\ *_a\ \textit{nat-assn}^k \to_a \textit{arl-assn id-assn})$
⟨*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*} ⇒ *'b*::*linorder*
begin

  interpretation *heapstruct-impl prio* ⟨*proof*⟩

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* :: $rel^d *_a nat\text{-}assn^k *_a id\text{-}assn^k \rightarrow_a rel$
⟨*proof*⟩
lemmas [*sepref-fr-rules*] = *update-impl.refine*

sepref-definition *val-of-impl* **is** *uncurry val-of-op* :: $rel^k *_a nat\text{-}assn^k \rightarrow_a id\text{-}assn$
⟨*proof*⟩
lemmas [*sepref-fr-rules*] = *val-of-impl.refine*

sepref-definition *exch-impl* **is** *uncurry2 exch-op* :: $rel^d *_a nat\text{-}assn^k *_a nat\text{-}assn^k \rightarrow_a rel$
⟨*proof*⟩
lemmas [*sepref-fr-rules*] = *exch-impl.refine*

sepref-definition *valid-impl* **is** *uncurry* (*RETURN oo valid*) :: $rel^k *_a nat\text{-}assn^k \rightarrow_a bool\text{-}assn$
⟨*proof*⟩
lemmas [*sepref-fr-rules*] = *valid-impl.refine*

sepref-definition *prio-of-impl* **is** *uncurry* (*PR-CONST prio-of-op*) :: $rel^k *_a nat\text{-}assn^k \rightarrow_a id\text{-}assn$
⟨*proof*⟩
lemmas [*sepref-fr-rules*] = *prio-of-impl.refine*

sepref-definition *swim-impl* **is** *uncurry* (*PR-CONST swim-op*) :: $rel^d *_a nat\text{-}assn^k \rightarrow_a rel$
⟨*proof*⟩

lemmas [*sepref-fr-rules*] = *swim-impl.refine*

sepref-definition *sink-impl* **is** *uncurry* (*PR-CONST sink-op*) :: $rel^d *_a nat\text{-}assn^k \rightarrow_a rel$
⟨*proof*⟩
lemmas [*sepref-fr-rules*] = *sink-impl.refine*

lemmas [*fcomp-norm-unfold*] = *heap-rel-def*[*symmetric*]

sepref-definition *empty-impl* **is** *uncurry0 empty-op* :: $unit\text{-}assn^k \rightarrow_a rel$
⟨*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* :: $(id\text{-}assn^k *_a rel^d \to_a rel)$
⟨*proof*⟩
sepref-decl-impl *heap-insert*: *insert-impl.refine*[*FCOMP insert-op-refine*] ⟨*proof*⟩

sepref-definition *pop-min-impl* **is** *pop-min-op* :: $(rel^d \to_a prod\text{-}assn\ id\text{-}assn\ rel)$
⟨*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* :: $(rel^k \to_a id\text{-}assn)$
⟨*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* ≡ *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* ≡ *do* {
q ← *nfoldli l* (λ-. *True*) (λ*x q*. *mop-mset-insert x q*) *heap-custom-empty*;
(*l*,*q*) ← *WHILET* (λ(*l*,*q*). ¬*op-mset-is-empty q*) (λ(*l*,*q*). *do* {
(*x*,*q*) ← *mop-prio-pop-min prio q*;
RETURN (*l*@[*x*],*q*)
}) (*op-arl-empty*,*q*);
RETURN l
}

context fixes *prio*:: *'a*::{*default*,*heap*} ⇒ *'b*::*linorder* **begin**
sepref-definition *sort-impl* **is**
sort-by-prio prio :: $(list\text{-}assn\ (id\text{-}assn::'a::\{default,heap\} \Rightarrow -))^k \to_a arl\text{-}assn\ id\text{-}assn$
⟨*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 K*⟧ ⟹ ⟨*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*)∈⟨*K*,*V*⟩*map-rel*
assumes *2*: *m k* = *Some v*
obtains *l w* **where** *n l* = *Some w* (*k*,*l*)∈*K* (*v*,*w*)∈*V*
⟨*proof*⟩

lemma *param-dom*[*param*]: $(dom,dom) \in \langle K,V\rangle$*map-rel* $\rightarrow \langle K\rangle$*set-rel*
$\langle proof\rangle$

3.8.2 Interface Type

sepref-decl-intf $('k,'v)$ *i-map* **is** $'k \rightharpoonup 'v$

lemma [*synth-rules*]: $[\![$*INTF-OF-REL K TYPE*$('k)$; *INTF-OF-REL V TYPE*$('v)]\!]$
$\Longrightarrow$ *INTF-OF-REL* $(\langle K,V\rangle$*map-rel*$)$ *TYPE*$(('k,'v)$ *i-map*$)$ $\langle proof\rangle$

3.8.3 Operations

sepref-decl-op *map-empty*: *Map.empty* :: $\langle K,V\rangle$*map-rel* $\langle proof\rangle$

sepref-decl-op *map-is-empty*: $(=)$ *Map.empty* :: $\langle K,V\rangle$*map-rel* $\rightarrow$ *bool-rel*
$\langle proof\rangle$

sepref-decl-op *map-update*: $\lambda k\ v\ m.\ m(k \mapsto v)$ :: $K \rightarrow V \rightarrow \langle K,V\rangle$*map-rel* $\rightarrow$ $\langle K,V\rangle$*map-rel*
where *single-valued K single-valued* (K^{-1})
$\langle proof\rangle$

sepref-decl-op *map-delete*: $\lambda k\ m.$ *fun-upd m k None* :: $K \rightarrow \langle K,V\rangle$*map-rel* $\rightarrow$ $\langle K,V\rangle$*map-rel*
where *single-valued K single-valued* (K^{-1})
$\langle proof\rangle$

sepref-decl-op *map-lookup*: $\lambda k\ (m::'k \rightharpoonup 'v).\ m\ k$ :: $K \rightarrow \langle K,V\rangle$*map-rel* $\rightarrow$ $\langle V\rangle$*option-rel*
$\langle proof\rangle$

lemma *in-dom-alt*: $k \in dom\ m \longleftrightarrow \neg$*is-None* $(m\ k)$ $\langle proof\rangle$

sepref-decl-op *map-contains-key*: $\lambda k\ m.\ k \in dom\ m$ :: $K \rightarrow \langle K,V\rangle$*map-rel* $\rightarrow$ *bool-rel*
$\langle proof\rangle$

3.8.4 Patterns

lemma *pat-map-empty*[*pat-rules*]: λ_2-. *None* $\equiv$ *op-map-empty* $\langle proof\rangle$

lemma *pat-map-is-empty*[*pat-rules*]:
$(=)$ \$*m*\$$(\lambda_2$-. *None*$)$ $\equiv$ *op-map-is-empty*\$*m*
$(=)$ \$$(\lambda_2$-. *None*$)$\$*m* $\equiv$ *op-map-is-empty*\$*m*
$(=)$ \$$(dom$m)$\${} $\equiv$ *op-map-is-empty*\$*m*
$(=)$ \${}\$$(dom$m)$ $\equiv$ *op-map-is-empty*\$*m*
$\langle proof\rangle$

lemma *pat-map-update*[*pat-rules*]:

fun-upd$m$k$(Some$v) ≡ op-map-update$'k$'v$'m
⟨*proof*⟩
lemma *pat-map-lookup*[*pat-rules*]: *m$k ≡ op-map-lookup$'k$'m*
⟨*proof*⟩

lemma *op-map-delete-pat*[*pat-rules*]:
(|`) $ *m* $ (*uminus* $ (*insert* $ *k* $ {})) ≡ *op-map-delete$'k$'m*
fun-upd$m$k$None ≡ op-map-delete$'k$'m
⟨*proof*⟩

lemma *op-map-contains-key*[*pat-rules*]:
(∈) $ *k* $ (*dom$m*) ≡ *op-map-contains-key$'k$'m*
Not$((=) $(m$k)$None) ≡ op-map-contains-key$'k$'m
⟨*proof*⟩

3.8.5 Parametricity

locale *map-custom-empty* =
fixes *op-custom-empty* :: *'k⇀'v*
assumes *op-custom-empty-def*: *op-custom-empty = op-map-empty*
begin
sepref-register *op-custom-empty* :: (*'kx*,*'vx*) *i-map*

lemma *fold-custom-empty*:
Map.empty = op-custom-empty
op-map-empty = op-custom-empty
mop-map-empty = RETURN op-custom-empty
⟨*proof*⟩
end

end

3.9 Priority Maps

theory *IICF-Prio-Map*
imports *IICF-Map*
begin

This interface inherits from maps, and adds some operations

lemma *uncurry-fun-rel-conv*:
(*uncurry f*, *uncurry g*) ∈ *A*×$_r$*B* → *R* ⟷ (*f*,*g*)∈*A*→*B*→*R*
⟨*proof*⟩

lemma *uncurry0-fun-rel-conv*:
(*uncurry0 f*, *uncurry0 g*) ∈ *unit-rel* → *R* ⟷ (*f*,*g*)∈*R*
⟨*proof*⟩

lemma *RETURN-rel-conv0*: (*RETURN f*, *RETURN g*)∈⟨*A*⟩*nres-rel* ⟷ (*f*,*g*)∈*A*
⟨*proof*⟩

lemma *RETURN-rel-conv1*: $(RETURN\ o\ f,\ RETURN\ o\ g)\in A \to \langle B\rangle nres\text{-}rel \longleftrightarrow (f,g)\in A\to B$
⟨*proof*⟩

lemma *RETURN-rel-conv2*: $(RETURN\ oo\ f,\ RETURN\ oo\ g)\in A \to B \to \langle R\rangle nres\text{-}rel \longleftrightarrow (f,g)\in A\to B\to R$
⟨*proof*⟩

lemma *RETURN-rel-conv3*: $(RETURN\ ooo\ f,\ RETURN\ ooo\ g)\in A\to B\to C \to \langle R\rangle nres\text{-}rel \longleftrightarrow (f,g)\in A\to B\to C\to R$
⟨*proof*⟩

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* :: $[\lambda((k,v),m).\ k\notin dom\ m]_f\ (K\times_r V)\times_r\langle K,V\rangle map\text{-}rel \to \langle K,V\rangle map\text{-}rel$
where *single-valued K single-valued* (K^{-1}) ⟨*proof*⟩

sepref-decl-op *map-update-ex*: *op-map-update* :: $[\lambda((k,v),m).\ k\in dom\ m]_f\ (K\times_r V)\times_r\langle K,V\rangle map\text{-}rel \to \langle K,V\rangle map\text{-}rel$
where *single-valued K single-valued* (K^{-1}) ⟨*proof*⟩

sepref-decl-op *map-delete-ex*: *op-map-delete* :: $[\lambda(k,m).\ k\in dom\ m]_f\ K\times_r\langle K,V\rangle map\text{-}rel \to \langle K,V\rangle map\text{-}rel$
where *single-valued K single-valued* (K^{-1}) ⟨*proof*⟩

context
fixes *prio* :: $'v \Rightarrow 'p$::*linorder*
begin
sepref-decl-op *pm-decrease-key*: *op-map-update*
:: $[\lambda((k,v),m).\ k\in dom\ m \wedge prio\ v \le prio\ (the\ (m\ k))]_f\ (K\times_r V)\times_r\langle K,V\rangle map\text{-}rel \to \langle K,(V::('v\times'v)\ set)\rangle map\text{-}rel$
where *single-valued K single-valued* (K^{-1}) *IS-BELOW-ID V*
⟨*proof*⟩

sepref-decl-op *pm-increase-key*: *op-map-update*
:: $[\lambda((k,v),m).\ k{\in}dom\ m \wedge prio\ v \geq prio\ (the\ (m\ k))]_f\ (K\times_r V)\times_r\langle K,V\rangle map\text{-}rel$
$\to \langle K,(V::('v\times'v)\ set)\rangle map\text{-}rel$
where *single-valued K single-valued* (K^{-1}) *IS-BELOW-ID V*
⟨*proof*⟩

lemma *IS-BELOW-ID-D*: $(a,b){\in}R \Longrightarrow IS\text{-}BELOW\text{-}ID\ R \Longrightarrow a{=}b$ ⟨*proof*⟩

sepref-decl-op *pm-peek-min*: $\lambda m.\ SPEC\ (\lambda(k,v).$
$m\ k = Some\ v \wedge (\forall k'\ v'.\ m\ k' = Some\ v' \longrightarrow prio\ v \leq prio\ v'))$
:: $[Not\ o\ op\text{-}map\text{-}is\text{-}empty]_f\ \langle K,V\rangle map\text{-}rel \to K\times_r(V::('v\times'v)\ set)$
where *IS-BELOW-ID V*
⟨*proof*⟩
apply1 (*intro exI conjI allI impI*; *assumption?*)
⟨*proof*⟩

sepref-decl-op *pm-pop-min*: $\lambda m.\ SPEC\ (\lambda((k,v),m').$
$m\ k = Some\ v$
$\wedge\ m' = op\text{-}map\text{-}delete\ k\ m$
$\wedge\ (\forall k'\ v'.\ m\ k' = Some\ v' \longrightarrow prio\ v \leq prio\ v')$
$) :: [Not\ o\ op\text{-}map\text{-}is\text{-}empty]_f\ \langle K,V\rangle map\text{-}rel \to (K\times_r(V::('v\times'v)\ set))\times_r\langle K,V\rangle map\text{-}rel$
where *single-valued K single-valued* (K^{-1}) *IS-BELOW-ID V*
⟨*proof*⟩
applyS *parametricity*
⟨*proof*⟩
end

end

3.10 Priority Maps implemented with List and Map

theory *IICF-Abs-Heapmap*
imports *IICF-Abs-Heap HOL−Library.Rewrite ../../Intf/IICF-Prio-Map*
begin

type-synonym $('k,'v)\ ahm = 'k\ list \times ('k \rightharpoonup 'v)$

3.10.1 Basic Setup

First, we define a mapping to list-based heaps

definition *hmr-α* :: $('k,'v)\ ahm \Rightarrow 'v\ heap$ **where**
$hmr\text{-}\alpha \equiv \lambda(pq,m).\ map\ (the\ o\ m)\ pq$

definition $hmr\text{-}invar \equiv \lambda(pq,m).\ distinct\ pq \wedge dom\ m = set\ pq$

definition $hmr\text{-}rel \equiv br\ hmr\text{-}\alpha\ hmr\text{-}invar$

lemmas *hmr-rel-defs* = *hmr-rel-def br-def hmr-α-def hmr-invar-def*

lemma *hmr-empty-invar*[*simp*]: *hmr-invar* ([],*Map.empty*)
⟨*proof*⟩

locale *hmstruct* = *h*: *heapstruct prio* **for** *prio* :: *'v* ⇒ *'b*::*linorder*
begin

Next, we define a mapping to priority maps.

definition *heapmap-α* :: (*'k*,*'v*) *ahm* ⇒ (*'k* ⇀ *'v*) **where**
heapmap-α ≡ λ(*pq*,*m*). *m*

definition *heapmap-invar* :: (*'k*,*'v*) *ahm* ⇒ *bool* **where**
heapmap-invar ≡ λ*hm*. *hmr-invar hm* ∧ *h.heap-invar* (*hmr-α hm*)

definition *heapmap-rel* ≡ *br heapmap-α heapmap-invar*

lemmas *heapmap-rel-defs* = *heapmap-rel-def br-def heapmap-α-def heapmap-invar-def*

lemma [*refine-dref-RELATES*]: *RELATES hmr-rel* ⟨*proof*⟩

lemma *h-heap-invarI*[*simp*]: *heapmap-invar hm* ⟹ *h.heap-invar* (*hmr-α hm*)

⟨*proof*⟩

lemma *hmr-invarI*[*simp*]: *heapmap-invar hm* ⟹ *hmr-invar hm*
⟨*proof*⟩

lemma *set-hmr-α*[*simp*]: *hmr-invar hm* ⟹ *set* (*hmr-α hm*) = *ran* (*heapmap-α hm*)
⟨*proof*⟩

lemma *in-h-hmr-α-conv*[*simp*]: *hmr-invar hm* ⟹ *x* ∈# *h.α* (*hmr-α hm*) ⟷ *x* ∈ *ran* (*heapmap-α hm*)
⟨*proof*⟩

3.10.2 Basic Operations

In this section, we define the basic operations on heapmaps, and their relations to heaps and maps.

Length

Length of the list that represents the heap

definition *hm-length* :: (*'k*,*'v*) *ahm* ⇒ *nat* **where**
hm-length ≡ λ(*pq*,-). *length pq*

lemma *hm-length-refine*: (*hm-length*, *length*) ∈ *hmr-rel* → *nat-rel*
⟨*proof*⟩

lemma *hm-length-hmr-α*[*simp*]: *length* (*hmr-α hm*) = *hm-length hm*
⟨*proof*⟩

lemmas [*refine*] = *hm-length-refine*[*param-fo*]

Valid

Check whether index is valid

definition *hm-valid hm i* ≡ *i*>*0* ∧ *i*≤ *hm-length hm*

lemma *hm-valid-refine*: (*hm-valid*,*h.valid*)∈*hmr-rel* → *nat-rel* → *bool-rel*
⟨*proof*⟩

lemma *hm-valid-hmr-α*[*simp*]: *h.valid* (*hmr-α hm*) = *hm-valid hm*
⟨*proof*⟩

Key-Of

definition *hm-key-of* :: (*'k*,*'v*) *ahm* ⇒ *nat* ⇒ *'k* **where**
hm-key-of ≡ λ(*pq*,*m*) *i*. *pq*!(*i* − *1*)

definition *hm-key-of-op* :: (*'k*,*'v*) *ahm* ⇒ *nat* ⇒ *'k nres* **where**
hm-key-of-op ≡ λ(*pq*,*m*) *i*. *ASSERT* (*i*>*0*) ≫ *mop-list-get pq* (*i* − *1*)

lemma *hm-key-of-op-unfold*:
shows *hm-key-of-op hm i* = *ASSERT* (*hm-valid hm i*) ≫ *RETURN* (*hm-key-of hm i*)
⟨*proof*⟩

lemma *val-of-hmr-α*[*simp*]: *hm-valid hm i* ⟹ *h.val-of* (*hmr-α hm*) *i*
= *the* (*heapmap-α hm* (*hm-key-of hm i*))
⟨*proof*⟩

lemma *hm-α-key-ex*[*simp*]:
⟦*hmr-invar hm*; *hm-valid hm i*⟧ ⟹ (*heapmap-α hm* (*hm-key-of hm i*) ≠ *None*)
⟨*proof*⟩

Lookup

abbreviation (*input*) *hm-lookup* **where** *hm-lookup* ≡ *heapmap-α*

definition *hm-the-lookup-op hm k* ≡
ASSERT (*heapmap-α hm k* ≠ *None* ∧ *hmr-invar hm*)

$\gg$ *RETURN* (*the* (*heapmap-α hm k*))

Exchange

Exchange two indices

definition *hm-exch-op* ≡ λ(*pq*,*m*) *i j*. *do* {
ASSERT (*hm-valid* (*pq*,*m*) *i*);
ASSERT (*hm-valid* (*pq*,*m*) *j*);
ASSERT (*hmr-invar* (*pq*,*m*));
pq ← *mop-list-swap pq* (*i* − *1*) (*j* − *1*);
RETURN (*pq*,*m*)
}

lemma *hm-exch-op-invar*: *hm-exch-op hm i j* ≤$_n$ *SPEC hmr-invar*
⟨*proof*⟩

lemma *hm-exch-op-refine*: (*hm-exch-op*,*h.exch-op*) ∈ *hmr-rel* → *nat-rel* → *nat-rel* → ⟨*hmr-rel*⟩*nres-rel*
⟨*proof*⟩

lemmas *hm-exch-op-refine′*[*refine*] = *hm-exch-op-refine*[*param-fo*, *THEN nres-relD*]

definition *hm-exch* :: (′*k*,′*v*) *ahm* ⇒ *nat* ⇒ *nat* ⇒ (′*k*,′*v*) *ahm*
where *hm-exch* ≡ λ(*pq*,*m*) *i j*. (*swap pq* (*i*−*1*) (*j*−*1*),*m*)

lemma *hm-exch-op-α-correct*: *hm-exch-op hm i j* ≤$_n$ *SPEC* (λ*hm′*.
hm-valid hm i ∧ *hm-valid hm j* ∧ *hm′*=*hm-exch hm i j*
)
⟨*proof*⟩

lemma *hm-exch-α*[*simp*]: *heapmap-α* (*hm-exch hm i j*) = (*heapmap-α hm*)
⟨*proof*⟩
lemma *hm-exch-valid*[*simp*]: *hm-valid* (*hm-exch hm i j*) = *hm-valid hm*
⟨*proof*⟩
lemma *hm-exch-length*[*simp*]: *hm-length* (*hm-exch hm i j*) = *hm-length hm*
⟨*proof*⟩

lemma *hm-exch-same*[*simp*]: *hm-exch hm i i* = *hm*
⟨*proof*⟩

lemma *hm-key-of-exch-conv*[*simp*]:
⟦*hm-valid hm i*; *hm-valid hm j*; *hm-valid hm k*⟧ ⟹
hm-key-of (*hm-exch hm i j*) *k* = (
if k=*i then hm-key-of hm j*
else if k=*j then hm-key-of hm i*
else hm-key-of hm k
)

⟨*proof*⟩

lemma *hm-key-of-exch-matching*[*simp*]:
⟦*hm-valid hm i*; *hm-valid hm j*⟧ ⟹ *hm-key-of* (*hm-exch hm i j*) *i* = *hm-key-of hm j*
⟦*hm-valid hm i*; *hm-valid hm j*⟧ ⟹ *hm-key-of* (*hm-exch hm i j*) *j* = *hm-key-of hm i*
⟨*proof*⟩

Index

Obtaining the index of a key

definition *hm-index* ≡ λ(*pq*,*m*) *k*. *index pq k* + *1*

lemma *hm-index-valid*[*simp*]: ⟦*hmr-invar hm*; *heapmap-α hm k* ≠ *None*⟧ ⟹ *hm-valid hm* (*hm-index hm k*)
⟨*proof*⟩

lemma *hm-index-key-of*[*simp*]: ⟦*hmr-invar hm*; *heapmap-α hm k* ≠ *None*⟧ ⟹ *hm-key-of hm* (*hm-index hm k*) = *k*
⟨*proof*⟩

definition *hm-index-op* ≡ λ(*pq*,*m*) *k*.
do {
ASSERT (*hmr-invar* (*pq*,*m*) ∧ *heapmap-α* (*pq*,*m*) *k* ≠ *None*);
i ← *mop-list-index pq k*;
RETURN (*i*+*1*)
}

lemma *hm-index-op-correct*:
assumes *hmr-invar hm*
assumes *heapmap-α hm k* ≠ *None*
shows *hm-index-op hm k* ≤ *SPEC* (λ*r*. *r*= *hm-index hm k*)
⟨*proof*⟩
lemmas [*refine-vcg*] = *hm-index-op-correct*

Update

Updating the heap at an index

definition *hm-update-op* :: ('*k*,'*v*) *ahm* ⇒ *nat* ⇒ '*v* ⇒ ('*k*,'*v*) *ahm nres* **where**
hm-update-op ≡ λ(*pq*,*m*) *i v*. *do* {
ASSERT (*hm-valid* (*pq*,*m*) *i* ∧ *hmr-invar* (*pq*,*m*));
k ← *mop-list-get pq* (*i* − *1*);
RETURN (*pq*, *m*(*k* ↦ *v*))
}

lemma *hm-update-op-invar*: *hm-update-op hm k v* ≤$_n$ *SPEC hmr-invar*

⟨*proof*⟩

lemma *hm-update-op-refine*: (*hm-update-op*, *h.update-op*) ∈ *hmr-rel* → *nat-rel* → *Id* → ⟨*hmr-rel*⟩*nres-rel*
⟨*proof*⟩

lemmas [*refine*] = *hm-update-op-refine*[*param-fo*, *THEN nres-relD*]

lemma *hm-update-op-α-correct*:
assumes *hmr-invar hm*
assumes *heapmap-α hm k* ≠ *None*
shows *hm-update-op hm* (*hm-index hm k*) *v* ≤$_n$ *SPEC* (λ*hm′*. *heapmap-α hm′* = (*heapmap-α hm*)(*k*↦*v*))
⟨*proof*⟩

Butlast

Remove last element

definition *hm-butlast-op* :: (′*k*,′*v*) *ahm* ⇒ (′*k*,′*v*) *ahm nres* **where**
hm-butlast-op ≡ λ(*pq*,*m*). *do* {
ASSERT (*hmr-invar* (*pq*,*m*));
k ← *mop-list-get pq* (*length pq* − *1*);
pq ← *mop-list-butlast pq*;
let m = *m*(*k*:=*None*);
RETURN (*pq*,*m*)
}

lemma *hm-butlast-op-refine*: (*hm-butlast-op*, *h.butlast-op*) ∈ *hmr-rel* → ⟨*hmr-rel*⟩*nres-rel*
⟨*proof*⟩

lemmas [*refine*] = *hm-butlast-op-refine*[*param-fo*, *THEN nres-relD*]

lemma *hm-butlast-op-α-correct*: *hm-butlast-op hm* ≤$_n$ *SPEC* (
λ*hm′*. *heapmap-α hm′* = (*heapmap-α hm*)(*hm-key-of hm* (*hm-length hm*) := *None*))
⟨*proof*⟩

Append

Append new element at end of heap

definition *hm-append-op* :: (′*k*,′*v*) *ahm* ⇒ ′*k* ⇒ ′*v* ⇒ (′*k*,′*v*) *ahm nres*
where *hm-append-op* ≡ λ(*pq*,*m*) *k v*. *do* {
ASSERT (*k* ∉ *dom m*);
ASSERT (*hmr-invar* (*pq*,*m*));
pq ← *mop-list-append pq k*;
let m = *m* (*k* ↦ *v*);
RETURN (*pq*,*m*)
}

lemma *hm-append-op-invar*: *hm-append-op hm k v* $\leq_n$ *SPEC hmr-invar*
⟨*proof*⟩

lemma *hm-append-op-refine*: ⟦ *heapmap-α hm k* = *None*; (*hm*,*h*)∈*hmr-rel* ⟧
⟹ (*hm-append-op hm k v*, *h.append-op h v*) ∈ ⟨*hmr-rel*⟩*nres-rel*
⟨*proof*⟩

lemmas *hm-append-op-refine'*[*refine*] = *hm-append-op-refine*[*param-fo*, *THEN nres-relD*]

lemma *hm-append-op-α-correct*:
hm-append-op hm k v $\leq_n$ *SPEC* (λ*hm'*. *heapmap-α hm'* = (*heapmap-α hm*) (*k* ↦ *v*))
⟨*proof*⟩

3.10.3 Auxiliary Operations

Auxiliary operations on heapmaps, which are derived from the basic operations, but do not correspond to operations of the priority map interface

We start with some setup

lemma *heapmap-hmr-relI*: (*hm*,*h*)∈*heapmap-rel* ⟹ (*hm*,*hmr-α hm*) ∈ *hmr-rel*

⟨*proof*⟩

lemma *heapmap-hmr-relI'*: *heapmap-invar hm* ⟹ (*hm*,*hmr-α hm*) ∈ *hmr-rel*

⟨*proof*⟩

The basic principle how we prove correctness of our operations: Invariant preservation is shown by relating the operations to operations on heaps. Then, only correctness on the abstraction remains to be shown, assuming the operation does not fail.

lemma *heapmap-nres-relI'*:
assumes *hm* ≤ ⇓*hmr-rel h'*
assumes *h'* ≤ *SPEC* (*h.heap-invar*)
assumes *hm* $\leq_n$ *SPEC* (λ*hm'*. *RETURN* (*heapmap-α hm'*) ≤ *h*)
shows *hm* ≤ ⇓*heapmap-rel h*
⟨*proof*⟩

lemma *heapmap-nres-relI''*:
assumes *hm* ≤ ⇓*hmr-rel h'*
assumes *h'* ≤ *SPEC* Φ
assumes ⋀*h'*. Φ *h'* ⟹ *h.heap-invar h'*
assumes *hm* $\leq_n$ *SPEC* (λ*hm'*. *RETURN* (*heapmap-α hm'*) ≤ *h*)
shows *hm* ≤ ⇓*heapmap-rel h*
⟨*proof*⟩

Val-of

Indexing into the heap

definition *hm-val-of-op* :: $('k,'v)$ *ahm* $\Rightarrow$ *nat* $\Rightarrow$ $'v$ *nres* **where**
hm-val-of-op $\equiv$ λ*hm i. do* {
k $\leftarrow$ *hm-key-of-op hm i*;
v $\leftarrow$ *hm-the-lookup-op hm k*;
RETURN v
}

lemma *hm-val-of-op-refine*: (*hm-val-of-op*,*h.val-of-op*) $\in$ (*hmr-rel* $\rightarrow$ *nat-rel* $\rightarrow$ $\langle$*Id*$\rangle$*nres-rel*)
$\langle$*proof*$\rangle$

lemmas [*refine*] = *hm-val-of-op-refine*[*param-fo*, *THEN nres-relD*]

Prio-of

Priority of key

definition *hm-prio-of-op h i* $\equiv$ *do* {*v* $\leftarrow$ *hm-val-of-op h i*; *RETURN* (*prio v*)}

lemma *hm-prio-of-op-refine*: (*hm-prio-of-op*, *h.prio-of-op*) $\in$ *hmr-rel* $\rightarrow$ *nat-rel* $\rightarrow$ $\langle$*Id*$\rangle$*nres-rel*
$\langle$*proof*$\rangle$

lemmas *hm-prio-of-op-refine'*[*refine*] = *hm-prio-of-op-refine*[*param-fo*, *THEN nres-relD*]

Swim

definition *hm-swim-op* :: $('k,'v)$ *ahm* $\Rightarrow$ *nat* $\Rightarrow$ $('k,'v)$ *ahm nres* **where**
hm-swim-op h i $\equiv$ *do* {
RECT (λ*swim* (*h*,*i*). *do* {
ASSERT (*hm-valid h i* $\wedge$ *h.swim-invar* (*hmr-*α *h*) *i*);
if hm-valid h (*h.parent i*) *then do* {
ppi $\leftarrow$ *hm-prio-of-op h* (*h.parent i*);
pi $\leftarrow$ *hm-prio-of-op h i*;
if ($\neg$*ppi* $\leq$ *pi*) *then do* {
h $\leftarrow$ *hm-exch-op h i* (*h.parent i*);
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*

⟨*proof*⟩

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*) ⟹ *hm-valid hm i* ∧ *h.swim-invar* (*hmr-α hm*) *i*
⟨*proof*⟩

lemma *hm-swim-op-α-correct*: *hm-swim-op hm i* ≤$_n$ *SPEC* (λ*hm'*. *heapmap-α hm'* = *heapmap-α hm*)
⟨*proof*⟩

Sink

definition *hm-sink-op*
where
hm-sink-op h k ≡ *RECT* (λ*D* (*h*,*k*). *do* {
ASSERT (*k*>0 ∧ *k*≤*hm-length h*);
let len = *hm-length h*;
if (2∗*k* ≤ *len*) *then do* {
let j = 2∗*k*;
pj ← *hm-prio-of-op h j*;

j ← (
if j<*len then do* {
psj ← *hm-prio-of-op h* (*Suc j*);
if pj>*psj then RETURN* (*j*+1) *else RETURN j*
} *else RETURN j*);

pj ← *hm-prio-of-op h j*;
pk ← *hm-prio-of-op h k*;
if (*pk* > *pj*) *then do* {
h ← *hm-exch-op h k j*;
D (*h*,*j*)
} *else*
RETURN h
} *else RETURN h*
}) (*h*,*k*)

lemma *hm-sink-op-refine*: (*hm-sink-op*, *h.sink-op*) ∈ *hmr-rel* → *nat-rel* → ⟨*hmr-rel*⟩*nres-rel*
⟨*proof*⟩

lemmas *hm-sink-op-refine'*[*refine*] = *hm-sink-op-refine*[*param-fo*, *THEN nres-relD*]

lemma *hm-sink-op-nofail-imp-valid*: *nofail* (*hm-sink-op hm i*) ⟹ *hm-valid hm i*
⟨*proof*⟩

lemma *hm-sink-op-α-correct*: *hm-sink-op hm i* $\leq_n$ *SPEC* (λ*hm′*. *heapmap-α hm′* = *heapmap-α hm*)
⟨*proof*⟩

Repair

definition *hm-repair-op hm i* ≡ *do* {
hm ← *hm-sink-op hm i*;
hm ← *hm-swim-op hm i*;
RETURN hm
}

lemma *hm-repair-op-refine*: (*hm-repair-op*, *h.repair-op*) ∈ *hmr-rel* → *nat-rel* → ⟨*hmr-rel*⟩*nres-rel*
⟨*proof*⟩

lemmas *hm-repair-op-refine′*[*refine*] = *hm-repair-op-refine*[*param-fo*, *THEN nres-relD*]

lemma *hm-repair-op-α-correct*: *hm-repair-op hm i* $\leq_n$ *SPEC* (λ*hm′*. *heapmap-α hm′* = *heapmap-α hm*)
⟨*proof*⟩

3.10.4 Operations

In this section, we define the operations that implement the priority-map interface

Empty

definition *hm-empty-op* :: (*′k*,*′v*) *ahm nres*
where *hm-empty-op* ≡ *RETURN* ([],*Map.empty*)

lemma *hm-empty-aref*: (*hm-empty-op*,*RETURN op-map-empty*) ∈ ⟨*heapmap-rel*⟩*nres-rel*

⟨*proof*⟩

Insert

definition *hm-insert-op* :: *′k* ⇒ *′v* ⇒ (*′k*,*′v*) *ahm* ⇒ (*′k*,*′v*) *ahm nres* **where**
hm-insert-op ≡ λ*k v h*. *do* {
ASSERT (*h.heap-invar* (*hmr-α h*));
h ← *hm-append-op h k v*;
let l = *hm-length h*;
h ← *hm-swim-op h l*;
RETURN h
}

lemma *hm-insert-op-refine*[*refine*]: ⟦ *heapmap-α hm k = None*; (*hm*,*h*)∈*hmr-rel* ⟧ ⟹
hm-insert-op k v hm ≤ ⇓*hmr-rel* (*h.insert-op v h*)
⟨*proof*⟩

lemma *hm-insert-op-aref*:
(*hm-insert-op*,*mop-map-update-new*) ∈ *Id* → *Id* → *heapmap-rel* → ⟨*heapmap-rel*⟩*nres-rel*
⟨*proof*⟩

Is-Empty

lemma *hmr-α-empty-iff*[*simp*]:
hmr-invar hm ⟹ *hmr-α hm* = [] ⟷ *heapmap-α hm* = *Map.empty*
⟨*proof*⟩

definition *hm-is-empty-op* :: (′*k*,′*v*) *ahm* ⇒ *bool nres* **where**
hm-is-empty-op ≡ λ*hm*. *do* {
ASSERT (*hmr-invar hm*);
let l = *hm-length hm*;
RETURN (*l*=*0*)
}

lemma *hm-is-empty-op-refine*: (*hm-is-empty-op*, *h.is-empty-op*) ∈ *hmr-rel* → ⟨*bool-rel*⟩*nres-rel*
⟨*proof*⟩

lemma *hm-is-empty-op-aref*: (*hm-is-empty-op*, *RETURN o op-map-is-empty*) ∈ *heapmap-rel* → ⟨*bool-rel*⟩*nres-rel*
⟨*proof*⟩

Lookup

definition *hm-lookup-op* :: ′*k* ⇒ (′*k*,′*v*) *ahm* ⇒ ′*v option nres*
where *hm-lookup-op* ≡ λ*k hm*. *ASSERT* (*heapmap-invar hm*) ≫ *RETURN* (*hm-lookup hm k*)

lemma *hm-lookup-op-aref*: (*hm-lookup-op*,*RETURN oo op-map-lookup*) ∈ *Id* → *heapmap-rel* → ⟨⟨*Id*⟩*option-rel*⟩*nres-rel*
⟨*proof*⟩

Contains-Key

definition *hm-contains-key-op* ≡ λ*k* (*pq*,*m*). *ASSERT* (*heapmap-invar* (*pq*,*m*)) ≫ *RETURN* (*k*∈*dom m*)
lemma *hm-contains-key-op-aref*: (*hm-contains-key-op*,*RETURN oo op-map-contains-key*) ∈ *Id* → *heapmap-rel* → ⟨*bool-rel*⟩*nres-rel*
⟨*proof*⟩

Decrease-Key

definition *hm-decrease-key-op* $\equiv \lambda k\ v\ hm.$ *do* {
ASSERT (*heapmap-invar hm*);
ASSERT (*heapmap-α hm k* $\neq$ *None* $\wedge$ *prio v* $\leq$ *prio* (*the* (*heapmap-α hm k*)));
i $\leftarrow$ *hm-index-op hm k*;
hm $\leftarrow$ *hm-update-op hm i v*;
hm-swim-op hm i
}

definition (**in** *heapstruct*) *decrease-key-op i v h* $\equiv$ *do* {
ASSERT (*valid h i* $\wedge$ *prio v* $\leq$ *prio-of h i*);
h $\leftarrow$ *update-op h i v*;
swim-op h i
}

lemma (**in** *heapstruct*) *decrease-key-op-invar*:
$[\![$*heap-invar h*; *valid h i*; *prio v* $\leq$ *prio-of h i*$]\!] \Longrightarrow$ *decrease-key-op i v h* $\leq$ *SPEC heap-invar*
⟨*proof*⟩

lemma *index-op-inline-refine*:
assumes *heapmap-invar hm*
assumes *heapmap-α hm k* $\neq$ *None*
assumes *f* (*hm-index hm k*) $\leq m$
shows *do* {*i* $\leftarrow$ *hm-index-op hm k*; *f i*} $\leq m$
⟨*proof*⟩

lemma *hm-decrease-key-op-refine*:
$[\![$(*hm*,*h*)$\in$*hmr-rel*; (*hm*,*m*)$\in$*heapmap-rel*; *m k* = *Some v*$]\!]$
$\Longrightarrow$ *hm-decrease-key-op k v hm* $\leq\Downarrow$*hmr-rel* (*h.decrease-key-op* (*hm-index hm k*) *v h*)
⟨*proof*⟩

lemma *hm-index-op-inline-leof*:
assumes *f* (*hm-index hm k*) $\leq_n m$
shows *do* {*i* $\leftarrow$ *hm-index-op hm k*; *f i*} $\leq_n m$
⟨*proof*⟩

lemma *hm-decrease-key-op-α-correct*:
heapmap-invar hm $\Longrightarrow$ *hm-decrease-key-op k v hm* $\leq_n$ *SPEC* ($\lambda hm'$. *heapmap-α hm'* = *heapmap-α hm*($k \mapsto v$))
⟨*proof*⟩

lemma *hm-decrease-key-op-aref*:
(*hm-decrease-key-op*, *PR-CONST* (*mop-pm-decrease-key prio*)) $\in$ *Id* $\rightarrow$ *Id* $\rightarrow$ *heapmap-rel* $\rightarrow$ ⟨*heapmap-rel*⟩*nres-rel*
⟨*proof*⟩

Increase-Key

definition *hm-increase-key-op* $\equiv \lambda k\ v\ hm.$ *do* {
ASSERT (*heapmap-invar hm*);
ASSERT (*heapmap-α hm k* $\neq$ *None* $\wedge$ *prio v* $\geq$ *prio* (*the* (*heapmap-α hm k*)));
i $\leftarrow$ *hm-index-op hm k*;
hm $\leftarrow$ *hm-update-op hm i v*;
hm-sink-op hm i
}

definition (**in** *heapstruct*) *increase-key-op i v h* $\equiv$ *do* {
ASSERT (*valid h i* $\wedge$ *prio v* $\geq$ *prio-of h i*);
h $\leftarrow$ *update-op h i v*;
sink-op h i
}

lemma (**in** *heapstruct*) *increase-key-op-invar*:
⟦*heap-invar h*; *valid h i*; *prio v* $\geq$ *prio-of h i*⟧ $\Longrightarrow$ *increase-key-op i v h* $\leq$ *SPEC heap-invar*
⟨*proof*⟩

lemma *hm-increase-key-op-refine*:
⟦(*hm*,*h*)$\in$*hmr-rel*; (*hm*,*m*)$\in$*heapmap-rel*; *m k* = *Some v*′⟧
$\Longrightarrow$ *hm-increase-key-op k v hm* $\leq\Downarrow$*hmr-rel* (*h.increase-key-op* (*hm-index hm k*) *v h*)
⟨*proof*⟩

lemma *hm-increase-key-op-α-correct*:
heapmap-invar hm $\Longrightarrow$ *hm-increase-key-op k v hm* $\leq_n$ *SPEC* (λ*hm*′. *heapmap-α hm*′ = *heapmap-α hm*(*k*$\mapsto$*v*))
⟨*proof*⟩

lemma *hm-increase-key-op-aref*:
(*hm-increase-key-op*, *PR-CONST* (*mop-pm-increase-key prio*)) $\in$ *Id* $\rightarrow$ *Id* $\rightarrow$ *heapmap-rel* $\rightarrow$ ⟨*heapmap-rel*⟩*nres-rel*
⟨*proof*⟩

Change-Key

definition *hm-change-key-op* $\equiv \lambda k\ v\ hm.$ *do* {
ASSERT (*heapmap-invar hm*);
ASSERT (*heapmap-α hm k* $\neq$ *None*);
i $\leftarrow$ *hm-index-op hm k*;
hm $\leftarrow$ *hm-update-op hm i v*;
hm-repair-op hm i
}

definition (**in** *heapstruct*) *change-key-op i v h* $\equiv$ *do* {
ASSERT (*valid h i*);

$h \leftarrow$ *update-op h i v*;
repair-op h i
}

lemma (**in** *heapstruct*) *change-key-op-invar*:
⟦*heap-invar h*; *valid h i*⟧ ⟹ *change-key-op i v h* ≤ *SPEC heap-invar*
⟨*proof*⟩

lemma *hm-change-key-op-refine*:
⟦(*hm*,*h*)∈*hmr-rel*; (*hm*,*m*)∈*heapmap-rel*; *m k* = *Some v'*⟧
⟹ *hm-change-key-op k v hm* ≤⇓*hmr-rel* (*h.change-key-op* (*hm-index hm k*) *v h*)
⟨*proof*⟩

lemma *hm-change-key-op-α-correct*:
heapmap-invar hm ⟹ *hm-change-key-op k v hm* ≤$_n$ *SPEC* (λ*hm'*. *heapmap-α hm'* = *heapmap-α hm*(*k*↦*v*))
⟨*proof*⟩

lemma *hm-change-key-op-aref*:
(*hm-change-key-op*, *mop-map-update-ex*) ∈ *Id* → *Id* → *heapmap-rel* → ⟨*heapmap-rel*⟩*nres-rel*
⟨*proof*⟩

Set

Realized as generic algorithm!

lemma (**in** −) *op-pm-set-gen-impl*: *RETURN ooo op-map-update* = (λ*k v m*. *do* {
c ← *RETURN* (*op-map-contains-key k m*);
if c then
mop-map-update-ex k v m
else
mop-map-update-new k v m
})
⟨*proof*⟩

definition *hm-set-op k v hm* ≡ *do* {
c ← *hm-contains-key-op k hm*;
if c then
hm-change-key-op k v hm
else
hm-insert-op k v hm
}

lemma *hm-set-op-aref*:
(*hm-set-op*, *RETURN ooo op-map-update*) ∈ *Id* → *Id* → *heapmap-rel* → ⟨*heapmap-rel*⟩*nres-rel*
⟨*proof*⟩

Pop-Min

definition *hm-pop-min-op* :: ('k,'v) *ahm* ⇒ (('k×'v) × ('k,'v) *ahm*) *nres* **where**
hm-pop-min-op hm ≡ *do* {
ASSERT (*heapmap-invar hm*);
ASSERT (*hm-valid hm 1*);
k ← *hm-key-of-op hm 1*;
v ← *hm-the-lookup-op hm k*;
let l = *hm-length hm*;
hm ← *hm-exch-op hm 1 l*;
hm ← *hm-butlast-op hm*;

if (*l*≠*1*) *then do* {
hm ← *hm-sink-op hm 1*;
RETURN ((*k*,*v*),*hm*)
} *else RETURN* ((*k*,*v*),*hm*)
}

lemma *hm-pop-min-op-refine*:
(*hm-pop-min-op*, *h.pop-min-op*) ∈ *hmr-rel* → ⟨*UNIV* ×$_r$ *hmr-rel*⟩*nres-rel*
⟨*proof*⟩

We demonstrate two different approaches for proving correctness here. The first approach uses the relation to plain heaps only to establish the invariant.

The second approach also uses the relation to heaps to establish correctness of the result.

The first approach seems to be more robust against badly set up simpsets, which may be the case in early stages of development.

Assuming a working simpset, the second approach may be less work, and the proof may look more elegant.

First approach Transfer heapmin-property to heapmap-domain

lemma *heapmap-min-prop*:
assumes *INV*: *heapmap-invar hm*
assumes *V'*: *heapmap-α hm k* = *Some v'*
assumes *NE*: *hm-valid hm* (*Suc 0*)
shows *prio* (*the* (*heapmap-α hm* (*hm-key-of hm* (*Suc 0*)))) ≤ *prio v'*
⟨*proof*⟩

With the above lemma, the correctness proof is straightforward

lemma *hm-pop-min-α-correct*: *hm-pop-min-op hm* ≤$_n$ *SPEC* (λ((*k*,*v*),*hm'*).
heapmap-α hm k = *Some v*
∧ *heapmap-α hm'* = (*heapmap-α hm*)(*k*:=*None*)
∧ (∀ *k' v'*. *heapmap-α hm k'* = *Some v'* ⟶ *prio v* ≤ *prio v'*))
⟨*proof*⟩

lemma *heapmap-nres-rel-prodI*:

assumes $hmx \leq \Downarrow(UNIV \times_r hmr\text{-}rel)\ h'x$
assumes $h'x \leq SPEC\ (\lambda(\text{-},h').\ h.heap\text{-}invar\ h')$
assumes $hmx \leq_n SPEC\ (\lambda(r,hm').\ RETURN\ (r,heapmap\text{-}\alpha\ hm') \leq \Downarrow(R\times_r Id)\ hx)$
shows $hmx \leq \Downarrow(R\times_r heapmap\text{-}rel)\ hx$
⟨*proof*⟩

lemma *hm-pop-min-op-aref*: $(hm\text{-}pop\text{-}min\text{-}op,\ PR\text{-}CONST\ (mop\text{-}pm\text{-}pop\text{-}min\ prio)) \in heapmap\text{-}rel \to \langle(Id\times_r Id)\times_r heapmap\text{-}rel\rangle nres\text{-}rel$
⟨*proof*⟩

Second approach **definition** *hm-kv-of-op hm i* ≡ *do* {
ASSERT (*hm-valid hm i* ∧ *hmr-invar hm*);
k ← *hm-key-of-op hm i*;
v ← *hm-the-lookup-op hm k*;
RETURN (*k*, *v*)
}

definition *kvi-rel hm i* ≡ {((*k*,*v*),*v*) | *k v*. *hm-key-of hm i* = *k*}

lemma *hm-kv-op-refine*[*refine*]:
assumes (*hm*,*h*)∈*hmr-rel*
shows *hm-kv-of-op hm i* ≤ ⇓(*kvi-rel hm i*) (*h.val-of-op h i*)
⟨*proof*⟩

definition *hm-pop-min-op'* :: ('*k*,'*v*) *ahm* ⇒ (('*k*×'*v*) × ('*k*,'*v*) *ahm*) *nres* **where**
hm-pop-min-op' hm ≡ *do* {
ASSERT (*heapmap-invar hm*);
ASSERT (*hm-valid hm 1*);
kv ← *hm-kv-of-op hm 1*;
let l = *hm-length hm*;
hm ← *hm-exch-op hm 1 l*;
hm ← *hm-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'*:
⟦ (*hm*,*h*)∈*hmr-rel* ⟧ ⟹ *hm-pop-min-op' hm* ≤ ⇓(*kvi-rel hm 1* ×$_r$ *hmr-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 $\bigwedge h'\ r.\ \Phi\ (r,h') \Longrightarrow h.heap\text{-}invar\ h'$
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*:
$(\exists\, r'.\ (r,r')\in kvi\text{-}rel\ hm\ i \wedge \Phi\ r') \longleftrightarrow (fst\ r = hm\text{-}key\text{-}of\ hm\ i \wedge \Phi\ (snd\ r))$
⟨*proof*⟩

lemma *hm-pop-min-aref'*: $(hm\text{-}pop\text{-}min\text{-}op', mop\text{-}pm\text{-}pop\text{-}min\ prio) \in heapmap\text{-}rel \to \langle(Id\times_r Id) \times_r heapmap\text{-}rel\rangle nres\text{-}rel$
⟨*proof*⟩

Remove

definition *hm-remove-op k hm* ≡ *do* {
ASSERT (*heapmap-invar hm*);
ASSERT ($k \in dom\ (heapmap\text{-}\alpha\ hm)$);
$i \leftarrow hm\text{-}index\text{-}op\ hm\ k$;
let $l = hm\text{-}length\ hm$;
$hm \leftarrow hm\text{-}exch\text{-}op\ hm\ i\ l$;
$hm \leftarrow hm\text{-}butlast\text{-}op\ hm$;
if $i \neq l$ *then*
hm-repair-op hm i
else
RETURN hm
}

definition (**in** *heapstruct*) *remove-op i h* ≡ *do* {
ASSERT (*heap-invar h*);
ASSERT (*valid h i*);
let $l = length\ h$;
$h \leftarrow exch\text{-}op\ h\ i\ l$;
$h \leftarrow butlast\text{-}op\ h$;
if $i \neq l$ *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*:
⟦ *heap-invar h*; *valid h i* ⟧ ⟹ *remove-op i h* ≤ *SPEC heap-invar*
⟨*proof*⟩

lemma *hm-remove-op-refine*[*refine*]:
⟦ (*hm*,*m*)∈*heapmap-rel*; (*hm*,*h*)∈*hmr-rel*; *heapmap-α hm k* ≠ *None*⟧ ⟹
hm-remove-op k hm ≤ ⇓*hmr-rel* (*h.remove-op* (*hm-index hm k*) *h*)
⟨*proof*⟩

lemma *hm-remove-op-α-correct*:
hm-remove-op k hm ≤$_n$ *SPEC* (λ*hm'*. *heapmap-α hm'* = (*heapmap-α hm*)(*k*:=*None*))

⟨*proof*⟩

lemma *hm-remove-op-aref*:
(*hm-remove-op*,*mop-map-delete-ex*) ∈ *Id* → *heapmap-rel* → ⟨*heapmap-rel*⟩*nres-rel*
⟨*proof*⟩

Peek-Min

definition *hm-peek-min-op* :: ('*k*,'*v*) *ahm* ⇒ ('*k*×'*v*) *nres* **where**
hm-peek-min-op hm ≡ *hm-kv-of-op hm 1*

lemma *hm-peek-min-op-aref*:
(*hm-peek-min-op*, *PR-CONST* (*mop-pm-peek-min prio*)) ∈ *heapmap-rel* →
⟨*Id*×$_r$*Id*⟩*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* ≡ *p*↦$_a$*l*

lemma *is-array-precise*[*safe-constraint-rules*]: *precise is-array*
⟨*proof*⟩

definition *array-assn* **where** *array-assn A* ≡ *hr-comp is-array* (⟨*the-pure A*⟩*list-rel*)
lemmas [*safe-constraint-rules*] = *CN-FALSEI*[*of is-pure array-assn A* **for** *A*]

definition [*simp*,*code-unfold*]: *heap-array-empty* ≡ *Array.of-list* []
definition [*simp*,*code-unfold*]: *heap-array-set p i v* ≡ *Array.upd i v p*

context
notes [*fcomp-norm-unfold*] = *array-assn-def*[*symmetric*]
notes [*intro*!] = *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*)) ∈ *unit-assn*^{k} →$_a$ *is-array*
⟨*proof*⟩
sepref-decl-impl (*no-register*) *array-empty*: *array-empty-hnr-aux* ⟨*proof*⟩

lemma *array-replicate-hnr-aux*:
(*uncurry Array.new*, *uncurry* (*RETURN oo op-list-replicate*))
∈ *nat-assn*^{k} ∗$_a$ *id-assn*^{k} →$_a$ *is-array*
⟨*proof*⟩
sepref-decl-impl (*no-register*) *array-replicate*: *array-replicate-hnr-aux* ⟨*proof*⟩

definition [*simp*]: *op-array-replicate* ≡ *op-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*
⟨*proof*⟩
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*) ∈ (*list-assn id-assn*)^k →$_a$ *is-array*
⟨*proof*⟩
sepref-decl-impl (*no-register*) *array-of-list*: *array-of-list-hnr-aux* ⟨*proof*⟩

definition [*simp*]: *op-array-of-list* ≡ *op-list-copy*
sepref-register *op-array-of-list*
lemma *array-fold-custom-of-list*:
l = *op-array-of-list l*
op-list-copy = *op-array-of-list*
mop-list-copy = *RETURN o op-array-of-list*
⟨*proof*⟩
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*) ∈ *is-array*^{k} →$_a$ *is-array*
⟨*proof*⟩
sepref-decl-impl *array-copy*: *array-copy-hnr-aux* ⟨*proof*⟩

lemma *array-get-hnr-aux*: (*uncurry Array.nth*,*uncurry* (*RETURN oo op-list-get*)) ∈ [λ(*l*,*i*). *i*<*length l*]$_a$ *is-array*^{k} *∗*$_a$ *nat-assn*^{k} → *id-assn*
⟨*proof*⟩
sepref-decl-impl *array-get*: *array-get-hnr-aux* ⟨*proof*⟩

lemma *array-set-hnr-aux*: (*uncurry2 heap-array-set*,*uncurry2* (*RETURN ooo op-list-set*)) ∈ [λ((*l*,*i*),-). *i*<*length l*]$_a$ *is-array*^{d} *∗*$_a$ *nat-assn*^{k} *∗*$_a$ *id-assn*^{k} → *is-array*

⟨*proof*⟩
sepref-decl-impl *array-set*: *array-set-hnr-aux* ⟨*proof*⟩

lemma *array-length-hnr-aux*: (*Array.len*,*RETURN o op-list-length*) ∈ *is-array*^{k} →$_a$ *nat-assn*
⟨*proof*⟩
sepref-decl-impl *array-length*: *array-length-hnr-aux* ⟨*proof*⟩

end

definition [*simp*]: *op-array-empty* ≡ *op-list-empty*
interpretation *array*: *list-custom-empty array-assn A heap-array-empty op-array-empty*
⟨*proof*⟩

end
theory *IICF-MS-Array-List*
imports
../Intf/IICF-List
Separation-Logic-Imperative-HOL.Array-Blit
../../../Separation-Logic-Imperative-HOL/Examples/Default-Insts
begin

type-synonym *'a ms-array-list* = *'a Heap.array* × *nat*

definition *is-ms-array-list ms l* ≡ λ(*a*,*n*). ∃$_A$*l'*. *a* ↦$_a$ *l'* ∗ ↑(*n* ≤ *length l'* ∧ *l* = *take n l'* ∧ *ms*=*length l'*)

lemma *is-ms-array-list-prec*[*safe-constraint-rules*]: *precise* (*is-ms-array-list ms*)
⟨*proof*⟩

definition *marl-empty-sz maxsize* ≡ *do* {
a ← *Array.new maxsize default*;
return (*a*,*0*)

}

definition *marl-append* ≡ λ(*a*,*n*) *x*. *do* {
a ← *Array.upd n x a*;
return (*a*,*n*+*1*)
}

definition *marl-length* :: *'a*::*heap ms-array-list* ⇒ *nat Heap* **where**
marl-length ≡ λ(*a*,*n*). *return* (*n*)

definition *marl-is-empty* :: *'a*::*heap ms-array-list* ⇒ *bool Heap* **where**
marl-is-empty ≡ λ(*a*,*n*). *return* (*n*=*0*)

definition *marl-last* :: *'a*::*heap ms-array-list* ⇒ *'a Heap* **where**
marl-last ≡ λ(*a*,*n*). *do* {
Array.nth a (*n* − *1*)
}

definition *marl-butlast* :: *'a*::*heap ms-array-list* ⇒ *'a ms-array-list Heap* **where**
marl-butlast ≡ λ(*a*,*n*). *do* {
return (*a*,*n* − *1*)
}

definition *marl-get* :: *'a*::*heap ms-array-list* ⇒ *nat* ⇒ *'a Heap* **where**
marl-get ≡ λ(*a*,*n*) *i*. *Array.nth a i*

definition *marl-set* :: *'a*::*heap ms-array-list* ⇒ *nat* ⇒ *'a* ⇒ *'a ms-array-list Heap*
where
marl-set ≡ λ(*a*,*n*) *i x*. *do* { *a* ← *Array.upd i x a*; *return* (*a*,*n*)}

lemma *marl-empty-sz-rule*[*sep-heap-rules*]: < *emp* > *marl-empty-sz N* <*is-ms-array-list*
N []>
⟨*proof*⟩

lemma *marl-append-rule*[*sep-heap-rules*]: *length l* < *N* ⟹
< *is-ms-array-list N l a* >
marl-append a x
<λ*a*. *is-ms-array-list N* (*l*@[*x*]) *a* >*t*
⟨*proof*⟩

lemma *marl-length-rule*[*sep-heap-rules*]:
<*is-ms-array-list N l a*>
marl-length a
<λ*r*. *is-ms-array-list N l a* ∗ ↑(*r*=*length l*)>
⟨*proof*⟩

lemma *marl-is-empty-rule*[*sep-heap-rules*]:
<*is-ms-array-list N l a*>
marl-is-empty a

$<\lambda r.\ is\text{-}ms\text{-}array\text{-}list\ N\ l\ a * \uparrow(r \longleftrightarrow (l=[]))>$
$\langle proof \rangle$

lemma *marl-last-rule*[*sep-heap-rules*]:
$l \neq [] \Longrightarrow$
$<is\text{-}ms\text{-}array\text{-}list\ N\ l\ a>$
marl-last a
$<\lambda r.\ is\text{-}ms\text{-}array\text{-}list\ N\ l\ a * \uparrow(r = last\ l)>$
$\langle proof \rangle$

lemma *marl-butlast-rule*[*sep-heap-rules*]:
$l \neq [] \Longrightarrow$
$<is\text{-}ms\text{-}array\text{-}list\ N\ l\ a>$
marl-butlast a
$<is\text{-}ms\text{-}array\text{-}list\ N\ (butlast\ l)>_t$
$\langle proof \rangle$

lemma *marl-get-rule*[*sep-heap-rules*]:
$i < length\ l \Longrightarrow$
$<is\text{-}ms\text{-}array\text{-}list\ N\ l\ a>$
marl-get a i
$<\lambda r.\ is\text{-}ms\text{-}array\text{-}list\ N\ l\ a * \uparrow(r = l!i)>$
$\langle proof \rangle$

lemma *marl-set-rule*[*sep-heap-rules*]:
$i < length\ l \Longrightarrow$
$<is\text{-}ms\text{-}array\text{-}list\ N\ l\ a>$
marl-set a i x
$<is\text{-}ms\text{-}array\text{-}list\ N\ (l[i:=x])>$
$\langle proof \rangle$

definition *marl-assn N A* $\equiv$ *hr-comp* (*is-ms-array-list N*) ($\langle$*the-pure A*$\rangle$*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*
⟨*proof*⟩

lemma *marl-empty-hnr-aux*: (*uncurry0* (*marl-empty-sz N*), *uncurry0* (*RETURN op-list-empty*)) ∈ *unit-assn*^{k} →$_a$ *is-ms-array-list N*
⟨*proof*⟩
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*)))) ∈ *unit-assn*^{k} →$_a$ *marl-assn N A*
⟨*proof*⟩

lemma *marl-append-hnr-aux*: (*uncurry marl-append*,*uncurry* (*RETURN oo op-list-append*)) ∈ [λ(*l*,-). *length l*<*N*]$_a$ ((*is-ms-array-list N*)^d *∗*$_a$ *id-assn*^{k}) → *is-ms-array-list N*
⟨*proof*⟩
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*) ∈ (*is-ms-array-list N*)^k →$_a$ *nat-assn*
⟨*proof*⟩
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*) ∈ (*is-ms-array-list N*)^k →$_a$ *bool-assn*
⟨*proof*⟩
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*) ∈ [λ*x*. *x*≠[]]$_a$ (*is-ms-array-list N*)^k → *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*) ∈ [λ*x*. *x*≠[]]$_a$ (*is-ms-array-list N*)^d → (*is-ms-array-list N*)
⟨*proof*⟩
lemmas *marl-butlast-hnr*[*sepref-fr-rules*] = *marl-butlast-hnr-aux*[*FCOMP op-list-butlast.fref*[*of the-pure A* **for** *A*]]
lemmas *marl-butlast-hnr-mop*[*sepref-fr-rules*] = *marl-butlast-hnr*[*FCOMP mk-mop-rl1*[*OF mop-list-butlast-alt*]]

lemma *marl-get-hnr-aux*: (*uncurry marl-get*,*uncurry* (*RETURN oo op-list-get*)) ∈ [λ(*l*,*i*). *i*<*length l*]$_a$ ((*is-ms-array-list N*)^k *$*_a$ *nat-assn*^{k}) → *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*)) ∈ [λ((*l*,*i*),-). *i*<*length l*]$_a$ ((*is-ms-array-list N*)^d *$*_a$ *nat-assn*^{k} *$*_a$ *id-assn*^{k}) → (*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*) *?Γ' ?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* × *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* ⊆ {*0*..<*length qp*}
assumes *qp-def*: ∀ *k*<*length qp*. *qp*!*k* = (*if k*∈*set l then List-Index.index l k else length qp*)
begin
lemma *l-len*: *length l* ≤ *length qp*
⟨*proof*⟩

lemma *idx-len*[*simp*]: *i*<*length l* ⟹ *l*!*i* < *length qp*
⟨*proof*⟩

lemma *l-set-simp*[*simp*]: *k*∈*set l* ⟹ *k* < *length qp*
⟨*proof*⟩

lemma *qpk-idx*: *k*<*length qp* ⟹ *qp* ! *k* < *length l* ⟷ *k* ∈ *set l*
⟨*proof*⟩

lemma *lqpk*[*simp*]: *k* ∈ *set l* ⟹ *l* ! (*qp* ! *k*) = *k*
⟨*proof*⟩

lemma ⟦*i*<*length l*; *j*<*length l*; *l*!*i*=*l*!*j*⟧ ⟹ *i*=*j*
⟨*proof*⟩

lemmas *index-swap*[*simp*] = *index-swap-if-distinct*[*folded swap-def*, *OF l-distinct*]

lemma *swap-invar*:
assumes *i*<*length l j*<*length l*
shows *ial-invar* (*length qp*) (*swap l i j*) (*qp*[*l* ! *j* := *i*, *l* ! *i* := *j*])

⟨*proof*⟩

end

definition *ial-rel1 maxsize* ≡ *br fst* (*uncurry* (*ial-invar maxsize*))

definition *ial-assn2* :: *nat* ⇒ *nat list* ∗ *nat list* ⇒ - **where**
ial-assn2 maxsize ≡ *prod-assn* (*marl-assn maxsize nat-assn*) (*array-assn nat-assn*)

definition *ial-assn maxsize A* ≡ *hr-comp* (*hr-comp* (*ial-assn2 maxsize*) (*ial-rel1 maxsize*)) (⟨*the-pure A*⟩*list-rel*)
lemmas [*safe-constraint-rules*] = *CN-FALSEI*[*of is-pure ial-assn maxsize A* **for** *maxsize A*]

3.11.1 Empty

definition *op-ial-empty-sz* :: *nat* ⇒ *'a list*
where [*simp*]: *op-ial-empty-sz ms* ≡ *op-list-empty*

lemma [*def-pat-rules*]: *op-ial-empty-sz*$*maxsize* ≡ *UNPROTECT* (*op-ial-empty-sz maxsize*)
⟨*proof*⟩

context fixes *maxsize* :: *nat* **begin**
sepref-register *PR-CONST* (*op-ial-empty-sz maxsize*)
end

context
fixes *maxsize* :: *nat*
notes [*fcomp-norm-unfold*] = *ial-assn-def*[*symmetric*]
notes [*simp*] = *hn-ctxt-def pure-def*
begin

definition *aial-empty* ≡ *do* {
let l = *op-marl-empty-sz maxsize*;
let qp = *op-array-replicate maxsize maxsize*;
RETURN (*l*,*qp*)
}

lemma *aial-empty-impl*: (*aial-empty*,*RETURN op-list-empty*) ∈ ⟨*ial-rel1 maxsize*⟩*nres-rel*
⟨*proof*⟩

context
notes [*id-rules*] = *itypeI*[*Pure.of maxsize TYPE*(*nat*)]
notes [*sepref-import-param*] = *IdI*[*of maxsize*]

begin
sepref-definition *ial-empty* **is** *uncurry0 aial-empty* :: $unit\text{-}assn^k \rightarrow_a ial\text{-}assn2\ maxsize$
⟨*proof*⟩
end

sepref-decl-impl (*no-register*) *ial-empty*: *ial-empty.refine*[*FCOMP aial-empty-impl*]
⟨*proof*⟩
lemma *ial-empty-sz-hnr*[*sepref-fr-rules*]:
(*uncurry0 local.ial-empty*, *uncurry0* (*RETURN* (*PR-CONST* (*op-ial-empty-sz maxsize*)))) ∈ $unit\text{-}assn^k \rightarrow_a ial\text{-}assn\ maxsize\ A$
⟨*proof*⟩

3.11.2 Swap

definition *aial-swap* ≡ λ(*l*,*qp*) *i j*. *do* {
vi ← *mop-list-get l i*;
vj ← *mop-list-get l j*;
l ← *mop-list-set l i vj*;
l ← *mop-list-set l j vi*;
qp ← *mop-list-set qp vj i*;
qp ← *mop-list-set qp vi j*;
RETURN (*l*,*qp*)
}

lemma *in-ial-rel1-conv*:
((*pq*, *qp*), *l*) ∈ *ial-rel1 ms* ⟷ *pq*=*l* ∧ *ial-invar ms l qp*
⟨*proof*⟩

lemma *aial-swap-impl*:
(*aial-swap*,*mop-list-swap*) ∈ *ial-rel1 maxsize* → *nat-rel* → *nat-rel* → ⟨*ial-rel1 maxsize*⟩*nres-rel*
⟨*proof*⟩

sepref-definition *ial-swap* **is**
uncurry2 aial-swap :: $(ial\text{-}assn2\ maxsize)^d *_a nat\text{-}assn^k *_a nat\text{-}assn^k \rightarrow_a ial\text{-}assn2\ maxsize$
⟨*proof*⟩

sepref-decl-impl (*ismop*) *test*: *ial-swap.refine*[*FCOMP aial-swap-impl*]
uses *mop-list-swap.fref* ⟨*proof*⟩

3.11.3 Length

definition *aial-length* :: *aial* ⇒ *nat nres*
where *aial-length* ≡ λ(*l*,-). *RETURN* (*op-list-length l*)

lemma *aial-length-impl*: (*aial-length*, *mop-list-length*) ∈ *ial-rel1 maxsize* → ⟨*nat-rel*⟩*nres-rel*
⟨*proof*⟩

sepref-definition *ial-length* **is** *aial-length* :: $(ial\text{-}assn2\ maxsize)^k \to_a nat\text{-}assn$
⟨*proof*⟩

sepref-decl-impl (*ismop*) *ial-length*: *ial-length.refine*[*FCOMP aial-length-impl*]
⟨*proof*⟩

3.11.4 Index

definition *aial-index* :: *aial* ⇒ *nat* ⇒ *nat nres* **where**
aial-index ≡ λ(*l*,*qp*) *k*. *do* {
ASSERT (*k*∈*set l*);
i ← *mop-list-get qp k*;
RETURN i
}

lemma *aial-index-impl*:
(*uncurry aial-index*, *uncurry mop-list-index*) ∈
[λ(*l*,*k*). *k*∈*set l*]$_f$ *ial-rel1 maxsize* ×$_r$ *nat-rel* → ⟨*nat-rel*⟩*nres-rel*
⟨*proof*⟩

sepref-definition *ial-index* **is** *uncurry aial-index* :: $(ial\text{-}assn2\ maxsize)^k$ $*_a$ $nat\text{-}assn^k \to_a nat\text{-}assn$
⟨*proof*⟩

sepref-decl-impl (*ismop*) *ial-index*: *ial-index.refine*[*FCOMP aial-index-impl*]
⟨*proof*⟩

3.11.5 Butlast

definition *aial-butlast* :: *aial* ⇒ *aial nres* **where**
aial-butlast ≡ λ(*l*,*qp*). *do* {
ASSERT (*l*≠[]);
len ← *mop-list-length l*;
k ← *mop-list-get l* (*len* − *1*);
l ← *mop-list-butlast l*;
qp ← *mop-list-set qp k* (*length qp*);
RETURN (*l*,*qp*)
}

lemma *aial-butlast-refine*: (*aial-butlast*, *mop-list-butlast*) ∈ *ial-rel1 maxsize* → ⟨*ial-rel1 maxsize*⟩*nres-rel*
⟨*proof*⟩

sepref-definition *ial-butlast* **is** *aial-butlast* :: $(ial\text{-}assn2\ maxsize)^d \to_a ial\text{-}assn2\ maxsize$
⟨*proof*⟩

sepref-decl-impl (*ismop*) *ial-butlast*: *ial-butlast.refine*[*FCOMP aial-butlast-refine*]
⟨*proof*⟩

3.11.6 Append

definition *aial-append* :: *aial* ⇒ *nat* ⇒ *aial nres* **where**
aial-append ≡ λ(*l*,*qp*) *k*. *do* {
ASSERT (*k*<*length qp* ∧ *k*∉*set l* ∧ *length l* < *length qp*);
len ← *mop-list-length l*;
l ← *mop-list-append l k*;
qp ← *mop-list-set qp k len*;
RETURN (*l*,*qp*)
}

lemma *aial-append-refine*:
(*uncurry aial-append*,*uncurry mop-list-append*) ∈
[λ(*l*,*k*). *k*<*maxsize* ∧ *k*∉*set l*]$_f$ *ial-rel1 maxsize* ×$_r$ *nat-rel* → ⟨*ial-rel1 maxsize*⟩*nres-rel*
⟨*proof*⟩ **lemma** *aial-append-impl-aux*: ((*l*, *qp*), *l'*) ∈ *ial-rel1 maxsize* ⟹ *l'*=*l* ∧ *maxsize* = *length qp*
⟨*proof*⟩

context
notes [*dest*!] = *aial-append-impl-aux*
begin

sepref-definition *ial-append* **is**
uncurry aial-append :: [λ(*lqp*,-). *lqp*∈*Domain* (*ial-rel1 maxsize*)]$_a$ (*ial-assn2 maxsize*)^d *$*_a$ *nat-assn*^{k} → *ial-assn2 maxsize*
⟨*proof*⟩
end

lemma (λ*b*. *b*<*maxsize*, *X*) ∈ *A* → *bool-rel*
⟨*proof*⟩

context begin

private lemma *append-fref'*: ⟦*IS-BELOW-ID R*⟧
⟹ (*uncurry mop-list-append*, *uncurry mop-list-append*) ∈ ⟨*R*⟩*list-rel* ×$_r$ *R* →$_f$ ⟨⟨*R*⟩*list-rel*⟩*nres-rel*
⟨*proof*⟩

sepref-decl-impl (*ismop*) *ial-append*: *ial-append.refine*[*FCOMP aial-append-refine*] **uses** *append-fref'*
⟨*proof*⟩
end

3.11.7 Get

definition *aial-get* :: *aial* ⇒ *nat* ⇒ *nat nres* **where**
aial-get ≡ λ(*l*,*qp*) *i*. *mop-list-get l i*

lemma *aial-get-refine*: (*aial-get*,*mop-list-get*) ∈ *ial-rel1 maxsize* → *nat-rel* → ⟨*nat-rel*⟩*nres-rel*
⟨*proof*⟩

sepref-definition *ial-get* **is** *uncurry aial-get* :: (*ial-assn2 maxsize*)^k *∗$_a$ nat-assn*^{k} →$_a$ *nat-assn*
⟨*proof*⟩

sepref-decl-impl (*ismop*) *ial-get*: *ial-get.refine*[*FCOMP aial-get-refine*] ⟨*proof*⟩

3.11.8 Contains

definition *aial-contains* :: *nat* ⇒ *aial* ⇒ *bool nres* **where**
aial-contains ≡ λ*k* (*l*,*qp*). *do* {
if k<*maxsize then do* {
i ← *mop-list-get qp k*;
RETURN (*i*<*maxsize*)
} *else RETURN False*
}

lemma *aial-contains-refine*: (*uncurry aial-contains*,*uncurry mop-list-contains*)
∈ (*nat-rel* ×$_r$ *ial-rel1 maxsize*) →$_f$ ⟨*bool-rel*⟩*nres-rel*
⟨*proof*⟩

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*^{k} *∗$_a$* (*ial-assn2 maxsize*)^k →$_a$ *bool-assn*
⟨*proof*⟩
end

sepref-decl-impl (*ismop*) *ial-contains*: *ial-contains.refine*[*FCOMP aial-contains-refine*] ⟨*proof*⟩
end

end

3.12 Implementation of Heaps by Arrays

theory *IICF-Impl-Heapmap*
imports *IICF-Abs-Heapmap ../IICF-Indexed-Array-List*
begin

Some setup to circumvent the really inefficient implementation of division in the code generator, which has to consider several cases for negative divisors and dividends.

definition [*code-unfold*]:

efficient-nat-div2 n
≡ *nat-of-integer* (*fst* (*Code-Numeral.divmod-abs* (*integer-of-nat n*) *2*))

lemma *efficient-nat-div2*[*simp*]: *efficient-nat-div2 n* = *n div 2*
⟨*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*,λ*k*. *if k*∈*set pq then Some* (*ml*!*k*) *else None*)

definition *hm-impl1-invar* ≡ λ(*pq*,*ml*).
hmr-invar (*hm-impl1-α* (*pq*,*ml*))
∧ *set pq* ⊆ {*0*..<*maxsize*}
∧ ((*pq*=[] ∧ *ml*=[]) ∨ (*length ml* = *maxsize*))

definition *hm-impl1-weak-invar* ≡ λ(*pq*,*ml*).
set pq ⊆ {*0*..<*maxsize*}
∧ ((*pq*=[] ∧ *ml*=[]) ∨ (*length ml* = *maxsize*))

definition *hm-impl1-rel* ≡ *br hm-impl1-α hm-impl1-invar*
definition *hm-weak-impl'-rel* ≡ *br hm-impl1-α hm-impl1-weak-invar*

lemmas *hm-impl1-rel-defs* =
hm-impl1-rel-def hm-weak-impl'-rel-def hm-impl1-weak-invar-def hm-impl1-invar-def hm-impl1-α-def in-br-conv

lemma *hm-impl-α-fst-eq*:
(*x1*, *x2*) = *hm-impl1-α* (*x1a*, *x2a*) ⟹ *x1* = *x1a*

⟨*proof*⟩

term *hm-empty-op*
definition *hm-empty-op'* :: *'v hma nres*
where *hm-empty-op'* ≡ *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*) ∈ ⟨*hm-impl1-rel*⟩*nres-rel*

⟨*proof*⟩

definition *hm-length'* :: *'v hma* ⇒ *nat* **where** *hm-length'* ≡ λ(*pq*,*ml*). *length pq*

lemma *hm-length'-refine*: (*RETURN o hm-length'*,*RETURN o hm-length*) ∈ *hm-impl1-rel* → ⟨*nat-rel*⟩*nres-rel*
⟨*proof*⟩

term *hm-key-of-op*
definition *hm-key-of-op'* ≡ λ(*pq*,*ml*) *i*. *ASSERT* (*i*>*0*) ≫ *mop-list-get pq* (*i* − *1*)
lemma *hm-key-of-op'-refine*: (*hm-key-of-op'*, *hm-key-of-op*) ∈ *hm-impl1-rel* → *nat-rel* → ⟨*nat-rel*⟩*nres-rel*
⟨*proof*⟩

term *hm-lookup*
definition *hm-lookup-op'* ≡ λ(*pq*,*ml*) *k*. *do* {
if (*k*<*maxsize*) *then do* { — TODO: This check can be eliminated, but this will complicate refinement of keys in basic ops
let c = op-list-contains k pq;
if c then do {
v ← *mop-list-get ml k*;
RETURN (*Some v*)
} *else RETURN None*
} *else RETURN None*
}

lemma *hm-lookup-op'-refine*: (*uncurry hm-lookup-op'*, *uncurry* (*RETURN oo hm-lookup*))
∈ (*hm-impl1-rel* ×*r nat-rel*) →*f* ⟨⟨*Id*⟩*option-rel*⟩*nres-rel*
⟨*proof*⟩

term *hm-contains-key-op*
definition *hm-contains-key-op'* ≡ λ*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*)

∈ (*nat-rel* ×$_r$ *hm-impl1-rel*) →$_f$ ⟨*bool-rel*⟩*nres-rel*
⟨*proof*⟩

term *hm-valid*

definition *hm-exch-op′* ≡ λ(*pq*,*ml*) *i j*. *do* {
ASSERT (*hm-valid* (*hm-impl1-α* (*pq*,*ml*)) *i*);
ASSERT (*hm-valid* (*hm-impl1-α* (*pq*,*ml*)) *j*);
pq ← *mop-list-swap pq* (*i* − *1*) (*j* − *1*);
RETURN (*pq*,*ml*)
}

lemma *hm-impl1-relI*:
assumes *hmr-invar b*
assumes (*a*,*b*)∈*hm-weak-impl′-rel*
shows (*a*,*b*)∈*hm-impl1-rel*
⟨*proof*⟩

lemma *hm-impl1-nres-relI*:
assumes *b* ≤$_n$ *SPEC hmr-invar*
assumes (*a*,*b*)∈⟨*hm-weak-impl′-rel*⟩*nres-rel*
shows (*a*,*b*)∈⟨*hm-impl1-rel*⟩*nres-rel*
⟨*proof*⟩

lemma *hm-exch-op′-refine*: (*hm-exch-op′*, *hm-exch-op*) ∈ *hm-impl1-rel* → *nat-rel* → *nat-rel* → ⟨*hm-impl1-rel*⟩*nres-rel*
⟨*proof*⟩

term *hm-index-op*

definition *hm-index-op′* ≡ λ(*pq*,*ml*) *k*.
do {
ASSERT (*hm-impl1-invar* (*pq*,*ml*) ∧ *heapmap-α* (*hm-impl1-α* (*pq*,*ml*)) *k* ≠ *None* ∧ *k*∈*set pq*);
i ← *mop-list-index pq k*;
RETURN (*i*+*1*)
}
lemma *hm-index-op′-refine*: (*hm-index-op′*,*hm-index-op*)

$\in$ *hm-impl1-rel* $\to$ *nat-rel* $\to$ $\langle$*nat-rel*$\rangle$*nres-rel*
⟨*proof*⟩

definition *hm-update-op′* **where**
hm-update-op′ ≡ λ(*pq*,*ml*) *i v*. *do* {
ASSERT (*hm-valid* (*hm-impl1-α* (*pq*,*ml*)) *i* ∧ *hm-impl1-invar* (*pq*,*ml*));
k ← *mop-list-get pq* (*i* − *1*);
ml ← *mop-list-set ml k v*;
RETURN (*pq*, *ml*)
}
lemma *hm-update-op′-refine*: (*hm-update-op′*, *hm-update-op*) ∈ *hm-impl1-rel* → *nat-rel* → *Id* → ⟨*hm-impl1-rel*⟩*nres-rel*
⟨*proof*⟩

term *hm-butlast-op*

lemma *hm-butlast-op-invar*: *hm-butlast-op hm* $\leq_n$ *SPEC hmr-invar*
⟨*proof*⟩

definition *hm-butlast-op′* **where**
hm-butlast-op′ ≡ λ(*pq*,*ml*). *do* {
ASSERT (*hmr-invar* (*hm-impl1-α* (*pq*,*ml*)));
pq ← *mop-list-butlast pq*;
RETURN (*pq*,*ml*)
}

lemma *set-butlast-distinct-conv*:
⟦*distinct l*⟧ ⟹ *set* (*butlast l*) = *set l* − {*last l*}
⟨*proof*⟩

lemma *hm-butlast-op′-refine*: (*hm-butlast-op′*, *hm-butlast-op*) ∈ *hm-impl1-rel* → ⟨*hm-impl1-rel*⟩*nres-rel*
⟨*proof*⟩

definition *hm-append-op′*
where *hm-append-op′* ≡ λ(*pq*,*ml*) *k v*. *do* {
ASSERT (*k* ∉ *set pq* ∧ *k*<*maxsize*);
ASSERT (*hm-impl1-invar* (*pq*,*ml*));
pq ← *mop-list-append pq k*;
ml ← (*if length ml* = *0 then mop-list-replicate maxsize v else RETURN ml*);
ml ← *mop-list-set ml k v*;
RETURN (*pq*,*ml*)
}

lemma *hm-append-op′-refine*: (*uncurry2 hm-append-op′*, *uncurry2 hm-append-op*)

∈ [λ((*hm*,*k*),*v*). *k*<*maxsize*]$_f$ (*hm-impl1-rel* ×$_r$ *nat-rel*) ×$_r$ *Id* → ⟨*hm-impl1-rel*⟩*nres-rel*

⟨*proof*⟩

definition *hm-impl2-rel* ≡ *prod-assn* (*ial-assn maxsize id-assn*) (*array-assn id-assn*)
definition *hm-impl-rel* ≡ *hr-comp hm-impl2-rel hm-impl1-rel*

lemmas [*fcomp-norm-unfold*] = *hm-impl-rel-def*[*symmetric*]

3.12.1 Implement Basic Operations

lemma *param-parent*: (*efficient-nat-div2*,*h.parent*) ∈ *Id* → *Id*
⟨*proof*⟩
lemmas [*sepref-import-param*] = *param-parent*
sepref-register *h.parent*

lemma *param-left*: (*h.left*,*h.left*) ∈ *Id* → *Id* ⟨*proof*⟩
lemmas [*sepref-import-param*] = *param-left*
sepref-register *h.left*

lemma *param-right*: (*h.right*,*h.right*) ∈ *Id* → *Id* ⟨*proof*⟩
lemmas [*sepref-import-param*] = *param-right*
sepref-register *h.right*

abbreviation (*input*) *prio-rel* ≡ (*Id*::(*'p*×*'p*) *set*)

lemma *param-prio-le*: ((≤), (≤)) ∈ *prio-rel* → *prio-rel* → *bool-rel* ⟨*proof*⟩
lemmas [*sepref-import-param*] = *param-prio-le*

lemma *param-prio-lt*: ((<), (<)) ∈ *prio-rel* → *prio-rel* → *bool-rel* ⟨*proof*⟩
lemmas [*sepref-import-param*] = *param-prio-lt*

abbreviation *I-HM-UNF* ≡ *TYPE*(*nat list* × *'v list*)

sepref-definition *hm-length-impl* **is** *RETURN o hm-length'* :: *hm-impl2-rel*^{k}→$_a$*nat-assn*
⟨*proof*⟩
lemmas [*sepref-fr-rules*] = *hm-length-impl.refine*[*FCOMP hm-length'-refine*]
sepref-register *hm-length*::(*nat*,*'v*) *ahm* ⇒ -

sepref-definition *hm-key-of-op-impl* **is** *uncurry hm-key-of-op'* :: *hm-impl2-rel*^{k}*$_a$*nat-assn*^{k} →$_a$*nat-assn*
⟨*proof*⟩
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* ⇒ -

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'* :: $(hm\text{-}impl2\text{-}rel^k *_a nat\text{-}assn^k \rightarrow_a option\text{-}assn\ id\text{-}assn)$
⟨*proof*⟩
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'* :: $hm\text{-}impl2\text{-}rel^d *_a nat\text{-}assn^k *_a nat\text{-}assn^k \rightarrow_a hm\text{-}impl2\text{-}rel$
⟨*proof*⟩
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'* :: $hm\text{-}impl2\text{-}rel^k *_a id\text{-}assn^k \rightarrow_a id\text{-}assn$
⟨*proof*⟩
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'* :: $hm\text{-}impl2\text{-}rel^d *_a id\text{-}assn^k *_a id\text{-}assn^k \rightarrow_a hm\text{-}impl2\text{-}rel$
⟨*proof*⟩
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'* :: $hm\text{-}impl2\text{-}rel^d \rightarrow_a hm\text{-}impl2\text{-}rel$
⟨*proof*⟩
lemmas [*sepref-fr-rules*] = *hm-butlast-op-impl.refine*[*FCOMP hm-butlast-op'-refine*]
sepref-register *hm-butlast-op*::$(nat,'v)\ ahm \Rightarrow$ -

sepref-definition *hm-append-op-impl* **is** *uncurry2 hm-append-op'* :: $hm\text{-}impl2\text{-}rel^d *_a id\text{-}assn^k *_a id\text{-}assn^k \rightarrow_a hm\text{-}impl2\text{-}rel$
⟨*proof*⟩
lemmas [*sepref-fr-rules*] = *hm-append-op-impl.refine*[*FCOMP hm-append-op'-refine*]
sepref-register *hm-append-op*::$(nat,'v)\ ahm \Rightarrow$ -

3.12.2 Auxiliary Operations

lemmas [*intf-of-assn*] = *intf-of-assnI*[**where** *R*=*hm-impl-rel* :: $(nat,'v)\ ahm \Rightarrow$ - **and** $'a='v$ *i-hma*]

sepref-definition *hm-valid-impl* **is** *uncurry* (*RETURN oo hm-valid*) :: $hm\text{-}impl\text{-}rel^k *_a nat\text{-}assn^k \rightarrow_a bool\text{-}assn$
⟨*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*^{k}*$_a$*id-assn*^{k} →$_a$ *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*^{k}*$_a$*id-assn*^{k} →$_a$ *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-impl-rel*^{k}*$_a$*id-assn*^{k} →$_a$ *id-assn*
⟨*proof*⟩
lemmas [*sepref-fr-rules*] = *hm-prio-of-op-impl.refine*
sepref-register *PR-CONST hm-prio-of-op*::(*nat*,'*v*) *ahm* ⇒ -
lemma [*def-pat-rules*]: *hmstruct.hm-prio-of-op*$*prio* ≡ *PR-CONST hm-prio-of-op*
⟨*proof*⟩

No code theorem preparation, as we define optimized version later

sepref-definition (*no-prep-code*) *hm-swim-op-impl* **is** *uncurry* (*PR-CONST hm-swim-op*) :: *hm-impl-rel*^{d}*$_a$*nat-assn*^{k} →$_a$ *hm-impl-rel*
⟨*proof*⟩
lemmas [*sepref-fr-rules*] = *hm-swim-op-impl.refine*
sepref-register *PR-CONST hm-swim-op*::(*nat*,'*v*) *ahm* ⇒ -
lemma [*def-pat-rules*]: *hmstruct.hm-swim-op*$*prio* ≡ *PR-CONST hm-swim-op*
⟨*proof*⟩

No code theorem preparation, as we define optimized version later

sepref-definition (*no-prep-code*) *hm-sink-op-impl* **is** *uncurry* (*PR-CONST hm-sink-op*) :: *hm-impl-rel*^{d}*$_a$*nat-assn*^{k} →$_a$ *hm-impl-rel*
⟨*proof*⟩
lemmas [*sepref-fr-rules*] = *hm-sink-op-impl.refine*

sepref-register *PR-CONST hm-sink-op*::(*nat*,'*v*) *ahm* ⇒ -
lemma [*def-pat-rules*]: *hmstruct.hm-sink-op*$*prio* ≡ *PR-CONST hm-sink-op*
⟨*proof*⟩

sepref-definition *hm-repair-op-impl* **is** *uncurry* (*PR-CONST hm-repair-op*) ::
hm-impl-rel^d ∗$_a$ *nat-assn*^{k} →$_a$ *hm-impl-rel*
⟨*proof*⟩
lemmas [*sepref-fr-rules*] = *hm-repair-op-impl.refine*
sepref-register *PR-CONST hm-repair-op*::(*nat*,'*v*) *ahm* ⇒ -
lemma [*def-pat-rules*]: *hmstruct.hm-repair-op*$*prio* ≡ *PR-CONST hm-repair-op*
⟨*proof*⟩

3.12.3 Interface Operations

definition *hm-rel-np* **where**
hm-rel-np ≡ *hr-comp hm-impl-rel heapmap-rel*
lemmas [*fcomp-norm-unfold*] = *hm-rel-np-def*[*symmetric*]

definition *hm-rel* **where**
hm-rel K V ≡ *hr-comp hm-rel-np* (⟨*the-pure K*,*the-pure V*⟩*map-rel*)
lemmas [*fcomp-norm-unfold*] = *hm-rel-def*[*symmetric*]

lemmas [*intf-of-assn*] = *intf-of-assnI*[**where** *R*=*hm-rel K V* **and** '*a*=('*kk*,'*vv*)
i-map **for** *K V*]

lemma *hm-rel-id-conv*: *hm-rel id-assn id-assn* = *hm-rel-np*
— Used for generic algorithms: Unfold with this, then let decl-impl compose
with *map-rel* again.
⟨*proof*⟩

Synthesis

definition *op-hm-empty-sz* :: *nat* ⇒ '*kk*⇀'*vv*
where [*simp*]: *op-hm-empty-sz sz* ≡ *op-map-empty*
sepref-register *PR-CONST* (*op-hm-empty-sz maxsize*) :: ('*k*,'*v*) *i-map*
lemma [*def-pat-rules*]: *op-hm-empty-sz*$*maxsize* ≡ *UNPROTECT* (*op-hm-empty-sz*
maxsize) ⟨*proof*⟩

lemma *hm-fold-custom-empty-sz*:
op-map-empty = *op-hm-empty-sz sz*
Map.empty = *op-hm-empty-sz sz*
⟨*proof*⟩

sepref-definition *hm-empty-op-impl* **is** *uncurry0 hm-empty-op'* :: *unit-assn*^{k} →$_a$
hm-impl2-rel
⟨*proof*⟩

sepref-definition *hm-insert-op-impl* **is** *uncurry2 hm-insert-op* :: [λ((*k*,-),-). *k*<*maxsize*]$_a$
id-assn^k ∗$_a$ *id-assn*^{k} ∗$_a$ *hm-impl-rel*^{d} → *hm-impl-rel*
⟨*proof*⟩

sepref-definition *hm-is-empty-op-impl* **is** *hm-is-empty-op* :: *hm-impl-rel*^{k} $\rightarrow_a$ *bool-assn*
⟨*proof*⟩

sepref-definition *hm-lookup-op-impl* **is** *uncurry hm-lookup-op* :: *id-assn*^{k} $*_a$ *hm-impl-rel*^{k} $\rightarrow_a$ *option-assn id-assn*
⟨*proof*⟩

sepref-definition *hm-contains-key-impl* **is** *uncurry hm-contains-key-op'* :: *id-assn*^{k} $*_a$ *hm-impl2-rel*^{k} $\rightarrow_a$ *bool-assn*
⟨*proof*⟩

sepref-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*
⟨*proof*⟩

sepref-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*
⟨*proof*⟩

sepref-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*
⟨*proof*⟩

sepref-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*^{k} $*_a$ *hm-impl-rel*^{d} $\rightarrow_a$ *hm-impl-rel*
⟨*proof*⟩

sepref-definition *hm-peek-min-op-impl* **is** *hm-peek-min-op* :: *hm-impl-rel*^{k} $\rightarrow_a$ *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*)
:: [λ((*k*,-),-). *k*<*maxsize*]$_a$ *id-assn*^{k} $*_a$ *id-assn*^{k} $*_a$ (*hm-rel id-assn id-assn*)^d →
hm-rel id-assn id-assn
⟨*proof*⟩

sepref-decl-impl *hm-upd-op-impl.refine*[*unfolded hm-rel-id-conv*] **uses** *op-map-update-fref'*
⟨*proof*⟩

end
end

interpretation *hm*: *map-custom-empty PR-CONST* (*op-hm-empty-sz maxsize*)
⟨*proof*⟩

lemma *op-hm-empty-sz-hnr*[*sepref-fr-rules*]:
(*uncurry0* (*hm-empty-op-impl maxsize*), *uncurry0* (*RETURN* (*PR-CONST* (*op-hm-empty-sz maxsize*)))) ∈ *unit-assn*^{k} →$_a$ *hm-rel maxsize prio K V*
⟨*proof*⟩

3.12.4 Manual fine-tuning of code-lemmas

context
notes [*simp del*] = *CNV-def efficient-nat-div2*

begin

lemma *nested-case-bind*:
(*case p of* (*a*,*b*) ⇒ *bind* (*case a of* (*a1*,*a2*) ⇒ *m a b a1 a2*) (*f a b*))
= (*case p of* ((*a1*,*a2*),*b*) ⇒ *bind* (*m* (*a1*,*a2*) *b a1 a2*) (*f* (*a1*,*a2*) *b*))
(*case p of* (*a*,*b*) ⇒ *bind* (*case b of* (*b1*,*b2*) ⇒ *m a b b1 b2*) (*f a b*))
= (*case p of* (*a*,*b1*,*b2*) ⇒ *bind* (*m a* (*b1*,*b2*) *b1 b2*) (*f a* (*b1*,*b2*)))
⟨*proof*⟩

lemma *it-case*: (*case p of* (*a*,*b*) ⇒ *f p a b*) = (*case p of* (*a*,*b*) ⇒ *f* (*a*,*b*) *a b*)
⟨*proof*⟩

lemma *c2l*: (*case p of* (*a*,*b*) ⇒ *bind* (*m a b*) (*f a b*)) =
do { *let* (*a*,*b*) = *p*; *bind* (*m a b*) (*f a b*)} ⟨*proof*⟩

lemma *bind-Let*: *do* { *x* ← *do* { *let y* = *v*; (*f y* :: *'a Heap*)}; *g x* } = *do* { *let y*=*v*; *x* ← *f y*; *g x* } ⟨*proof*⟩
lemma *bind-case*: *do* { *x* ← (*case y of* (*a*,*b*) ⇒ *f a b*); (*g x* :: *'a Heap*) } = *do* { *let* (*a*,*b*) = *y*; *x* ← *f a b*; *g x* }
⟨*proof*⟩

lemma *bind-case-mvup*: *do* { *x* ← *f*; *case y of* (*a*,*b*) ⇒ *g a b x* }
= *do* { *let* (*a*,*b*) = *y*; *x* ← *f*; (*g a b x* :: *'a Heap*) }
⟨*proof*⟩

lemma *if-case-mvup*: (*if b then case p of* (*x1*,*x2*) ⇒ *f x1 x2 else e*)
= (*case p of* (*x1*,*x2*) ⇒ *if b then f x1 x2 else e*) ⟨*proof*⟩

lemma *nested-case*: (*case p of* (*a*,*b*) ⇒ (*case p of* (*c*,*d*) ⇒ *f a b c d*)) =
(*case p of* (*a*,*b*) ⇒ *f a b a b*)
⟨*proof*⟩

lemma *split-prod-bound*: (λ*p*. *f p*) = (λ(*a*,*b*). *f* (*a*,*b*)) ⟨*proof*⟩

lemma *bpc-conv*: *do* { (*a*,*b*) ← (*m*::(-∗-) *Heap*); *f a b* } = *do* {
ab ← (*m*);
f (*fst ab*) (*snd ab*)
}
⟨*proof*⟩

lemma *it-case-pp*: (*case p of* ((*p1*,*p2*)) ⇒ *case p of* ((*p1'*,*p2'*)) ⇒ *f p1 p2 p1' p2'*)
= (*case p of* ((*p1*,*p2*)) ⇒ *f p1 p2 p1 p2*)
⟨*proof*⟩

lemma *it-case-ppp*: (*case p of* ((*p1*,*p2*),*p3*) ⇒ *case p of* ((*p1'*,*p2'*),*p3'*) ⇒ *f p1 p2 p3 p1' p2' p3'*)
= (*case p of* ((*p1*,*p2*),*p3*) ⇒ *f p1 p2 p3 p1 p2 p3*)

⟨*proof*⟩

lemma *it-case-pppp*: (*case a1 of*
(((*a*, *b*), *c*), *d*) ⇒
case a1 of
(((*a'*, *b'*), *c'*), *d'*) ⇒ *f a b c d a' b' c' d'*) =
(*case a1 of*
(((*a*, *b*), *c*), *d*) ⇒ *f a b c d a b c d*)
⟨*proof*⟩ **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*
⟨*proof*⟩

schematic-goal *hm-swim-op-impl prio hm i* ≡ *?f*
⟨*proof*⟩

lemma *hm-swim-op-impl-code*[*code*]: *hm-swim-op-impl prio hm i* ≡ *ccpo.fixp* (*fun-lub Heap-lub*) (*fun-ord Heap-ord*)
(λ*cf* (*a1*, *a2*).
case a1 of
((*a1b*, *a2b*), *a2a*) ⇒
case a1b of
(*a*, *b*) ⇒ *do* {
let d2 = *efficient-nat-div2 a2*;
if 0 < *d2* ∧ *d2* ≤ *b*
then do {
x ← (*case a1b of* (*a*, *n*) ⇒ *Array.nth a*) (*d2* − *Suc* 0);
x ← *Array.nth a2a x*;
xa ← (*case a1b of* (*a*, *n*) ⇒ *Array.nth a*) (*a2* − *Suc* 0);
xa ← *Array.nth a2a xa*;
if prio x ≤ *prio xa then return a1*
else do {
x'g ← *hm-exch-op-impl a1 a2* (*d2*);
cf (*x'g*, *d2*)
}
}
else return a1
})
(*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 *IICF-Matrix*
imports *../../Sepref*
begin

3.13.1 Relator and Interface

definition [*to-relAPP*]: *mtx-rel A* ≡ *nat-rel* ×$_r$ *nat-rel* → *A*

lemma *mtx-rel-id*[*simp*]: ⟨*Id*⟩*mtx-rel* = *Id* ⟨*proof*⟩

type-synonym *'a mtx* = *nat*×*nat* ⇒ *'a*
sepref-decl-intf *'a i-mtx* **is** *nat*×*nat* ⇒ *'a*

lemma [*synth-rules*]: *INTF-OF-REL A TYPE*('*a*) ⟹ *INTF-OF-REL* (⟨*A*⟩*mtx-rel*) *TYPE*('*a i-mtx*)
⟨*proof*⟩

3.13.2 Operations

definition *op-mtx-new* :: *'a mtx* ⇒ *'a mtx* **where** [*simp*]: *op-mtx-new c* ≡ *c*

sepref-decl-op (*no-def*) *mtx-new*: *op-mtx-new* :: (*nat-rel*×$_r$*nat-rel* → *A*) → ⟨*A*⟩*mtx-rel*
⟨*proof*⟩

lemma *mtx-init-adhoc-frame-match-rule*[*sepref-frame-match-rules*]:
hn-val (*nat-rel*×$_r$*nat-rel* → *A*) *x y* ⟹$_t$ *hn-val* (*nat-rel*×$_r$*nat-rel* → *the-pure* (*pure A*)) *x y*
⟨*proof*⟩

definition *op-mtx-copy* :: *'a mtx* ⇒ *'a mtx* **where** [*simp*]: *op-mtx-copy c* ≡ *c*

sepref-decl-op (*no-def*) *mtx-copy*: *op-mtx-copy* :: ⟨*A*⟩*mtx-rel* → ⟨*A*⟩*mtx-rel* ⟨*proof*⟩

sepref-decl-op *mtx-get*: λ(*c*::*'a mtx*) *ij*. *c ij* :: ⟨*A*⟩*mtx-rel* → (*nat-rel*×$_r$*nat-rel*) → *A*
⟨*proof*⟩

sepref-decl-op *mtx-set*: *fun-upd*::*'a mtx* ⇒ - :: ⟨*A*⟩*mtx-rel* → (*nat-rel*×$_r$*nat-rel*) → *A* → ⟨*A*⟩*mtx-rel*
⟨*proof*⟩

definition *mtx-nonzero* :: - *mtx* ⇒ (*nat*×*nat*) *set* **where** *mtx-nonzero m* ≡ {(*i*,*j*). *m* (*i*,*j*)≠*0*}

sepref-decl-op *mtx-nonzero*: *mtx-nonzero* :: ⟨*A*⟩*mtx-rel* → ⟨*nat-rel*×$_r$*nat-rel*⟩*set-rel*
where *IS-ID* (*A*::(-×(-::*zero*)) *set*)
⟨*proof*⟩

3.13.3 Patterns

lemma *pat-amtx-get*: *c*$*e*≡*op-mtx-get*$′*c*$′*e* ⟨*proof*⟩
lemma *pat-amtx-set*: *fun-upd*$*c*$*e*$*v*≡*op-mtx-set*$′*c*$′*e*$′*v* ⟨*proof*⟩

lemmas *amtx-pats*[*pat-rules*] = *pat-amtx-get pat-amtx-set*

3.13.4 Pointwise Operations

Auxiliary Definitions and Lemmas

locale *pointwise-op* =
fixes *f* :: ′*p* ⇒ ′*s* ⇒ ′*s*
fixes *q* :: ′*s* ⇒ ′*p* ⇒ ′*a*
assumes *upd-indep1*[*simp*, *intro*]: *p*≠*p*′ ⟹ *q* (*f p s*) *p*′ = *q s p*′
assumes *upd-indep2*[*simp*, *intro*]: *p*≠*p*′ ⟹ *q* (*f p* (*f p*′ *s*)) *p* = *q* (*f p s*) *p*
begin
lemma *pointwise-upd-fold*: *distinct ps* ⟹
q (*fold f ps s*) *p* = (*if p*∈*set ps then q* (*f p s*) *p else q s p*)
⟨*proof*⟩

end

lemma *pointwise-fun-fold*:
fixes *f* :: ′*a* ⇒ (′*a* ⇒ ′*b*) ⇒ (′*a* ⇒ ′*b*)
fixes *s* :: ′*a* ⇒ ′*b*
assumes *indep1*: ⋀*x x*′ *s*. *x* ≠ *x*′ ⟹ *f x s x*′ = *s x*′
assumes *indep2*: ⋀*x x*′ *s*. *x* ≠ *x*′ ⟹ *f x* (*f x*′ *s*) *x* = *f x s x*
assumes [*simp*]: *distinct xs*
shows *fold f xs s x* = (*if x* ∈ *set xs then f x s x else s x*)
⟨*proof*⟩

lemma *list-prod-divmod-eq*: *List.product* [*0*..<*M*] [*0*..<*N*] = *map* (λ*i*. (*i div N*, *i mod N*)) [*0*..<*N*∗*M*]
⟨*proof*⟩

lemma *nfoldli-prod-divmod-conv*:
nfoldli (*List.product* [*0*..<*N*] [*0*..<*M*]) *ctd* (λ(*i*,*j*). *f i j*) = *nfoldli* [*0*..<*N*∗*M*] *ctd* (λ*i*. *f* (*i div M*) (*i mod M*))
⟨*proof*⟩

lemma *nfoldli-prod-divmod-conv'*:
nfoldli [0..<*M*] *ctd* (λ*i*. *nfoldli* [0..<*N*] *ctd* (*f i*)) = *nfoldli* [0..<*N*∗*M*] *ctd* (λ*i*. *f* (*i div N*) (*i mod N*))
⟨*proof*⟩

lemma *foldli-prod-divmod-conv'*:
foldli [0..<*M*] *ctd* (λ*i*. *foldli* [0..<*N*] *ctd* (*f i*)) = *foldli* [0..<*N*∗*M*] *ctd* (λ*i*. *f* (*i div N*) (*i mod N*))
(**is** *?lhs=?rhs*)
⟨*proof*⟩

lemma *fold-prod-divmod-conv'*: *fold* (λ*i*. *fold* (*f i*) [0..<*N*]) [0..<*M*] = *fold* (λ*i*. *f* (*i div N*) (*i mod N*)) [0..<*N*∗*M*]
⟨*proof*⟩

lemma *mtx-nonzero-cases*[*consumes 0*, *case-names nonzero zero*]:
obtains (*i*,*j*)∈*mtx-nonzero m* | *m* (*i*,*j*) = *0*
⟨*proof*⟩

Unary Pointwise

definition *mtx-pointwise-unop* :: (*nat*×*nat* ⇒ *'a* ⇒ *'a*) ⇒ *'a mtx* ⇒ *'a mtx* **where**
mtx-pointwise-unop f m ≡ λ(*i*,*j*). *f* (*i*,*j*) (*m*(*i*,*j*))

context fixes *f* :: *nat*×*nat* ⇒ *'a* ⇒ *'a* **begin**
sepref-register *PR-CONST* (*mtx-pointwise-unop f*) :: *'a i-mtx* ⇒ *'a i-mtx*
lemma [*def-pat-rules*]: *mtx-pointwise-unop*$*f* ≡ *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*]: ⟦ *i*≥*N* ∨ *j*≥*M* ⟧ ⟹ *f* (*i*,*j*) *0* = *0*
begin
definition *opr-fold-impl* ≡ *fold* (λ*i*. *fold* (λ*j m*. *m*((*i*,*j*) := *f* (*i*,*j*) (*m*(*i*,*j*)))) [0..<*M*]) [0..<*N*]

lemma *opr-fold-impl-eq*:
assumes *mtx-nonzero m* ⊆ {0..<*N*}×{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*) ∈ [λ*m*. *mtx-nonzero m* ⊆ {0..<*N*}×{0..<*M*}]$_f$ *Id* → *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*}×{*0*..<*M*}
assumes *get-impl-hnr*:
(*uncurry get-impl*,*uncurry* (*RETURN oo op-mtx-get*)) ∈ *assn*^{k} *∗*$_a$ (*prod-assn* (*nbn-assn N*) (*nbn-assn M*))^k →$_a$ *A*
assumes *set-impl-hnr*:
(*uncurry2 set-impl*,*uncurry2* (*RETURN ooo op-mtx-set*)) ∈ *assn*^{d} *∗*$_a$ (*prod-assn* (*nbn-assn N*) (*nbn-assn M*))^k *∗*$_a$ *A*^{k} →$_a$ *assn*
assumes *fi-hnr*:
(*uncurry fi*,*uncurry* (*RETURN oo f*)) ∈ (*prod-assn nat-assn nat-assn*)^k *∗*$_a$ *A*^{k} →$_a$ *A*
begin

lemma *this-loc*: *mtx-pointwise-unop-gen-impl N M f assn A get-impl set-impl fi*
⟨*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} →$_a$ *assn*
⟨*proof*⟩
end

concrete-definition (**in** −) *mtx-pointwise-unnop-fold-impl1* **uses** *mtx-pointwise-unop-gen-impl.opr-fold-im*
prepare-code-thms (**in** −) *mtx-pointwise-unnop-fold-impl1-def*

lemma *op-hnr*[*sepref-fr-rules*]: (*mtx-pointwise-unnop-fold-impl1 N M get-impl set-impl fi*, *RETURN* ∘ *PR-CONST* (*mtx-pointwise-unop f*)) ∈ *assn*^{d} →$_a$ *assn*
⟨*proof*⟩

end

Binary Pointwise

definition *mtx-pointwise-binop* :: ($'a$ ⇒ $'a$ ⇒ $'a$) ⇒ $'a$ *mtx* ⇒ $'a$ *mtx* ⇒ $'a$ *mtx*
where
mtx-pointwise-binop f m n ≡ λ(*i*,*j*). *f* (*m*(*i*,*j*)) (*n*(*i*,*j*))
context fixes *f* :: $'a$ ⇒ $'a$ ⇒ $'a$ **begin**

sepref-register *PR-CONST* (*mtx-pointwise-binop f*) :: *'a i-mtx* ⇒ *'a i-mtx* ⇒ *'a i-mtx*

lemma [*def-pat-rules*]: *mtx-pointwise-binop$f* ≡ *UNPROTECT* (*mtx-pointwise-binop f*) ⟨*proof*⟩

end

locale *mtx-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* (λ*i*. *fold* (λ*j m*. *m*((*i*,*j*) := *f* (*m*(*i*,*j*)) (*n*(*i*,*j*)))) [*0*..<*M*]) [*0*..<*N*] *m*

lemma *opr-fold-impl-eq*:
assumes *mtx-nonzero m* ⊆ {*0*..<*N*}×{*0*..<*M*}
assumes *mtx-nonzero n* ⊆ {*0*..<*N*}×{*0*..<*M*}
shows *mtx-pointwise-binop f m n* = *opr-fold-impl m n*
⟨*proof*⟩

lemma *opr-fold-impl-refine*: (*uncurry opr-fold-impl*, *uncurry* (*mtx-pointwise-binop f*)) ∈ [λ(*m*,*n*). *mtx-nonzero m* ⊆ {*0*..<*N*}×{*0*..<*M*} ∧ *mtx-nonzero n* ⊆ {*0*..<*N*}×{*0*..<*M*}]$_f$ *Id*×$_r$*Id* → *Id*
⟨*proof*⟩

end

locale *mtx-pointwise-binop-gen-impl* = *mtx-pointwise-binop-loc* +
fixes *assn* :: *'a mtx* ⇒ *'i* ⇒ *assn*
fixes *A* :: *'a* ⇒ *'ai* ⇒ *assn*
fixes *get-impl* :: *'i* ⇒ *nat*×*nat* ⇒ *'ai Heap*
fixes *set-impl* :: *'i* ⇒ *nat*×*nat* ⇒ *'ai* ⇒ *'i Heap*
fixes *fi* :: *'ai* ⇒ *'ai* ⇒ *'ai Heap*
assumes *assn-range*: *rdomp assn m* ⟹ *mtx-nonzero m* ⊆ {*0*..<*N*}×{*0*..<*M*}
assumes *get-impl-hnr*:
(*uncurry get-impl*,*uncurry* (*RETURN oo op-mtx-get*)) ∈ *assn*^{k} *a* (*prod-assn* (*nbn-assn N*) (*nbn-assn M*))^k →$_a$ *A*
assumes *set-impl-hnr*:
(*uncurry2 set-impl*,*uncurry2* (*RETURN ooo op-mtx-set*)) ∈ *assn*^{d} *a* (*prod-assn* (*nbn-assn N*) (*nbn-assn M*))^k *a* *A*^{k} →$_a$ *assn*
assumes *fi-hnr*:
(*uncurry fi*,*uncurry* (*RETURN oo f*)) ∈ *A*^{k} *a* *A*^{k} →$_a$ *A*

begin

lemma *this-loc*: *mtx-pointwise-binop-gen-impl N M f assn A get-impl set-impl fi*
⟨*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*) :: $assn^d *_a assn^k \rightarrow_a assn$
⟨*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*))) ∈ $assn^d *_a assn^k \rightarrow_a assn$
⟨*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* ⇒ '*a* ⇒ *bool* **begin**
sepref-register *PR-CONST* (*mtx-pointwise-cmpop f g*) :: '*a i-mtx* ⇒ '*a i-mtx* ⇒ *bool*
lemma [*def-pat-rules*]: *mtx-pointwise-cmpop*$*f*$*g* ≡ *UNPROTECT* (*mtx-pointwise-cmpop f g*) ⟨*proof*⟩
end

lemma *mtx-nonzeroD*:
⟦¬*i*<*N*; *mtx-nonzero m* ⊆ {*0*..<*N*}×{*0*..<*M*}⟧ ⟹ *m*(*i*,*j*) = *0*
⟦¬*j*<*M*; *mtx-nonzero m* ⊆ {*0*..<*N*}×{*0*..<*M*}⟧ ⟹ *m*(*i*,*j*) = *0*
⟨*proof*⟩

locale *mtx-pointwise-cmpop-loc* =
fixes *N* :: *nat* **and** *M* :: *nat*
fixes *f g* :: '*a*::{*zero*} ⇒ '*a* ⇒ *bool*
assumes *pres-zero*[*simp*]: *f 0 0* = *True g 0 0* = *False*
begin
definition *opr-fold-impl m n* ≡ *do* {
s ← *nfoldli* (*List.product* [*0*..<*N*] [*0*..<*M*]) (λ*s*. *s*≠*2*) (λ(*i*,*j*) *s*. *do* {

```
      if f (m(i,j)) (n(i,j)) then
        if s=0 then
          if g (m(i,j)) (n(i,j)) then RETURN 1 else RETURN s
        else
          RETURN s

      else RETURN 2
    }) (0::nat);
    RETURN (s=1)
  }

  lemma opr-fold-impl-eq:
    assumes mtx-nonzero m ⊆ {0..<N}×{0..<M}
    assumes mtx-nonzero n ⊆ {0..<N}×{0..<M}
    shows opr-fold-impl m n ≤ RETURN (mtx-pointwise-cmpop f g m n)
  ⟨proof⟩

  lemma opr-fold-impl-refine:
    (uncurry opr-fold-impl, uncurry (RETURN oo mtx-pointwise-cmpop f g)) ∈
[λ(m,n). mtx-nonzero m ⊆ {0..<N}×{0..<M} ∧ mtx-nonzero n ⊆ {0..<N}×{0..<M}]_f
Id×_rId → ⟨bool-rel⟩nres-rel
    ⟨proof⟩

end

locale mtx-pointwise-cmpop-gen-impl = mtx-pointwise-cmpop-loc +
  fixes assn :: 'a mtx ⇒ 'i ⇒ assn
  fixes A :: 'a ⇒ 'ai ⇒ 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)) ∈ assn^k *_a (prod-assn
(nbn-assn N) (nbn-assn M))^k →_a A
  assumes fi-hnr:
    (uncurry fi,uncurry (RETURN oo f)) ∈ A^k *_a A^k →_a bool-assn
  assumes gi-hnr:
    (uncurry gi,uncurry (RETURN oo g)) ∈ A^k *_a A^k →_a bool-assn
begin

  lemma this-loc: mtx-pointwise-cmpop-gen-impl N M f g assn A get-impl fi gi
    ⟨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* :: $assn^d *_a assn^k \rightarrow_a$ *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*))) ∈ $assn^d *_a assn^k \rightarrow_a$ *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* ≡ $\exists_A l.$ *mtx* $\mapsto_a$ *l* ∗ ↑(
length l = *N*∗*M*
∧ (∀ *i*<*N*. ∀ *j*<*M*. *l*!(*i*∗*M*+*j*) = *c* (*i*,*j*))
∧ (∀ *i j*. (*i*≥*N* ∨ *j*≥*M*) ⟶ *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* ⟹ *mtx-nonzero m* ⊆ {*0*..<*N*}×{*0*..<*M*}
⟨*proof*⟩

definition *mtx-tabulate N M c* ≡ *do* {
m ← *Array.new* (*N*∗*M*) *0*;
(-,-,*m*) ← *imp-for'* *0* (*N*∗*M*) (λ*k* (*i*,*j*,*m*). *do* {
Array.upd k (*c* (*i*,*j*)) *m*;
let j=*j*+*1*;
if j<*M then return* (*i*,*j*,*m*)
else return (*i*+*1*,*0*,*m*)

```
  }) (0,0,m);
  return m
}


definition amtx-copy ≡ array-copy

definition amtx-dflt N M v ≡ Array.make (N*M) (λi. v)

definition mtx-get M mtx e ≡ Array.nth mtx (fst e * M + snd e)
definition mtx-set M mtx e v ≡ Array.upd (fst e * M + snd e) v mtx

lemma mtx-idx-valid[simp]: ⟦i < (N::nat); j < M⟧ ⟹ 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') ⟷ (i=i' ∧ j=j')
  ⟨proof⟩



lemma mtx-tabulate-rl[sep-heap-rules]:
  assumes NONZ: mtx-nonzero c ⊆ {0..<N}×{0..<M}
  shows <emp> mtx-tabulate N M c <IICF-Array-Matrix.is-amtx N M c>
⟨proof⟩

lemma mtx-copy-rl[sep-heap-rules]:
  <is-amtx N M c mtx> amtx-copy mtx <λr. is-amtx N M c mtx * is-amtx N M
c r>
  ⟨proof⟩

definition PRES-ZERO-UNIQUE A ≡ (A``{0}={0} ∧ A^-1``{0} = {0})
lemma IS-ID-imp-PRES-ZERO-UNIQUE[constraint-rules]: IS-ID A ⟹ PRES-ZERO-UNIQUE
A
  ⟨proof⟩

definition op-amtx-dfltNxM :: nat ⇒ nat ⇒ 'a::zero ⇒ nat×nat⇒'a where
  [simp]: op-amtx-dfltNxM N M v ≡ λ(i,j). if i<N ∧ 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 → ⟨A⟩mtx-rel
  where CONSTRAINT PRES-ZERO-UNIQUE A
  ⟨proof⟩
end

lemma mtx-dflt-rl[sep-heap-rules]: <emp> amtx-dflt N M k <is-amtx N M
(op-amtx-dfltNxM N M k)>
  ⟨proof⟩
```

lemma *mtx-get-rl*[*sep-heap-rules*]: ⟦*i*<*N*; *j*<*M* ⟧ ⟹ <*is-amtx N M c mtx*> *mtx-get M mtx* (*i*,*j*) <λ*r*. *is-amtx N M c mtx* ∗ ↑(*r* = *c* (*i*,*j*))>
⟨*proof*⟩

lemma *mtx-set-rl*[*sep-heap-rules*]: ⟦*i*<*N*; *j*<*M* ⟧
⟹ <*is-amtx N M c mtx*> *mtx-set M mtx* (*i*,*j*) *v* <λ*r*. *is-amtx N M* (*c*((*i*,*j*) := *v*)) *r*>
⟨*proof*⟩

definition *amtx-assn N M A* ≡ *hr-comp* (*is-amtx N M*) (⟨*the-pure A*⟩*mtx-rel*)
lemmas [*fcomp-norm-unfold*] = *amtx-assn-def*[*symmetric*]
lemmas [*safe-constraint-rules*] = *CN-FALSEI*[*of is-pure amtx-assn N M A* **for** *N M A*]

lemma [*intf-of-assn*]: *intf-of-assn A TYPE*('*a*) ⟹ *intf-of-assn* (*amtx-assn N M A*) *TYPE*('*a i-mtx*)
⟨*proof*⟩

abbreviation *asmtx-assn N A* ≡ *amtx-assn N N A*

lemma *mtx-rel-pres-zero*:
assumes *PRES-ZERO-UNIQUE A*
assumes (*m*,*m*')∈⟨*A*⟩*mtx-rel*
shows *m ij* = *0* ⟷ *m' ij* = *0*
⟨*proof*⟩
apply1 (*clarsimp simp*: *IS-PURE-def PRES-ZERO-UNIQUE-def is-pure-conv mtx-rel-def*)
⟨*proof*⟩ **applyS** (*rule IdI*[*of ij*]) **applyS** *auto*
⟨*proof*⟩

lemma *amtx-assn-bounded*:
assumes *CONSTRAINT* (*IS-PURE PRES-ZERO-UNIQUE*) *A*
shows *rdomp* (*amtx-assn N M A*) *m* ⟹ *mtx-nonzero m* ⊆ {*0*..<*N*}×{*0*..<*M*}
⟨*proof*⟩

lemma *mtx-tabulate-aref*:
(*mtx-tabulate N M*, *RETURN o op-mtx-new*)
∈ [λ*c*. *mtx-nonzero c* ⊆ {*0*..<*N*}×{*0*..<*M*}]$_a$ *id-assn*^{k} → *IICF-Array-Matrix.is-amtx N M*
⟨*proof*⟩

lemma *mtx-copy-aref*:
(*amtx-copy*, *RETURN o op-mtx-copy*) ∈ (*is-amtx N M*)^k →$_a$ *is-amtx N M*
⟨*proof*⟩

lemma *mtx-nonzero-bid-eq*:
assumes *R*⊆*Id*

assumes $(a,\ a') \in Id \to R$
shows *mtx-nonzero a = mtx-nonzero a'*
⟨*proof*⟩

lemma *mtx-nonzero-zu-eq*:
assumes *PRES-ZERO-UNIQUE R*
assumes $(a,\ a') \in Id \to R$
shows *mtx-nonzero a = mtx-nonzero a'*
⟨*proof*⟩

lemma *op-mtx-new-fref'*:
CONSTRAINT PRES-ZERO-UNIQUE A ⟹ (*RETURN* ∘ *op-mtx-new*, *RETURN* ∘ *op-mtx-new*) ∈ (*nat-rel* ×$_r$ *nat-rel* → *A*) →$_f$ ⟨⟨*A*⟩*mtx-rel*⟩*nres-rel*
⟨*proof*⟩

sepref-decl-impl (*no-register*) *amtx-new-by-tab*: *mtx-tabulate-aref* **uses** *op-mtx-new-fref'*
⟨*proof*⟩

sepref-decl-impl *amtx-copy*: *mtx-copy-aref* ⟨*proof*⟩

definition [*simp*]: *op-amtx-new* (*N*::*nat*) (*M*::*nat*) ≡ *op-mtx-new*
lemma *amtx-fold-custom-new*:
op-mtx-new ≡ *op-amtx-new N M*
mop-mtx-new ≡ λ*c*. *RETURN* (*op-amtx-new N M c*)
⟨*proof*⟩

context fixes *N M* :: *nat* **begin**
sepref-register *PR-CONST* (*op-amtx-new N M*) :: (*nat* × *nat* ⇒ *'a*) ⇒ *'a i-mtx*
end

lemma *amtx-new-hnr*[*sepref-fr-rules*]:
fixes *A* :: *'a*::*zero* ⇒ *'b*::{*zero*,*heap*} ⇒ *assn*
shows *CONSTRAINT* (*IS-PURE PRES-ZERO-UNIQUE*) *A* ⟹
(*mtx-tabulate N M*, (*RETURN* ∘ *PR-CONST* (*op-amtx-new N M*)))
∈ [λ*x*. *mtx-nonzero x* ⊆ {*0*..<*N*} × {*0*..<*M*}]$_a$ (*pure* (*nat-rel* ×$_r$ *nat-rel* → *the-pure A*))^k → *amtx-assn N M A*
⟨*proof*⟩

lemma [*def-pat-rules*]: *op-amtx-new*$*N*$*M* ≡ *UNPROTECT* (*op-amtx-new N M*) ⟨*proof*⟩

context fixes *N M* :: *nat* **notes** [*param*] = *IdI*[*of N*] *IdI*[*of M*] **begin**

lemma *mtx-dflt-aref*:
(*amtx-dflt N M*, *RETURN o PR-CONST* (*op-amtx-dfltNxM N M*)) ∈ *id-assn*^{k}

$\rightarrow_a$ *is-amtx N M*
⟨*proof*⟩
sepref-decl-impl *amtx-dflt*: *mtx-dflt-aref* ⟨*proof*⟩

lemma *amtx-get-aref*:
(*uncurry* (*mtx-get M*), *uncurry* (*RETURN oo op-mtx-get*)) ∈ [λ(-,(*i*,*j*)). *i*<*N* ∧ *j*<*M*]$_a$ (*is-amtx N M*)^k $*_a$ (*prod-assn nat-assn nat-assn*)^k → *id-assn*
⟨*proof*⟩
sepref-decl-impl *amtx-get*: *amtx-get-aref* ⟨*proof*⟩

lemma *amtx-set-aref*: (*uncurry2* (*mtx-set M*), *uncurry2* (*RETURN ooo op-mtx-set*))

∈ [λ((-,(*i*,*j*)),-). *i*<*N* ∧ *j*<*M*]$_a$ (*is-amtx N M*)^d $*_a$ (*prod-assn nat-assn nat-assn*)^k $*_a$ *id-assn*^{k} → *is-amtx N M*
⟨*proof*⟩

sepref-decl-impl *amtx-set*: *amtx-set-aref* ⟨*proof*⟩

lemma *amtx-get-aref′*:
(*uncurry* (*mtx-get M*), *uncurry* (*RETURN oo op-mtx-get*)) ∈ (*is-amtx N M*)^k $*_a$ (*prod-assn* (*pure* (*nbn-rel N*)) (*pure* (*nbn-rel M*)))^k $\rightarrow_a$ *id-assn*
⟨*proof*⟩

sepref-decl-impl *amtx-get′*: *amtx-get-aref′* ⟨*proof*⟩

lemma *amtx-set-aref′*: (*uncurry2* (*mtx-set M*), *uncurry2* (*RETURN ooo op-mtx-set*))

∈ (*is-amtx N M*)^d $*_a$ (*prod-assn* (*pure* (*nbn-rel N*)) (*pure* (*nbn-rel M*)))^k $*_a$ *id-assn*^{k} $\rightarrow_a$ *is-amtx N M*
⟨*proof*⟩

sepref-decl-impl *amtx-set′*: *amtx-set-aref′* ⟨*proof*⟩

end

3.14.1 Pointwise Operations

context
fixes *M N* :: *nat*
begin
sepref-decl-op *amtx-lin-get*: λ*f i*. *op-mtx-get f* (*i div M*, *i mod M*) :: ⟨*A*⟩*mtx-rel* → *nat-rel* → *A*
⟨*proof*⟩

sepref-decl-op *amtx-lin-set*: λ*f i x*. *op-mtx-set f* (*i div M*, *i mod M*) *x* :: ⟨*A*⟩*mtx-rel* → *nat-rel* → *A* → ⟨*A*⟩*mtx-rel*
⟨*proof*⟩

lemma *op-amtx-lin-get-aref*: (*uncurry Array.nth*, *uncurry* (*RETURN oo PR-CONST*

op-amtx-lin-get)) ∈ [λ(-,*i*). *i*<*N*∗*M*]$_a$ (*is-amtx N M*)^k ∗$_a$ *nat-assn*^{k} → *id-assn*
⟨*proof*⟩

sepref-decl-impl *amtx-lin-get*: *op-amtx-lin-get-aref* ⟨*proof*⟩

lemma *op-amtx-lin-set-aref*: (*uncurry2* (λ*m i x*. *Array.upd i x m*), *uncurry2* (*RETURN ooo PR-CONST op-amtx-lin-set*)) ∈ [λ((-,*i*),-). *i*<*N*∗*M*]$_a$ (*is-amtx N M*)^d ∗$_a$ *nat-assn*^{k} ∗$_a$ *id-assn*^{k} → *is-amtx N M*
⟨*proof*⟩

sepref-decl-impl *amtx-lin-set*: *op-amtx-lin-set-aref* ⟨*proof*⟩
end

lemma *amtx-fold-lin-get*: *m* (*i div M*, *i mod M*) = *op-amtx-lin-get M m i* ⟨*proof*⟩
lemma *amtx-fold-lin-set*: *m* ((*i div M*, *i mod M*) := *x*) = *op-amtx-lin-set M m i x* ⟨*proof*⟩

locale *amtx-pointwise-unop-impl* = *mtx-pointwise-unop-loc* +
fixes *A* :: *'a* ⇒ *'ai*::{*zero*,*heap*} ⇒ *assn*
fixes *fi* :: *nat*×*nat* ⇒ *'ai* ⇒ *'ai Heap*
assumes *fi-hnr*:
(*uncurry fi*,*uncurry* (*RETURN oo f*)) ∈ (*prod-assn nat-assn nat-assn*)^k ∗$_a$ *A*^{k} →$_a$ *A*
begin

lemma *this-loc*: *amtx-pointwise-unop-impl N M f A fi* ⟨*proof*⟩

context
assumes *PURE*: *CONSTRAINT* (*IS-PURE PRES-ZERO-UNIQUE*) *A*
begin
context
notes [[*sepref-register-adhoc f N M*]]
notes [*sepref-import-param*] = *IdI*[*of N*] *IdI*[*of M*]
notes [*sepref-fr-rules*] = *fi-hnr*
notes [*safe-constraint-rules*] = *PURE*
notes [*simp*] = *algebra-simps*
begin
sepref-thm *opr-fold-impl1* **is** *RETURN o opr-fold-impl* :: (*amtx-assn N M A*)^d →$_a$ *amtx-assn N M A*
⟨*proof*⟩
end
end
concrete-definition (**in** −) *amtx-pointwise-unnop-fold-impl1* **uses** *amtx-pointwise-unop-impl.opr-fold-impl*
prepare-code-thms (**in** −) *amtx-pointwise-unnop-fold-impl1-def*

lemma *op-hnr*[*sepref-fr-rules*]:
assumes *PURE*: *CONSTRAINT* (*IS-PURE PRES-ZERO-UNIQUE*) *A*

shows (*amtx-pointwise-unnop-fold-impl1 N M fi*, *RETURN* ∘ *PR-CONST* (*mtx-pointwise-unop f*)) ∈ (*amtx-assn N M A*)^d →$_a$ *amtx-assn N M A*
⟨*proof*⟩
end

locale *amtx-pointwise-binop-impl* = *mtx-pointwise-binop-loc* +
fixes *A* :: *'a* ⇒ *'ai*::{*zero*,*heap*} ⇒ *assn*
fixes *fi* :: *'ai* ⇒ *'ai* ⇒ *'ai Heap*
assumes *fi-hnr*: (*uncurry fi*,*uncurry* (*RETURN oo f*)) ∈ A^k *∗*$_a$ A^k →$_a$ *A*
begin

lemma *this-loc*: *amtx-pointwise-binop-impl f A fi*
⟨*proof*⟩

context
notes [[*sepref-register-adhoc f N M*]]
notes [*sepref-import-param*] = *IdI*[*of N*] *IdI*[*of M*]
notes [*sepref-fr-rules*] = *fi-hnr*
assumes *PURE*[*safe-constraint-rules*]: *CONSTRAINT* (*IS-PURE PRES-ZERO-UNIQUE*) *A*
notes [*simp*] = *algebra-simps*
begin
sepref-thm *opr-fold-impl1* **is** *uncurry* (*RETURN oo opr-fold-impl*) :: (*amtx-assn N M A*)^d*∗*$_a$(*amtx-assn N M A*)^k →$_a$ *amtx-assn N M A*
⟨*proof*⟩

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*,-)∈-
prepare-code-thms (**in** −) *amtx-pointwise-binop-fold-impl1-def*

lemma *op-hnr*[*sepref-fr-rules*]:
assumes *PURE*: *CONSTRAINT* (*IS-PURE PRES-ZERO-UNIQUE*) *A*
shows (*uncurry* (*amtx-pointwise-binop-fold-impl1 fi N M*), *uncurry* (*RETURN oo PR-CONST* (*mtx-pointwise-binop f*))) ∈ (*amtx-assn N M A*)^d *∗*$_a$ (*amtx-assn N M A*)^k →$_a$ *amtx-assn N M A*
⟨*proof*⟩

end

locale *amtx-pointwise-cmpop-impl* = *mtx-pointwise-cmpop-loc* +
fixes *A* :: *'a* ⇒ *'ai*::{*zero*,*heap*} ⇒ *assn*
fixes *fi* :: *'ai* ⇒ *'ai* ⇒ *bool Heap*
fixes *gi* :: *'ai* ⇒ *'ai* ⇒ *bool Heap*
assumes *fi-hnr*:
(*uncurry fi*,*uncurry* (*RETURN oo f*)) ∈ A^k *∗*$_a$ A^k →$_a$ *bool-assn*
assumes *gi-hnr*:

(*uncurry gi*,*uncurry* (*RETURN oo g*)) ∈ A^k $*_a$ A^k $\to_a$ *bool-assn*
begin

lemma *this-loc*: *amtx-pointwise-cmpop-impl f g A fi gi*
⟨*proof*⟩

context
notes [[*sepref-register-adhoc f g N M*]]
notes [*sepref-import-param*] = *IdI*[*of N*] *IdI*[*of M*]
assumes *PURE*[*safe-constraint-rules*]: *CONSTRAINT* (*IS-PURE PRES-ZERO-UNIQUE*) *A*
begin
sepref-thm *opr-fold-impl1* **is** *uncurry opr-fold-impl* :: (*amtx-assn N M A*)^d$*_a$(*amtx-assn N M A*)^k $\to_a$ *bool-assn*
⟨*proof*⟩
end

concrete-definition (**in** −) *amtx-pointwise-cmpop-fold-impl1* **for** *N M fi gi*
uses *amtx-pointwise-cmpop-impl.opr-fold-impl1.refine-raw* **is** (*uncurry ?f*,-)∈-
prepare-code-thms (**in** −) *amtx-pointwise-cmpop-fold-impl1-def*

lemma *op-hnr*[*sepref-fr-rules*]:
assumes *PURE*: *CONSTRAINT* (*IS-PURE PRES-ZERO-UNIQUE*) *A*
shows (*uncurry* (*amtx-pointwise-cmpop-fold-impl1 N M fi gi*), *uncurry* (*RETURN oo PR-CONST* (*mtx-pointwise-cmpop f g*))) ∈ (*amtx-assn N M A*)^d $*_a$ (*amtx-assn N M A*)^k $\to_a$ *bool-assn*
⟨*proof*⟩

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 ⇒ 'a mtx ⇒ 'a mtx nres where
  scmul x m ≡ nfoldli [0..<N] (λ-. True) (λi m.
    nfoldli [0..<M] (λ-. True) (λj m. do {
      let mij = m(i,j);
      RETURN (m((i,j) := x * mij))
    }
    ) m
  ) m
```

After declaration of an implementation for multiplication, refinement is straightforward. Note that we use the fixed N in the refinement assertions.

```
private lemma times-param: ((*),(*)::'a⇒-) ∈ Id → Id → Id ⟨proof⟩

context
  notes [sepref-import-param] = times-param
begin
  sepref-definition scmul-impl
    is uncurry scmul :: (id-assn^k *_a (amtx-assn N M id-assn)^d →_a amtx-assn
N M id-assn)
    ⟨proof⟩
end
```

Initialization with default value

```
private definition init-test ≡ do {
  let m = op-amtx-dfltNxM 10 5 (0::nat);
  RETURN (m(1,2))
}
private sepref-definition init-test-impl is uncurry0 init-test :: unit-assn^k→_a 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 ≡ UNPRO-
TECT 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^k $\rightarrow_a$ *pure* (*nat-rel* $\times_r$ *nat-rel* $\rightarrow$ *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*^{k}$\rightarrow_a$*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-Sepl-Binding*
imports
Separation-Logic-Imperative-HOL.Imp-Map-Spec
Separation-Logic-Imperative-HOL.Imp-Set-Spec
Separation-Logic-Imperative-HOL.Imp-List-Spec

Separation-Logic-Imperative-HOL.Hash-Map-Impl
Separation-Logic-Imperative-HOL.Array-Map-Impl

Separation-Logic-Imperative-HOL.To-List-GA
Separation-Logic-Imperative-HOL.Hash-Set-Impl
Separation-Logic-Imperative-HOL.Array-Set-Impl

Separation-Logic-Imperative-HOL.Open-List
Separation-Logic-Imperative-HOL.Circ-List

../Intf/IICF-Map

../Intf/IICF-Set
../Intf/IICF-List

Collections.Locale-Code
begin

This theory binds collection data structures from the basic collection framework established in *AFP/Separation-Logic-Imperative-HOL* for usage with Sepref.

locale *imp-map-contains-key* = *imp-map* +
  constrains *is-map* :: ('*k* ⇀ '*v*) ⇒ '*m* ⇒ *assn*
  fixes *contains-key* :: '*k* ⇒ '*m* ⇒ *bool Heap*
  assumes *contains-key-rule*[*sep-heap-rules*]:
    <*is-map m p*> *contains-key k p* <λ*r*. *is-map m p* ∗ ↑(*r*⟷*k*∈*dom m*)>$_t$

locale *gen-contains-key-by-lookup* = *imp-map-lookup*
begin
  definition *contains-key k m* ≡ *do* {*r* ← *lookup k m*; *return* (¬*is-None r*)}

  sublocale *imp-map-contains-key is-map contains-key*
    ⟨*proof*⟩

end

locale *imp-list-tail* = *imp-list* +
  constrains *is-list* :: '*a list* ⇒ '*l* ⇒ *assn*
  fixes *tail* :: '*l* ⇒ '*l Heap*
  assumes *tail-rule*[*sep-heap-rules*]:
    l≠[] ⟹ <*is-list l p*> *tail p* <*is-list* (*tl l*)>$_t$

definition *os-head* :: '*a*::*heap os-list* ⇒ ('*a*) *Heap* **where**
  os-head p ≡ *case p of*
    None ⇒ *raise STR ''os-Head: Empty list''*
  | *Some p* ⇒ *do* { *m* ←!*p*; *return* (*val m*) }

primrec *os-tl* :: '*a*::*heap os-list* ⇒ ('*a os-list*) *Heap* **where**
  os-tl None = *raise STR ''os-tl: Empty list''*
| *os-tl* (*Some p*) = *do* { *m* ←!*p*; *return* (*next m*) }

interpretation *os*: *imp-list-head os-list os-head*
  ⟨*proof*⟩

interpretation *os*: *imp-list-tail os-list os-tl*
  ⟨*proof*⟩

definition *cs-is-empty* :: *'a::heap cs-list* ⇒ *bool Heap* **where**
cs-is-empty p ≡ *return* (*is-None p*)
interpretation *cs*: *imp-list-is-empty cs-list cs-is-empty*
⟨*proof*⟩

definition *cs-head* :: *'a::heap cs-list* ⇒ *'a Heap* **where**
cs-head p ≡ *case p of*
None ⇒ *raise STR ''cs-head: Empty list''*
| *Some p* ⇒ *do* { *n* ← !*p*; *return* (*val n*)}
interpretation *cs*: *imp-list-head cs-list cs-head*
⟨*proof*⟩

definition *cs-tail* :: *'a::heap cs-list* ⇒ *'a cs-list Heap* **where**
cs-tail p ≡ *do* { (-,*r*) ← *cs-pop p*; *return r* }
interpretation *cs*: *imp-list-tail cs-list cs-tail*
⟨*proof*⟩

lemma *is-hashmap-finite*[*simp*]: *h* ⊨ *is-hashmap m mi* ⟹ *finite* (*dom m*)
⟨*proof*⟩

lemma *is-hashset-finite*[*simp*]: *h* ⊨ *is-hashset s si* ⟹ *finite s*
⟨*proof*⟩

definition *ias-is-it s a si* ≡ λ(*a'*,*i*).
∃$_A$*l*. *a*↦$_a$*l* ∗ ↑(*a'*=*a* ∧ *s*=*ias-of-list l* ∧ (*i*=*length l* ∧ *si*={} ∨ *i*<*length l* ∧ *i*∈*s* ∧ *si*=*s* ∩ {*x*. *x*≥*i*}))

context begin
private function *first-memb* **where**
first-memb lmax a i = *do* {
if i<*lmax then do* {
x ← *Array.nth a i*;
if x then return i else first-memb lmax a (*Suc i*)
} *else*
return i
}
⟨*proof*⟩
termination ⟨*proof*⟩
declare *first-memb.simps*[*simp del*]

private lemma *first-memb-rl-aux*:
assumes *lmax* ≤ *length l i*≤*lmax*
shows

$< a \mapsto_a l >$
first-memb lmax a i
$<\lambda k.\ a \mapsto_a l * \uparrow(k \leq lmax \wedge (\forall j.\ i \leq j \wedge j < k \longrightarrow \neg l!j) \wedge i \leq k \wedge (k = lmax \vee l!k)) >$
⟨*proof*⟩ **lemma** *first-memb-rl*[*sep-heap-rules*]:
assumes $lmax \leq$ *length* l $i \leq lmax$
shows $< a \mapsto_a l >$
first-memb lmax a i
$<\lambda k.\ a \mapsto_a l * \uparrow(ias\text{-}of\text{-}list\ l \cap \{i..<k\} = \{\} \wedge i \leq k \wedge (k < lmax \wedge k \in ias\text{-}of\text{-}list\ l \vee k = lmax)\ ) >$
⟨*proof*⟩

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* $l \subseteq \{0..<length\ l\}$ ⟨*proof*⟩

end

interpretation *ias*: *imp-set-iterate is-ias ias-is-it ias-it-init ias-it-has-next ias-it-next*
⟨*proof*⟩

lemma *ias-of-list-finite*[*simp*, *intro*!]: *finite* (*ias-of-list* l)
⟨*proof*⟩

lemma *is-ias-finite*[*simp*]: $h \models$ *is-ias* S $x \Longrightarrow$ *finite* S
⟨*proof*⟩

lemma *to-list-ga-rec-rule*:
assumes *imp-set-iterate is-set is-it it-init it-has-next it-next*
assumes *imp-list-prepend is-list l-prepend*
assumes *FIN*: *finite it*

assumes *DIS*: *distinct l set l* ∩ *it* = {}
shows
< *is-it s si it iti* ∗ *is-list l li* >
to-list-ga-rec it-has-next it-next l-prepend iti li
< λ*r*. ∃$_A$*l*′. *is-set s si*
∗ *is-list l*′ *r*
∗ ↑(*distinct l*′ ∧ *set l*′ = *set l* ∪ *it*) >$_t$
⟨*proof*⟩
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
<*is-set s si*>
to-list-ga it-init it-has-next it-next
l-empty l-prepend si
<λ*r*. ∃$_A$*l*. *is-set s si* ∗ *is-list l r* ∗ *true* ∗ ↑(*distinct l* ∧ *set l* = *s*)>
⟨*proof*⟩

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* ⇀ ′*vi*) ⇒ ′*m* ⇒ *assn*
begin
definition *assn K V* ≡ *hr-comp is-map* (⟨*the-pure K*,*the-pure V*⟩*map-rel*)
lemmas [*fcomp-norm-unfold*] = *assn-def*[*symmetric*]
lemmas [*safe-constraint-rules*] = *CN-FALSEI*[*of is-pure assn K V* **for** *K V*]

end

locale *bind-map-empty* = *imp-map-empty* + *bind-map*
begin
lemma *empty-hnr-aux*: (*uncurry0 empty*,*uncurry0* (*RETURN op-map-empty*))
∈ *unit-assn*^{k} →$_a$ *is-map*
⟨*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*) ∈ *is-map*^{k} →$_a$ *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*)) ∈ *id-assn*^{k} *∗*$_a$ *id-assn*^{k} *∗*$_a$ *is-map*^{d} →$_a$ *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*)) ∈ *id-assn*^{k} *∗*$_a$ *is-map*^{d} →$_a$ *is-map*
⟨*proof*⟩

sepref-decl-impl *delete*: *delete-hnr-aux* ⟨*proof*⟩
end

locale *bind-map-lookup* = *imp-map-lookup* + *bind-map*
begin
lemma *lookup-hnr-aux*: (*uncurry lookup*,*uncurry* (*RETURN oo op-map-lookup*)) ∈ *id-assn*^{k} *∗*$_a$ *is-map*^{k} →$_a$ *id-assn*
⟨*proof*⟩

sepref-decl-impl *lookup*: *lookup-hnr-aux* ⟨*proof*⟩
end

locale *bind-map-contains-key* = *imp-map-contains-key* + *bind-map*
begin
lemma *contains-key-hnr-aux*: (*uncurry contains-key*,*uncurry* (*RETURN oo op-map-contains-key*)) ∈ *id-assn*^{k} *∗*$_a$ *is-map*^{k} →$_a$ *bool-assn*
⟨*proof*⟩

sepref-decl-impl *contains-key*: *contains-key-hnr-aux* ⟨*proof*⟩
end

Set

locale *bind-set* = *imp-set is-set* **for** *is-set* :: (*'ai set*) ⇒ *'m* ⇒ *assn* +
fixes *A* :: *'a* ⇒ *'ai* ⇒ *assn*
begin
definition *assn* ≡ *hr-comp is-set* (⟨*the-pure A*⟩*set-rel*)
lemmas [*safe-constraint-rules*] = *CN-FALSEI*[*of is-pure assn*]
end

locale *bind-set-setup* = *bind-set*
begin

lemmas [*fcomp-norm-unfold*] = *assn-def*[*symmetric*]
lemma *APA*: ⟦*PROP Q*; *CONSTRAINT is-pure A*⟧ ⟹ *PROP Q* ⟨*proof*⟩
lemma *APAlu*: ⟦*PROP Q*; *CONSTRAINT* (*IS-PURE IS-LEFT-UNIQUE*) *A*⟧
⟹ *PROP Q* ⟨*proof*⟩
lemma *APAru*: ⟦*PROP Q*; *CONSTRAINT* (*IS-PURE IS-RIGHT-UNIQUE*)
A⟧ ⟹ *PROP Q* ⟨*proof*⟩
lemma *APAbu*: ⟦*PROP Q*; *CONSTRAINT* (*IS-PURE IS-LEFT-UNIQUE*) *A*;
CONSTRAINT (*IS-PURE IS-RIGHT-UNIQUE*) *A*⟧ ⟹ *PROP Q* ⟨*proof*⟩

end

locale *bind-set-empty* = *imp-set-empty* + *bind-set*
begin
lemma *hnr-empty-aux*: (*uncurry0 empty*,*uncurry0* (*RETURN op-set-empty*))∈*unit-assn*^{k}
→$_a$ *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*^{k} →$_a$
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*^{k} $*_a$ *is-set*^{k} $\rightarrow_a$ *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*^{k} $*_a$ *is-set*^{d} $\rightarrow_a$ *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*^{k} $*_a$ *is-set*^{d} $\rightarrow_a$ *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

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* ⊨ *is-set S x* ⟹ *finite S*
begin
context begin
private lemma *is-imp-set-iterate*: *imp-set-iterate is-set is-it it-init it-has-next it-next* ⟨*proof*⟩ **lemma** *is-imp-list-empty*: *imp-list-empty* (*list-assn id-assn*) (*return* [])
⟨*proof*⟩ **lemma** *is-imp-list-prepend*: *imp-list-prepend* (*list-assn id-assn*) (*return oo List.Cons*)
⟨*proof*⟩

definition *to-list* ≡ *to-list-ga it-init it-has-next it-next* (*return* []) (*return oo List.Cons*)
private lemmas *tl-rl* = *to-list-ga-rule*[*OF is-imp-set-iterate is-imp-list-empty is-imp-list-prepend*, *folded to-list-def*]

private lemma *to-list-sorted1*: (*to-list*,*PR-CONST* (*it-to-sorted-list* (λ- -. *True*))) ∈ *is-set*^{k} →$_a$ *list-assn id-assn*
⟨*proof*⟩ **lemma** *to-list-sorted2*: ⟦
CONSTRAINT (*IS-PURE IS-LEFT-UNIQUE*) *A*;
CONSTRAINT (*IS-PURE IS-RIGHT-UNIQUE*) *A*⟧ ⟹
(*PR-CONST* (*it-to-sorted-list* (λ- -. *True*)), *PR-CONST* (*it-to-sorted-list* (λ- -. *True*))) ∈ ⟨*the-pure A*⟩*set-rel* → ⟨⟨*the-pure A*⟩*list-rel*⟩*nres-rel*
⟨*proof*⟩

lemmas *to-list-hnr* = *to-list-sorted1*[*FCOMP to-list-sorted2*, *folded assn-def*]

lemmas *to-list-is-to-sorted-list* = *IS-TO-SORTED-LISTI*[*OF to-list-hnr*]
lemma *to-list-gen*[*sepref-gen-algo-rules*]: ⟦*CONSTRAINT* (*IS-PURE IS-LEFT-UNIQUE*) *A*; *CONSTRAINT* (*IS-PURE IS-RIGHT-UNIQUE*) *A*⟧
⟹ *GEN-ALGO to-list* (*IS-TO-SORTED-LIST* (λ- -. *True*) (*bind-set.assn is-set A*) *A*)
⟨*proof*⟩

end

end

List

locale *bind-list* = *imp-list is-list* **for** *is-list* :: (*'ai list*) ⇒ *'m* ⇒ *assn* +
fixes *A* :: *'a* ⇒ *'ai* ⇒ *assn*
begin

definition *assn* ≡ *hr-comp is-list* (⟨*the-pure A*⟩*list-rel*)
lemmas [*safe-constraint-rules*] = *CN-FALSEI*[*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*))∈(*pure unit-rel*)^k →$_a$ *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*)∈(*is-list*)^k →$_a$ *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*))
∈(*is-list*)^d ∗$_a$ (*pure Id*)^k →$_a$ *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*))
∈(*pure Id*)^k ∗$_a$ (*is-list*)^d →$_a$ *is-list* ⟨*proof*⟩

lemmas *hnr*[*sepref-fr-rules*]
= *hnr-aux*[*FCOMP op-list-prepend.fref*,*of A*, *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*)
∈[λ*l*. *l*≠[]]$_a$ (*is-list*)^d → *pure Id* ⟨*proof*⟩

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*)
∈[λ*l*. *l*≠[]]$_a$ (*is-list*)^d → *is-list*
⟨*proof*⟩

lemmas *hnr*[*sepref-fr-rules*] = *hnr-aux*[*FCOMP op-list-tl.fref*,*of the-pure A*, *folded assn-def*]
lemmas *hnr-mop*[*sepref-fr-rules*] = *hnr*[*FCOMP mk-mop-rl1*[*OF mop-list-tl-alt*]]
end

locale *bind-list-rotate1* = *imp-list-rotate* + *bind-list*
begin
lemma *hnr-aux*: (*rotate*,*RETURN o op-list-rotate1*)
∈(*is-list*)^d →$_a$ *is-list*
⟨*proof*⟩

lemmas *hnr*[*sepref-fr-rules*] = *hnr-aux*[*FCOMP op-list-rotate1.fref*,*of the-pure A*, *folded assn-def*]
lemmas *hnr-mop*[*sepref-fr-rules*] = *hnr*[*FCOMP mk-mop-rl1-np*[*OF mop-list-rotate1-alt*]]
end

locale *bind-list-rev* = *imp-list-reverse* + *bind-list*
begin
lemma *hnr-aux*: (*reverse*,*RETURN o op-list-rev*)
∈(*is-list*)^d →$_a$ *is-list*
⟨*proof*⟩

lemmas *hnr*[*sepref-fr-rules*] = *hnr-aux*[*FCOMP op-list-rev.fref*,*of the-pure A*, *folded assn-def*]
lemmas *hnr-mop*[*sepref-fr-rules*] = *hnr*[*FCOMP mk-mop-rl1-np*[*OF mop-list-rev-alt*]]

end

3.15.2 Array Map (iam)

definition *op-iam-empty* ≡ *IICF-Map.op-map-empty*
interpretation *iam*: *bind-map-empty is-iam iam-new*
⟨*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*) ≡ *IICF-Map.op-map-empty*
lemma [*def-pat-rules*]: *op-iam-empty-sz*$*N* ≡ *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*)))) ∈ *unit-assn*^{k} →$_a$ *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* ≡ *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*
⟨*proof*⟩

lemmas [*sepref-fr-rules*] = *ias.hnr-op-empty*[*folded op-ias-empty-def*]

definition [*simp*]: *op-ias-empty-sz* (*N*::*nat*) ≡ *op-set-empty*
lemma [*def-pat-rules*]: *op-ias-empty-sz*$*N* ≡ *UNPROTECT* (*op-ias-empty-sz N*)
⟨*proof*⟩

interpretation *ias-sz*: *bind-set-empty is-ias ias-new-sz N* **for** *N A*
⟨*proof*⟩

interpretation *ias-sz*: *set-custom-empty ias-new-sz N PR-CONST* (*op-ias-empty-sz N*) **for** *A*
⟨*proof*⟩
lemma [*sepref-fr-rules*]:
(*uncurry0* (*ias-new-sz N*), *uncurry0* (*RETURN* (*PR-CONST* (*op-ias-empty-sz N*)))) ∈ *unit-assn*^{k} →$_a$ *ias.assn A*
⟨*proof*⟩

interpretation *ias*: *bind-set-member is-ias Array-Set-Impl.ias-memb* **for** *A*
⟨*proof*⟩

interpretation *ias*: *bind-set-insert is-ias Array-Set-Impl.ias-ins* **for** *A*
⟨*proof*⟩

interpretation *ias*: *bind-set-delete is-ias Array-Set-Impl.ias-delete* **for** *A*
⟨*proof*⟩

⟨*ML*⟩
interpretation *ias*: *bind-set-iterate is-ias ias-is-it ias-it-init ias-it-has-next ias-it-next* **for** *A*
⟨*proof*⟩
⟨*ML*⟩

3.15.4 Hash Map (hm)

interpretation *hm*: *bind-map-empty is-hashmap hm-new*
⟨*proof*⟩

definition *op-hm-empty* ≡ *IICF-Map.op-map-empty*
interpretation *hm*: *map-custom-empty op-hm-empty*
⟨*proof*⟩
lemmas [*sepref-fr-rules*] = *hm.empty-hnr*[*folded op-hm-empty-def*]

interpretation *hm*: *bind-map-is-empty is-hashmap Hash-Map.hm-isEmpty*
⟨*proof*⟩

interpretation *hm*: *bind-map-update is-hashmap Hash-Map.hm-update*
⟨*proof*⟩

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* ≡ *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*⟩

⟨*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*⟩

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* ≡ *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* ≡ *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 *csll-tl*: *bind-list-tl cs-list cs-tail* **for** *A*
⟨*proof*⟩

interpretation *csll-rotate1*: *bind-list-rotate1 cs-list cs-rotate* **for** *A*
⟨*proof*⟩

schematic-goal *hn-refine* (*emp*) (*?c*::*?'c Heap*) *?Γ' ?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 *../IICF/IICF*
begin

4.1.1 Introduction

Sepref is an Isabelle/HOL tool to semi-automatically synthesize imperative code from abstract specifications.

The synthesis works by replacing operations on abstract data by operations on concrete data, leaving the structure of the program (mostly) unchanged. Speref 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 Isabelel/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-of-list* :: *'a::linorder list* ⇒ *'a nres* **where**
min-of-list l ≡ *ASSERT* (*l*≠[]) ≫ *SPEC* (λ*x*. ∀ *y*∈*set l*. *x*≤*y*)

This specification asserts the precondition and then specifies the valid results *x*. The ≫ 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* Φ is a syntax abbreviation for *SPEC* Φ).

Thus, *min-of-list* will fail if the list is empty, and otherwise nondeterministically return one of the minimal elements.

Abstract Algorithm

Next, we develop an abstract algorithm for the problem. A natural choice for a functional programmer is folding over the list, initializing the fold with the first element.

definition *min-of-list1* :: *'a::linorder list* ⇒ *'a nres*
where *min-of-list1 l* ≡ *ASSERT* (*l*≠[]) ≫ *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-of-list1-refine*: (*min-of-list1*,*min-of-list*) ∈ *Id* → ⟨*Id*⟩*nres-rel*

This lemma has to be read as follows: If the argument given to *min-of-list1* and *min-of-list* are related by *Id* (i.e. are identical), then the result of *min-of-list1* is a refinement of the result of *min-of-list*, wrt. relation *Id*.

For an explanation, lets simplify the statement first:

⟨*proof*⟩

A more concise proof of the same lemma omits the initial simplification, which we only inserted to explain the refinement ordering:

lemma (*min-of-list1*,*min-of-list*) ∈ *Id* → ⟨*Id*⟩*nres-rel*
⟨*proof*⟩

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* ⇒ *'a nres*
where *min-of-list2 l* ≡ *ASSERT* (*l*≠[]) ≫ *RETURN* (*fold* (λ*i*. *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*)∈*Id* → ⟨*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*)^k →$_a$ *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 nat-assn*. Here, *array-assn A* relates arrays with lists, such that the elements are related A — in our case by *nat-assn*, which relates natural numbers to themselves. We also say that we *implement* lists of nats by arrays of nats. The result is also implemented by natural numbers.

Moreover, the parameters may be stored on the heap, and we have to indicate whether the function keeps them intact or not. Here, we use the annotation -^k (for *keep*) to indicate that the parameter is kept intact, and -^d (for *destroy*) to indicate that it is destroyed.

Overall Correctness Statement

Finally, we can use transitivity of refinement to link our implementation to the specification. The *FCOMP* attribute is able to compose refinement theorems:

theorem *min-of-list3-correct*: (*min-of-list3*,*min-of-list*) ∈ (*array-assn nat-assn*)^k →$_a$ *nat-assn*
⟨*proof*⟩

While the above statement is suited to re-use the algorithm within the sepref-framework, a more low-level correctness theorem can be stated using separation logic. This has the advantage that understanding the statement depends on less definitional overhead:

lemma $l{\neq}[] \Longrightarrow$ <*array-assn nat-assn l a*> *min-of-list3 a* <λx. *array-assn nat-assn l a* $*$ ↑($\forall$ *y*∈*set l*. *x*≤*y*)>$_t$

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:

⟨*proof*⟩

If the automatic method does not work, here is a more explicit proof, that can be adapted for proving similar statements:

lemma $l{\neq}[] \Longrightarrow$ <*array-assn nat-assn l a*> *min-of-list3 a* <λx. *array-assn nat-assn l a* $*$ ↑($\forall$ *y*∈*set l*. *x*≤*y*)>$_t$
⟨*proof*⟩

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 2^32 in s* # *rand-list-aux s n*)
definition *rand-list* ≡ *rand-list-aux 42*

definition *min-of-rand-list n* = *min-of-list* (*rand-list n*)

And use Sepref to synthesize a concrete version

We use a feature of Sepref to combine imperative and purely functional code, and leave the generation of the list purely functional, then copy it into an array, and invoke our algorithm. We have to declare the *rand-list* operation:

sepref-register *rand-list*
lemma [*sepref-import-param*]: (*rand-list*,*rand-list*)∈*nat-rel* → ⟨*nat-rel*⟩*list-rel* ⟨*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* :: $nat\text{-}assn^k \rightarrow_a nat\text{-}assn$
⟨*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* ≡ *ASSERT* (*sorted xs*) ≫ *RETURN* (*x*∈*set xs*)

And give a standard iterative implementation:

definition *in-sorted-list1-invar x xs* ≡ λ(*l*,*u*,*found*).
  (*l*≤*u* ∧ *u*≤*length xs*)
∧ (*found* ⟶ *x*∈*set xs*)
∧ (¬*found* ⟶ (*x*∉*set* (*take l xs*) ∧ *x*∉*set* (*drop u xs*))
)

definition *in-sorted-list1 x xs* ≡ *do* {
  let l=*0*;
  let u=*length xs*;
  (-,-,*r*) ← *WHILEIT* (*in-sorted-list1-invar x xs*)
    (λ(*l*,*u*,*found*). *l*<*u* ∧ ¬*found*) (λ(*l*,*u*,*found*). *do* {
      let i = (*l*+*u*) *div 2*;
      ASSERT (*i*<*length xs*); — Added here to help synthesis to prove precondition for array indexing
      let xi = *xs*!*i*;
      if x=*xi then*
        RETURN (*l*,*u*,*True*)
      else if x<*xi then*
        RETURN (*l*,*i*,*False*)
      else
        RETURN (*i*+*1*,*u*,*False*)

    }) (*l*,*u*,*False*);
  RETURN r
}

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 (*?*Φ ⟹ *?M* ≤ *?M′*) ⟹ *?M* ≤ *ASSERT ?*Φ ≫= (λ-. *?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*≠*b* ⟹ ¬(*a*<*b*) ⟹ *a*>*b* ⟨*proof*⟩ **lemma** *isl1-aux1*:
assumes *sorted xs*
assumes *i*<*length xs*
assumes *xs*!*i* < *x*
shows *x*∉*set* (*take i xs*)
⟨*proof*⟩ **lemma** *isl1-aux2*:
assumes *x* ∉ *set* (*take n xs*)
shows *x*∉*set* (*drop n xs*) ⟷ *x*∉*set xs*
⟨*proof*⟩

lemma *in-sorted-list1-refine*: (*in-sorted-list1* , *in-sorted-list*)∈*Id* → *Id* → ⟨*Id*⟩*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 λ*f*. *uncurry2* (*uncurry2* (*uncurry f*)). If a function has no parameters, *uncurry0* adds a unit parameter.

sepref-definition *in-sorted-list2* **is** *uncurry in-sorted-list1* :: *nat-assn*^{k} *∗*$_a$ (*array-assn nat-assn*)^k →$_a$ *bool-assn*
⟨*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 ≡ do {
  let l=0;
  let u=length xs;
  (-,-,r) ← WHILEIT (in-sorted-list1-invar x xs)
    (λ(l,u,found). l<u ∧ ¬found) (λ(l,u,found). do {
      let i = (l+u) div 2;
      let xi = xs!i; — It's not trivial to show that i is in range
      if x=xi then
        RETURN (l,u,True)
      else if x<xi then
        RETURN (l,i,False)
      else
        RETURN (i+1,u,False)

    }) (l,u,False);
  RETURN r
}
```

We try to synthesize the implementation. Note that **sepref-thm** behaves like **sepref-definition**, but actually defines no constant. It only generates a refinement theorem.

```
sepref-thm in-sorted-list2 is uncurry in-sorted-list1' :: nat-assn^k *_a (array-assn
nat-assn)^k →_a bool-assn
  ⟨proof⟩
```

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.
  ⟨proof⟩
```

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** $\lambda x.\ RETURN\ (2{+}x) :: nat\text{-}assn^k \to_a nat\text{-}assn$
⟨*proof*⟩

Translation Phase

In most cases, the translation phase will fail. Let's try the following refinement:

sepref-thm *test* **is** $\lambda l.\ RETURN\ (l!1 + 2) :: (array\text{-}assn\ nat\text{-}assn)^k \to_a nat\text{-}assn$

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.

⟨*proof*⟩

Inserting an assertion into the abstract program solves the problem:

sepref-thm *test* **is** $\lambda l.\ ASSERT\ (length\ l > 1) \gg RETURN\ (l!1 + 2) :: (array\text{-}assn\ nat\text{-}assn)^k \to_a nat\text{-}assn$
⟨*proof*⟩

Here is an example for an unimplemented operation:

sepref-thm *test* **is** $\lambda l.\ RETURN\ (Min\ (set\ l)) :: (array\text{-}assn\ nat\text{-}assn)^k \to_a nat\text{-}assn$
⟨*proof*⟩

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* ≡
ASSERT (*finite s*) ≫ *SPEC* (λ*m*. *dom m* = *s* ∧ *ran m* = {*0*..<*card s*})

We implement the function by iterating over the set, and building the map

definition *nat-seg-map1 s* ≡ *do* {
ASSERT (*finite s*);
(*m*,-) ← *FOREACHi* (λ*it* (*m*,*i*). *dom m* = *s*−*it* ∧ *ran m* = {*0*..<*i*} ∧ *i*=*card* (*s* − *it*))
s (λ*x* (*m*,*i*). *RETURN* (*m*(*x*↦*i*),*i*+*1*)) (*Map.empty*,*0*);
RETURN m
}

lemma *nat-seg-map1-refine*: (*nat-seg-map1*, *nat-seg-map*) ∈ *Id* → ⟨*Id*⟩*nres-rel*
⟨*proof*⟩

We use hashsets *hs.assn* and hashmaps (*hm.assn*).

sepref-definition *nat-seg-map2* **is** *nat-seg-map1* :: (*hs.assn id-assn*)^k →$_a$ *hm.assn id-assn nat-assn*
⟨*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*.

sepref-definition *nat-seg-map2* **is** *nat-seg-map1* :: (*hs.assn id-assn*)^k →$_a$ *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* ≡ *map* ((+) *1*) *l*

We might want to implement it as follows

definition *incr-list1 l* ≡ *fold* (λ*i l*. *l*[*i*:=*1* + *l*!*i*]) [*0*..<*length l*] *l*

lemma *incr-list1-refine*: (*incr-list1*, *incr-list*)∈*Id* → *Id*
⟨*proof*⟩

Trying to refine this reveals a problem:

sepref-thm *incr-list2* **is** *RETURN o incr-list1* :: $(\textit{array-assn nat-assn})^d \to_a \textit{array-assn nat-assn}$
⟨*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* ≡ *nfoldli*
[*0..<length l*]
(λ-. *True*)
(λ*i l*. *ASSERT* (*i*<*length l*) ≫ *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*) ∈ *Id* → ⟨*Id*⟩*nres-rel*
⟨*proof*⟩

sepref-definition *incr-list3* **is** *incr-list2* :: $(\textit{array-assn nat-assn})^d \to_a \textit{array-assn nat-assn}$
⟨*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* (λ*l i j*. *do* {

```
  ASSERT (i<length l ∧ j<length l);
  RETURN (l[i:=l!j, j:=l!i])
}) :: (array-assn id-assn)^d *_a nat-assn^k *_a nat-assn^k →_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 = l!i;
  RETURN (l[i:=l!j, j:=tmp])
}) :: (array-assn id-assn)^d *_a nat-assn^k *_a nat-assn^k →_a array-assn id-assn
  ⟨proof⟩
```

Note that also the argument must be marked as destroyed $()^d$. Otherwise, we get a similar error as above, but in a different phase:

```
sepref-thm swap-with-tmp is uncurry2 (λl i j. do {
  ASSERT (i<length l ∧ j<length l);
  let tmp = l!i;
  RETURN (l[i:=l!j, j:=tmp])
}) :: (array-assn id-assn)^k *_a nat-assn^k *_a nat-assn^k →_a array-assn id-assn
  ⟨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 $()^k$ 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, loosing the first component. Inside a case distinction, you cannot access the compound object.

  These limitations are somewhat relaxed for pure data types, which can always be restored.

- Most IICF data structures only support pure component types. Exceptions are HOL-lists, and the list-based set and multiset implementations *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-assn*^{k} →$_a$ *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-assn*^{k} →$_a$ *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-assn*^{k} →$_a$ *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-assn*^{k} →$_a$ *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-assn*^{k} →$_a$ *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-assn*^{k} →$_a$ *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 $\{0::'a..<N\}$, and drop duplicate numbers. The following abstract algorithm may work:

```
definition remdup l ≡ do {
  (s,r) ← nfoldli l (λ-. True)
    (λx (s,r). do {
      ASSERT (distinct r ∧ set r ⊆ set l ∧ s = set r); — Will be required to prove
       that list does not grow too long
      if x∈s then RETURN (s,r) else RETURN (insert x s, r@[x])
    })
    ({},[]);
  RETURN r
}
```

We want to use *remdup* in our abstract code, so we have to register it.

sepref-register *remdup*

The straightforward version with dynamic data-structures is:

sepref-definition *remdup1* **is** *remdup* :: $(\textit{list-assn nat-assn})^k \to_a \textit{arl-assn nat-assn}$
  ⟨*proof*⟩

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**

sepref-definition *remdup1-initsz* **is** *remdup* :: $(\textit{list-assn nat-assn})^k \to_a \textit{arl-assn nat-assn}$
  ⟨*proof*⟩

end

To get a usable function, we may add the fixed N as a parameter, effectively converting the initialization hint to a parameter, which, however, has no abstract meaning

definition *remdup-initsz* (N::*nat*) ≡ *remdup*
lemma *remdup-init-hnr*:
  $(\textit{uncurry remdup1-initsz}, \textit{uncurry remdup-initsz}) \in \textit{nat-assn}^k *_a (\textit{list-assn nat-assn})^k \to_a \textit{arl-assn nat-assn}$
  ⟨*proof*⟩

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* ⟨*proof*⟩

Note that this will often require to use N as a usual constant, which is refined. For pure refinements, we can use the *sepref-import-param* attribute, which will convert a parametricity theorem to a rule for Sepref:

  sepref-register N
  lemma *N-hnr*[*sepref-import-param*]: (N,N)∈*nat-rel* ⟨*proof*⟩
  thm *N-hnr*

Alternatively, we could directly prove the following rule, which, however, is more cumbersome:

  lemma *N-hnr'*: (*uncurry0* (*return N*), *uncurry0* (*RETURN N*))∈*unit-assn*^{k} →$_a$ *nat-assn*
    ⟨*proof*⟩

Next, we use an array-list with a fixed maximum capacity. Note that the capacity is part of the refinement assertion now.

  sepref-definition *remdup1-fixed* **is** *remdup* :: (*list-assn nat-assn*)^k →$_a$ *marl-assn N nat-assn*
    ⟨*proof*⟩

Moreover, we add a precondition on the list

  sepref-definition *remdup1-fixed* **is** *remdup* :: [λ*l*. *set l* ⊆ {*0*..<*N*}]$_a$ (*list-assn nat-assn*)^k → *marl-assn N nat-assn*
    ⟨*proof*⟩

We can prove the remaining subgoal, e.g., by *auto* with the following lemma declared as introduction rule:

  lemma *aux1*[*intro*]: ⟦ *set l* ⊂ {*0*..<*N*}; *distinct l* ⟧ ⟹ *length l* < *N*
    ⟨*proof*⟩

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 :: [λl. set l ⊆ {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-assn^k →_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*) ∈ $[$*my-remdup-impl-loc*$]_a$ *nat-assn*^{k} → *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*) ∈ $[\lambda N.$ *N*>*0*$]_a$ *nat-assn*^{k} → *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, *nbn-assn N* relates natural numbers less than *N* to themselves.

We will repeat the above development, using the bounded relation approach:

```
definition bremdup l ≡ do {
  (s,r) ← nfoldli l (λ-. True)
    (λx (s,r). do {
      ASSERT (distinct r ∧ s = set r); — Less assertions than last time
      if x∈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)∈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*]: $[\![$ *set l* $\subset \{0..<N\}$; *distinct l* $]\!] \Longrightarrow$ *length l* $<$ *N*
⟨*proof*⟩

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*
⟨*proof*⟩

concrete-definition (**in** −) *bremdup1-fixed* **uses** *my-bremdup-impl-loc.remdup1-fixed.refine-raw* **is** (*?f*,-)∈-
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* ← *remdup* [*0..<N*]; *RETURN* (*length l*) }
sepref-register *test-remdup*

This refinement depends on the (somewhat experimental) subtyping feature to convert from *nat-assn* to *elem-assn*, based on context information

sepref-thm *test-remdup1* **is**
uncurry0 (*PR-CONST test-remdup*) :: *unit-assn*^{k} $\rightarrow_a$ *nat-assn*
⟨*proof*⟩

concrete-definition (**in** −) *test-bremdup1* **uses** *my-bremdup-impl-loc.test-remdup1.refine-raw* **is** (*uncurry0 ?f*,-)∈-
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*) ∈ [*my-bremdup-impl-loc*]$_a$ *nat-assn*^{k} $\rightarrow$ *nat-assn*
⟨*proof*⟩

lemma *test-bremdup1-refine*: (*test-bremdup1*, *my-bremdup-impl-loc.test-remdup*) ∈ $[\lambda N.$ *N*>*0*$]_a$ *nat-assn*^{k} $\rightarrow$ *nat-assn*
⟨*proof*⟩

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 = *length* l; — Introduce a *let*, such that we have a single variable as size-init
  let l' = *op-arl-empty-sz* N;
  nfoldli l ($\lambda x.$ *True*) ($\lambda x\ s.$ *mop-list-append* $s\ x$) l'
} :: (*list-assn nat-assn*)^k $\rightarrow_a$ *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 $'a$ *mtx* (which is *nat* $\times$ *nat* $\Rightarrow$ $'a$) as the abstract representation. The (currently only) implementation is by arrays, mapping points at coordinates out of range to $0::'a$.

Pointwise unary operations are those that modify every point of a matrix independently. Moreover, a zero-value must be mapped to a zero-value. As an example, we duplicate every value on the diagonal of a matrix

Abstractly, we apply the following function to every value. The first parameter are the coordinates.

definition *mtx-dup-diag-f*:: *nat*$\times$*nat* $\Rightarrow$ $'a$::{*numeral*,*times*,*mult-zero*} $\Rightarrow$ $'a$
  where *mtx-dup-diag-f* $\equiv \lambda(i,j)\ x.$ *if* i=j *then* x*(*2*) *else* x

We refine this function to a heap-function, using the identity mapping for values.

context
  fixes *dummy* :: $'a$::{*numeral*,*times*,*mult-zero*}
  notes [[*sepref-register-adhoc PR-CONST* (*2*::$'a$)]]
    — Note: The setup for numerals, like *2*, is a bit subtle in that numerals are always treated as constants, but have to be registered for any type they shall be used with. By default, they are only registered for *int* and *nat*.
  notes [*sepref-import-param*] = *IdI*[*of PR-CONST* (*2*::$'a$)]
  notes [*sepref-import-param*] = *IdI*[*of* (∗)::$'a$⇒-, *folded fun-rel-id-simp*]
begin

sepref-definition *mtx-dup-diag-f1* **is** *uncurry* (*RETURN oo* (*mtx-dup-diag-f*::-⇒$'a$⇒-))
:: (*prod-assn nat-assn nat-assn*)^k∗$_a$*id-assn*^{k} $\rightarrow_a$ *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* ≡ *mtx-pointwise-unop mtx-dup-diag-f*

The operation is usable now:

sepref-thm *mtx-dup-test* **is** λ*m*. *RETURN* (*mtx-dup-diag* (*mtx-dup-diag m*)) :: (*asmtx-assn N int-assn*)^d →$_a$ *asmtx-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*) ⇒ -) *id-assn return oo* ((+))
for *N M*
⟨*proof*⟩
abbreviation *mtx-add* ≡ *mtx-pointwise-binop* ((+))

sepref-thm *mtx-add-test* **is** *uncurry2* (λ*m1 m2 m3*. *RETURN* (*mtx-add m1* (*mtx-add m2 m3*)))
:: (*amtx-assn N M int-assn*)^d *$_a$ (*amtx-assn N M int-assn*)^d *$_a$ (*amtx-assn N M int-assn*)^k →$_a$ *amtx-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** (λ*m*. *RETURN* (*mtx-add m m*))
:: (*amtx-assn N M int-assn*)^d →$_a$ *amtx-assn N M int-assn*
⟨*proof*⟩

Of course, you can always copy the matrix manually:

sepref-thm *mtx-dup-alt-test* **is** (λ*m*. *RETURN* (*mtx-add* (*op-mtx-copy m*) *m*))
:: (*amtx-assn N M int-assn*)^k →$_a$ *amtx-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* ((≤)::(-::*order*) ⇒ -) ((≠)::(-::*order*) ⇒ -) *id-assn return oo* (≤) *return oo* (≠)

for *N M*
⟨*proof*⟩
abbreviation *mtx-lt* ≡ *mtx-pointwise-cmpop* (≤) (≠)

sepref-thm *test-mtx-cmp* **is** (λ*m*. *do* { *RETURN* (*mtx-lt* (*op-amtx-dfltNxM N M 0*) *m*) }) :: (*amtx-assn N M int-assn*)^k →$_a$ *bool-assn*
⟨*proof*⟩

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* ≡ *prod-assn* (*nbn-assn N*) (*nbn-assn M*)

And one for integer matrices

abbreviation *mtx-assn* ≡ *amtx-assn N M int-assn*

definition *co-set-gen* ≡ *do* {
nfoldli [*0*..<*N*] (λ-. *True*) (λ*i*. *nfoldli* [*0*..<*M*] (λ-. *True*) (λ*j s*.
if max i j − *min i j* ≤ *1 then RETURN* (*insert* (*i*,*j*) *s*)
else RETURN s
)) {}
}

sepref-definition *co-set-gen1* **is** *uncurry0 co-set-gen* :: *unit-assn*^{k} →$_a$ *hs.assn co-assn*
⟨*proof*⟩

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} →$_a$ *hs.assn co-assn*
⟨*proof*⟩
lemmas [*sepref-fr-rules*] = *co-set-gen1.refine*

sepref-register *co-set-gen*

Now we can use the entries from the set as coordinates, without any worries about them being out of range

sepref-thm *co-set-use* **is** (λm. *do* {
  co ← *co-set-gen*;
  FOREACH co ($\lambda(i,j)$ *m*. *RETURN* (*m*((*i*,*j*) := *1*))) *m*
}) :: *mtx-assn*^{d} →$_a$ *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* (λl. *length l* > *0* ∧ *hd l*) :: (*list-assn bool-assn*)^k →$_a$ *bool-assn*
  ⟨*proof*⟩

sepref-thm *test-sc-eval* **is** *RETURN o* (λl. *length l* > *0* ∧ *hd l*) :: (*list-assn bool-assn*)^k →$_a$ *bool-assn*
  ⟨*proof*⟩

end

4.2 Reference Guide

theory *Sepref-Guide-Reference*
imports *../IICF/IICF*
begin

This guide contains a short reference of the most important Sepref commands, methods, and attributes, as well as a short description of the internal working, and troubleshooting information with examples.

Note: To get an impression how to actually use the Sepref-tool, read the quickstart guide first!

4.2.1 The Sepref Method

The *sepref* method is the central method of the tool. Given a schematic goal of the form *hn-refine* Γ *?c ?Γ′ ?R f*, it tries to synthesize terms for the schematics and prove the theorem. Note that the *?Γ′* and *?R* may also be fixed terms, in which case frame inference is used to match the generated assertions with the given ones. Γ 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*)∈$[P]_a$ $A \to R$ is accepted, where A describes the refinement and preservation of the arguments, and R the refinement of the result. f must be in uncurried form (i.e. have exactly one argument).

We give some very basic examples here. In practice, you would almost always use the higher-level commands **sepref-definition** and **sepref-register**.

In its most primitive form, the Sepref-tool is applied like this:

schematic-goal
  notes [*id-rules*] = *itypeI*[*of x TYPE*(*nat*)] *itypeI*[*of a TYPE*(*bool list*)]
  shows *hn-refine*
    (*hn-ctxt nat-assn x xi* ∗ *hn-ctxt* (*array-assn bool-assn*) *a ai*)
    (*?c*::*?′c Heap*) *?Γ′ ?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* (λ*x a*. *do* {*ASSERT* (*x*<*length a*); *RETURN* (*a*!*x*)}))
  ∈ *nat-assn*^{k} ∗$_a$ (*array-assn bool-assn*)^k →$_a$ *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 *hfref* 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* (λ- -. *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 λ- -. *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 $m::'k \Rightarrow 'v$ *option*. If one writes $m(k \mapsto v)$, this is a map update. However, in Isabelle/HOL maps are encoded as functions $'k \Rightarrow 'v$ *option*, and the map update is just syntactic sugar for *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-map*. The *id-rule* for map lookup is *op-map-lookup* $::_i$ $TYPE(?'a \Rightarrow (?'a, ?'b)$ *i-map* $\Rightarrow ?'b$ *option*$)$. Moreover, there is a rule to rewrite function application to map lookup (*?m* \$ *?k* $\equiv$ *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 $:::_i$ $TYPE('a)$).

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., $'k \Rightarrow 'v$ *option* with $('k, 'v)$ *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*
    ⟨*proof*⟩

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[1].

The monadify phase consists of a number of sub-phases. The method *sepref-dbg-monadify* executes the monadify phase, the method *sepref-dbg-monadify-keep* stops at a failing sub-phase and presents the internal goal state before the failing sub-phase.

Monadify: Arity

In the first sub-phase, the rules from *sepref-monadify-arity* are used to standardize the number of operands applied to a constant. The rules work by rewriting each constant to a lambda-expression with the desired number of arguments, and the using beta-reduction to account for already existing

[1] Using fractional permissions or some other more fine grained ownership model might lift this restriction in the future.

arguments. Also higher-order arguments can be enforced, for example, the rule for fold enforces three arguments, the function itself having two arguments ($fold \equiv \lambda x.\ (\#\lambda xa.\ (\#\lambda xb.\ (\#SP\ fold\ \$\ (\lambda xa.\ (\#\lambda xb.\ (\#x\ \$\ xa\ \$\ xb\#)\#))\ \$\ xa\ \$\ xb\#)\#)\#)$).

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. $EVAL\ \$\ (If\ \$\ ?b\ \$\ ?t\ \$\ ?e) \equiv (\gg\!=)\ \$\ (EVAL\ \$\ ?b)\ \$\ (\lambda x.\ (\#If\ \$\ x\ \$\ (EVAL\ \$\ ?t)\ \$\ (EVAL\ \$\ ?e)\#))$

$EVAL\ \$\ (case\text{-}list\ \$\ ?fn\ \$\ (\lambda x.\ (\#\lambda xa.\ (\#\ ?fc\ x\ xa\#)\#))\ \$\ ?l) \equiv (\gg\!=)\ \$\ (EVAL\ \$\ ?l)\ \$\ (\lambda x.\ (\#case\text{-}list\ \$\ (EVAL\ \$\ ?fn)\ \$\ (\lambda x.\ (\#\lambda xa.\ (\#EVAL\ \$\ ?fc\ x\ xa\#)\#))\ \$\ x\#))$

$EVAL\ \$\ (case\text{-}prod\ \$\ (\lambda x.\ (\#\lambda xa.\ (\#\ ?fp\ x\ xa\#)\#))\ \$\ ?p) \equiv (\gg\!=)\ \$\ (EVAL\ \$\ ?p)\ \$\ (\lambda x.\ (\#case\text{-}prod\ \$\ (\lambda x.\ (\#\lambda xa.\ (\#EVAL\ \$\ ?fp\ x\ xa\#)\#))\ \$\ x\#))$

$EVAL\ \$\ (case\text{-}option\ \$\ ?fn\ \$\ (\lambda x.\ (\#\ ?fs\ x\#))\ \$\ ?ov) \equiv (\gg\!=)\ \$\ (EVAL\ \$\ ?ov)\ \$\ (\lambda x.\ (\#case\text{-}option\ \$\ (EVAL\ \$\ ?fn)\ \$\ (\lambda x.\ (\#EVAL\ \$\ ?fs\ x\#))\ \$\ x\#))$

$EVAL\ \$\ (Let\ \$\ ?v\ \$\ (\lambda x.\ (\#\ ?f\ x\#))) \equiv (\gg\!=)\ \$\ (EVAL\ \$\ ?v)\ \$\ (\lambda x.\ (\#EVAL\ \$\ ?f\ x\#)))$. 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* ≡ *fold*
sepref-thm *my-fold-test* **is** λ*l*. *do* { *RETURN* (*my-fold* (λ*x y*. *x*+*y*∗*2*) *l* *0*)} ::
(*list-assn nat-assn*)^k→$_a$*nat-assn*
⟨*proof*⟩

Solution: Register appropriate arity and combinator-rules

lemma *my-fold-arity*[*sepref-monadify-arity*]: *my-fold* ≡ λ$_2$*f l s*. *SP my-fold*$(λ$_2$*x s*. *f*$*x*$*s*)$*l*$*s* ⟨*proof*⟩

The combinator-rule rewrites to the already existing and set up combinator *nfoldli*:

lemma *monadify-plain-my-fold*[*sepref-monadify-comb*]:
EVAL$(*my-fold*$(λ$_2$*x s*. *f x s*)$*l*$*s*) ≡ (≫=)$(*EVAL*$*l*)$(λ$_2$*l*. (≫=)$(*EVAL*$*s*)$(λ$_2$*s*. *nfoldli*$*l*$(λ$_2$-. *True*)$(λ$_2$*x s*. *EVAL*$(*f x s*))$*s*))
⟨*proof*⟩

sepref-thm *my-fold-test* **is** λ*l*. *do* { *RETURN* (*my-fold* (λ*x y*. *x*+*y*∗*2*) *l* *0*)} ::
(*list-assn nat-assn*)^k→$_a$*nat-assn*
⟨*proof*⟩

Monadify: Dup

The last three phases, *mark-params*, *dup*, *remove-pass* are to detect duplicate parameters, and insert *COPY* tags. The first phase, *mark-params*, adds *PASS* tags around all parameters. Parameters are bound variables and terms that have a refinement in the precondition.

The second phase detects duplicate parameters and inserts *COPY* tags to remove them. Finally, the last phase removes the *PASS* tags again.

The methods *sepref-dbg-monadify-mark-params*, *sepref-dbg-monadify-dup*, and *sepref-dbg-monadify-remove-pass* gives you access to these phases.

Monadify: Step-Through Example

We give an annotated example of the monadify phase. Note that the program utilizes a few features of monadify:

- The fold function is higher-order, and gets flattened
- The first argument to fold is eta-contracted. The missing argument is added.
- The multiplication uses the same argument twice. A copy-tag is inserted.

sepref-thm *monadify-step-thru-test* **is** $\lambda l.$ *do* {
  let i = *length* l;
  RETURN (*fold* ($\lambda x.$ (+) (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 ⟦*hn-refine* ?Γ ?*c* ?Γ′ ?*R* ?*a*; *CNV* ?*c* ?*c*′⟧ ⟹ *hn-refine* ?Γ ?*c*′ ?Γ′ ?*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* ≡ *hn-val R x xi* and *hn-invalid A x xi* ≡ *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* Γ Γ', which is used to restore invalidated arguments if possible. Finally, it generates an assumption of the form *vassn-tag* Γ', 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* $x_1 \ldots x_n$)). Deprecated. Used to instantiate a schematic variable such that it does not depend on any bound variables any more. Originally used to make goals more readable, we are considering of dropping this.

Constraints (*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 ($?P\ ?x \Longrightarrow CONSTRAINT\ ?P\ ?x$) 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 Γ' with those in Γ. 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 $?P \Longrightarrow_t ?P$

$?F \Longrightarrow_t ?F' \Longrightarrow ?F * \textit{hn-ctxt}\ ?A\ ?x\ ?y \Longrightarrow_t ?F' * \textit{hn-ctxt}\ ?A\ ?x\ ?y$

$?F \Longrightarrow_t ?F' \Longrightarrow ?F * \textit{hn-ctxt}\ ?A\ ?x\ ?y \Longrightarrow_t ?F'$

$?P \Longrightarrow_t \textit{emp}$. The matching pairs are resolved by repeatedly applying rules from $?P \Longrightarrow_t ?P$

$[\![?P \Longrightarrow_t ?P'; ?F \Longrightarrow_t ?F']\!] \Longrightarrow ?F * ?P \Longrightarrow_t ?F' * ?P'$

$\textit{hn-ctxt}\ ?R\ ?x\ ?y \Longrightarrow_t \textit{hn-invalid}\ ?R\ ?x\ ?y$

$\textit{hn-ctxt}\ ?R\ ?x\ ?y \Longrightarrow_t \textit{hn-ctxt}\ (\lambda\text{- -}.\ \textit{true})\ ?x\ ?y$

$\textit{CONSTRAINT is-pure}\ ?R \Longrightarrow \textit{hn-invalid}\ ?R\ ?x\ ?y \Longrightarrow_t \textit{hn-ctxt}\ ?R\ ?x\ ?y$

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 $?F \vee_A ?F \Longrightarrow_t ?F$

$[\![\textit{hn-ctxt}\ ?R1.0\ ?x\ ?x' \vee_A \textit{hn-ctxt}\ ?R2.0\ ?x\ ?x' \Longrightarrow_t \textit{hn-ctxt}\ ?R\ ?x\ ?x'; ?Fl \vee_A ?Fr \Longrightarrow_t ?F]\!] \Longrightarrow ?Fl * \textit{hn-ctxt}\ ?R1.0\ ?x\ ?x' \vee_A ?Fr * \textit{hn-ctxt}\ ?R2.0\ ?x\ ?x' \Longrightarrow_t ?F * \textit{hn-ctxt}\ ?R\ ?x\ ?x'$

$\textit{hn-invalid}\ ?R\ ?x\ ?x' \vee_A \textit{hn-ctxt}\ ?R\ ?x\ ?x' \Longrightarrow_t \textit{hn-invalid}\ ?R\ ?x\ ?x'$

$\textit{hn-ctxt}\ ?R\ ?x\ ?x' \vee_A \textit{hn-invalid}\ ?R\ ?x\ ?x' \Longrightarrow_t \textit{hn-invalid}\ ?R\ ?x\ ?x'$ 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: $(\bigwedge x\ x'.\ [\![x \in \mathit{set}\ ?l;\ x' \in \mathit{set}\ ?l]\!] \Longrightarrow \mathit{hn\text{-}ctxt}\ ?A\ x\ x' \Longrightarrow_t \mathit{hn\text{-}ctxt}\ ?A'\ x\ x') \Longrightarrow \mathit{hn\text{-}ctxt}\ (\mathit{list\text{-}assn}\ ?A)\ ?l\ ?l' \Longrightarrow_t \mathit{hn\text{-}ctxt}\ (\mathit{list\text{-}assn}\ ?A')\ ?l\ ?l'$. 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 ($[\![\mathit{hn\text{-}ctxt}\ (\mathit{b\text{-}assn}\ ?A\ ?P)\ ?x\ ?y \Longrightarrow_t \mathit{hn\text{-}ctxt}\ ?A'\ ?x\ ?y;\ [\![\mathit{vassn\text{-}tag}\ (\mathit{hn\text{-}ctxt}\ ?A\ ?x\ ?y);\ \mathit{vassn\text{-}tag}\ (\mathit{hn\text{-}ctxt}\ ?A'\ ?x\ ?y);\ ?P\ ?x]\!] \Longrightarrow ?P'\ ?x]\!] \Longrightarrow \mathit{hn\text{-}ctxt}\ (\mathit{b\text{-}assn}\ ?A\ ?P)\ ?x\ ?y \Longrightarrow_t \mathit{hn\text{-}ctxt}\ (\mathit{b\text{-}assn}\ ?A'\ ?P')\ ?x\ ?y$).

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 *nat-assn* restricted to indices that are in range: *nbn-assn N*. When inserting natural numbers into this list, one has to prove that they are actually in range (conversion from *nat-assn* to *nbn-assn*). Elements of the list can be used as natural numbers (conversion from *nbn-assn* to *nat-assn*). Additionally, the side condition solver can derive that the predicate holds on the abstract variable (via the *vassn-tag* inserted by the operator steps).

Translation: Annotated Example

context
  fixes *N*::*nat*
  notes [[*sepref-register-adhoc N*]]
  notes [*sepref-import-param*] = *IdI*[*of N*]
begin

This worked example utilizes the following features of the translation phase:

- We have a fold combinator, which gets translated by its combinator rule
- We add a type annotation which enforces converting the natural numbers inserted into the list being refined by *nbn-assn N*, i.e., smaller than *N*.
- We can only prove the numbers inserted into the list to be smaller than *N* because the combinator rule for *If* inserts congruence assumptions.

- By moving the elements from the list to the set, they get invalidated. However, as *nat-assn* is pure, they can be recovered later, allowing us to mark the list argument as read-only.

sepref-thm *filter-N-test* **is** $\lambda l.$ *RETURN* (*fold* ($\lambda x\ s.$
  if $x<N$ *then insert* (*ASSN-ANNOT* (*nbn-assn N*) *x*) *s else s*
) *l op-hs-empty*) :: (*list-assn nat-assn*)^k $\rightarrow_a$ *hs.assn* (*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** $\lambda n.$ *do* { *let r* = *fold* (+) [*1*..<*n*] *0*; *RETURN* (*n*∗*n*+*2*)
}
  :: *nat-assn*^{k} $\rightarrow_a$ *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 $P \Longrightarrow$ *hn-refine* Γ *c*

Γ' R a states that, under precondition P, for a heap described by Γ, the Imperative/HOL program c produces a heap described by Γ' 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:

- Γ must have the form *hn-ctxt* A_1 x_1 xi_1 $*$ ... $*$ *hn-ctxt* A_n x_n xi_n
- Γ' must have the form *hn-ctxt* B_1 x_1 xi_1 $*$ ... $*$ *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 $PC \Longrightarrow (uncurry_x\ f,\ uncurry_x\ g) \in [P]_a\ {A_1}^{k1} *_a \ldots *_a\ {A_n}^{kn} \to R$, where ki is k if the argument is preserved (kept) or d is it is destroyed. PC are preconditions of the rule that do not depend on the arguments, usually restrictions on the relations. P is a predicate on the single argument of g, representing the precondition that depends on the arguments.

Optionally, g may be of the form $RETURN\ o \ldots o\ g'$, in which case the rule applies to a plain function.

If there is no precondition, there is a shorter syntax: $Args \to_a R \equiv Args \to_a R$.

For example, consider *arl-swap-hnr* [*unfolded pre-list-swap-def*]. It reads *CONSTRAINT is-pure A* $\Longrightarrow$ (*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 $*_a$ *nat-assn*^{k} $*_a$ *nat-assn*$^k \to$ *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* $\to \langle A\rangle$*list-rel* $\to \langle A\rangle$*list-rel*), while lifting/transfer uses a predicate based notation (e.g. *rel-fun* (*list-all2 A*) (*rel-fun* (*list-all2 A*) (*list-all2 A*)) (@) (@)).

Currently, we only support the Autoref style, but provide a few lemmas that ease manual conversion from the Lifting/Transfer style.

Given a parametricity theorem, the attribute *sepref-param* converts it to a hfref theorem, the attribute *sepref-import-param* does the conversion and registers the result as operator rule. Relation variables are converted to assertion variables with an *is-pure* constraint.

The behaviour can be customized by *sepref-import-rewrite*, which contains rewrite rules applied in the last but one step of the conversion, before converting relation variables to assertion variables. These theorems can be used to convert relations to there corresponding assertions, e.g., *pure* $(\langle ?R\rangle$*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. $[\![?l \neq []; (?l', ?l) \in \langle ?A\rangle list\text{-}rel]\!] \Longrightarrow (hd\ ?l', hd\ ?l) \in ?A$). 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 $(uncurry_x\ f, uncurry_x\ g) \in [P]_f\ (...(R_1 \times_r R_2) \times_r ...) \times_r R_n) \to R$, and without precondition, we have $(...(R_1 \times_r R_2) \times_r ...) \times_r R_n) \to_f R$. 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

composition over products, such that composition can be done argument-wise. For this, we unfold with *fcomp-norm-unfold*, and then simplify with *fcomp-norm-simps*.

The *FCOMP* attribute tries to convert its arguments to hfref/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* ≡ *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 (′*k*, ′*v*) *i-map* represents maps, which are encoded as ′*k* ⇒ ′*v option* in HOL.

New interface types must be registered by the command **sepref-decl-intf**.

sepref-decl-intf (′*a*,′*b*) *i-my-intf* **is** ′*a*∗′*a* ⇒ ′*b option*
— Declares (′*a*, ′*b*) *i-my-intf* as new interface type, and registers it to correspond to ′*a* × ′*a* ⇒ ′*b option*. Note: For HOL, the interface type is just an arbitrary new type, which is not related to he corresponding HOL type.

sepref-decl-intf (′*a*,′*b*) *i-my-intf2* (**infix** ∗→$_i$ *0*) **is** ′*a*∗′*a* ⇒ ′*b option*
— There is also a version that declares infix-syntax for the interface type. In this case we have ′*a* ∗→$_i$ ′*b*. ′*a* ⇁ ′*b* Be aware of syntax space pollution, as the syntax for interface types and HOL types is the same.

4.2.5 Registration of Abstract Operations

Registering a new abstract operation requires some amount of setup, which is automated by the *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 there HOL-types. Note that the interface type must correspond to the HOL type of the registered term, otherwise, you'll get an error message.

If the term is not a single constant or variable, and does not already start with a *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* ≡ $m(n \mapsto n+1)$
definition *map-op2 m n* ≡ $m(n \mapsto n+2)$
definition *map-op3 m n* ≡ $m(n \mapsto n+3)$
definition *map-op-to-map* $(m::'a \rightharpoonup 'b) \equiv m$

context
  notes [*map-type-eqs*] = *map-type-eqI*[*of* *TYPE*($'a \rightharpoonup 'b$) *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* :: ($'a \rightharpoonup 'b$) ⇒ ($'a$,$'b$) *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* ≡ *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* ≡ *SPEC* (λ*i*. *i*<*length l* ∧ *l*!*i* = (*1*::*nat*))

sepref-register *select-a-one*
print-theorems
— Arity and mcomb theorem is generated

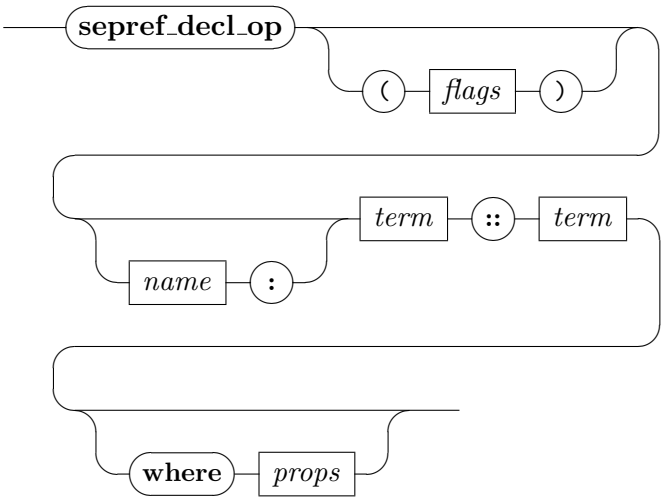
The following command fails, as the specified interface type does not correspond to the HOL type of the term: **sepref-register** *hd* :: (*nat*,*nat*) *i-map*

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.

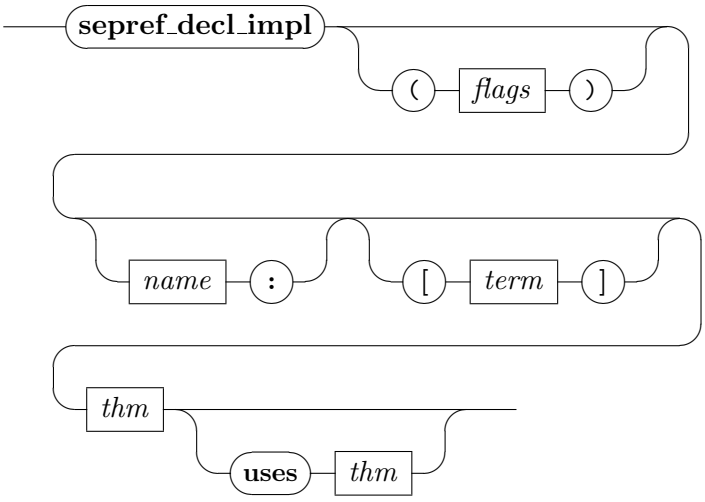
Finally, the following flags can be specified. Each flag can be prefixed by *no-* to invert its meaning:

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 specifiying 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 hfref-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 *rprems* resolves the current subgoal with one of its premises. It returns a sequence of possible resolvents. Optionally, the number of the premise to resolve with can be specified.

First-Order Resolution

The *fo-rule* applies a rule with first-order matching. It is very useful to be used with theorems like $?x = ?y \Longrightarrow ?f\ ?x = ?f\ ?y$.

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×ᵣA⟩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*∈*l*!*i* }

private definition *graph-rel1* ≡ *br graph-α1* (λ-. *True*)

private definition *succ1 l i* ≡ *if i*<*length l then l*!*i else* {}

private lemma *succ1-refine*: (*succ1*,*op-graph-succ*) ∈ *graph-rel1* → *Id* → ⟨*Id*⟩*set-rel*
⟨*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*)) :: (*assn2*^{k} $*_a$ *id-assn*^{k} $\rightarrow_a$ *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* ∘∘ *op-graph-succ*)) ∈ (*adjg-assn A*)^k $*_a$ *A*^{k} $\rightarrow_a$ *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 $E :: 'v\ rel$ **and** $v0 :: 'v$ **and** $tgt :: 'v \Rightarrow bool$
begin
definition $dfs :: ('v\ set \times bool)\ nres$ **where**
$dfs \equiv do \{$
$(V,r) \leftarrow RECT\ (\lambda dfs\ (V,v).$
$if\ v \in V\ then\ RETURN\ (V,False)$
$else\ do \{$
$let\ V = insert\ v\ V;$
$if\ tgt\ v\ then$
$RETURN\ (V,True)$
$else$
$FOREACH_C\ (E``\{v\})\ (\lambda(-,b).\ \neg b)\ (\lambda v'\ (V,-).\ dfs\ (V,v'))\ (V,False)$
$\}$
$)\ (\{\},v0);$
$RETURN\ (V,r)$
$\}$

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-correct*:
assumes *fr*: *finite reachable*
shows $dfs \leq dfs\text{-}spec$
⟨*proof*⟩

end

lemma *dfs-correct'*: (*uncurry2 dfs*, *uncurry2 dfs-spec*)
∈ [λ((*E*,*s*),*t*). *finite* (*reachable E s*)]$_f$ ((*Id*×$_r$*Id*)×$_r$*Id*) → ⟨*Id*⟩*nres-rel*
⟨*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*)^k*$_a$*nat-assn*^{k}*$_a$(*pure* (*nat-rel* → *bool-rel*))^k
→$_a$ *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
< λ(*Vi*,*r*). ∃$_A$*V*. *adjg-assn nat-assn E Ei* ∗ *ias.assn nat-assn V Vi* ∗ ↑((*r* ⟷ *reachable E s* ∩ *Collect tgt* ≠ {}) ∧ (¬*r* ⟶ *V*=*reachable E s*))>$_t$
⟨*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* ⟹ *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* ⟹ *IS-BELOW-ID* (⟨*R*⟩*infty-rel*)

⟨*proof*⟩

sepref-register *Num Infty*

lemma *Num-hnr*[*sepref-fr-rules*]: (*return o Num*,*RETURN o Num*)∈A^d →$_a$ *infty-assn A*
⟨*proof*⟩

lemma *Infty-hnr*[*sepref-fr-rules*]: (*uncurry0* (*return Infty*),*uncurry0* (*RETURN Infty*))∈*unit-assn*^{k} →$_a$ *infty-assn A*
⟨*proof*⟩

sepref-register *case-infty*
lemma [*sepref-monadify-arity*]: *case-infty* ≡ λ$_2$*f1 f2 x*. *SP case-infty*$*f1*$(λ$_2$*x*. *f2*$*x*)$*x*
⟨*proof*⟩
lemma [*sepref-monadify-comb*]: *case-infty*$*f1*$*f2*$*x* ≡ (≫=)$(*EVAL*$*x*)$(λ$_2$*x*. *SP*

case-infty$*f1*$*f2*$*x*) ⟨*proof*⟩
lemma [*sepref-monadify-comb*]: *EVAL*$(*case-infty*$*f1*$(λ$_2$*x*. *f2 x*)$*x*)
≡ (≫=)$(*EVAL*$*x*)$(λ$_2$*x*. *SP case-infty*$(*EVAL* $ *f1*)$(λ$_2$*x*. *EVAL* $ *f2 x*)$*x*)
⟨*proof*⟩

lemma *infty-assn-ctxt*: *infty-assn A x y* = *z* ⟹ *hn-ctxt* (*infty-assn A*) *x y* = *z*
⟨*proof*⟩

lemma *infty-cases-hnr*[*sepref-prep-comb-rule*, *sepref-comb-rules*]:
fixes *A e e*′
defines [*simp*]: *INVe* ≡ *hn-invalid* (*infty-assn A*) *e e*′
assumes *FR*: Γ ⟹$_t$ *hn-ctxt* (*infty-assn A*) *e e*′ ∗ *F*
assumes *Infty*: ⟦*e* = *Infty*; *e*′ = *Infty*⟧ ⟹ *hn-refine* (*hn-ctxt* (*infty-assn A*) *e e*′ ∗ *F*) *f1*′ (*hn-ctxt XX1 e e*′ ∗ Γ*1*′) *R f1*
assumes *Num*: ⋀*x1 x1a*. ⟦*e* = *Num x1*; *e*′ = *Num x1a*⟧ ⟹ *hn-refine* (*hn-ctxt A x1 x1a* ∗ *INVe* ∗ *F*) (*f2*′ *x1a*) (*hn-ctxt A*′ *x1 x1a* ∗ *hn-ctxt XX2 e e*′ ∗ Γ*2*′) *R* (*f2 x1*)
assumes *MERGE2*[*unfolded hn-ctxt-def*]: Γ*1*′ ∨$_A$ Γ*2*′ ⟹$_t$ Γ′
shows *hn-refine* Γ (*case-infty f1*′ *f2*′ *e*′) (*hn-ctxt* (*infty-assn A*′) *e e*′ ∗ Γ′) *R* (*case-infty*$*f1*$(λ$_2$*x*. *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*) ∈ [λ*x*. *x*≠*Infty*]$_a$ (*infty-assn A*)^d → *A*
⟨*proof*⟩

context
fixes *A* :: *'a*::*weight* ⇒ *'b* ⇒ *assn*
fixes *plusi*
assumes *GA*[*unfolded GEN-ALGO-def*, *sepref-fr-rules*]: *GEN-ALGO plusi* (λ*f*. (*uncurry f*,*uncurry* (*RETURN oo* (+)))∈*A*^{k} ∗$_a$ *A*^{k} →$_a$ *A*)
begin
sepref-thm *infty-plus-impl* **is** *uncurry* (*RETURN oo* (+)) :: ((*infty-assn A*)^k ∗$_a$ (*infty-assn A*)^k →$_a$ *infty-assn A*)
⟨*proof*⟩
end
concrete-definition *infty-plus-impl* **uses** *infty-plus-impl.refine-raw* **is** (*uncurry ?impl*,-)∈-

lemmas [*sepref-fr-rules*] = *infty-plus-impl.refine*

definition *infty-less* **where**
infty-less lt a b ≡ *case* (*a*,*b*) *of* (*Num a*, *Num b*) ⇒ *lt a b* | (*Num* -, *Infty*) ⇒ *True* | - ⇒ *False*

lemma *infty-less-param*[*param*]:
(*infty-less*,*infty-less*) ∈ (*R*→*R*→*bool-rel*) → ⟨*R*⟩*infty-rel* → ⟨*R*⟩*infty-rel* → *bool-rel*
⟨*proof*⟩

lemma *infty-less-eq-less*: *infty-less* (<) = (<)
⟨*proof*⟩

context
fixes *A* :: *'a*::*weight* ⇒ *'b* ⇒ *assn*
fixes *lessi*
assumes *GA*[*unfolded GEN-ALGO-def*, *sepref-fr-rules*]: *GEN-ALGO lessi* (λ*f*. (*uncurry f*,*uncurry* (*RETURN oo* (<)))∈A^k *a* A^k →*a* *bool-assn*)
begin
sepref-thm *infty-less-impl* **is** *uncurry* (*RETURN oo* (<)) :: ((*infty-assn A*)^k *a* (*infty-assn A*)^k →*a* *bool-assn*)
⟨*proof*⟩
end
concrete-definition *infty-less-impl* **uses** *infty-less-impl.refine-raw* **is** (*uncurry ?impl*,-)∈-
lemmas [*sepref-fr-rules*] = *infty-less-impl.refine*

lemma *param-mpath'*: (*mpath'*,*mpath'*)
∈ ⟨⟨*A*×*r B* ×*r A*⟩*list-rel* ×*r B*⟩*option-rel* → ⟨⟨*A*×*r B* ×*r A*⟩*list-rel*⟩*option-rel*
⟨*proof*⟩
lemmas (**in** −) [*sepref-import-param*] = *param-mpath'*

lemma *param-mpath-weight'*:
(*mpath-weight'*, *mpath-weight'*) ∈ ⟨⟨*A*×*r B*×*r A*⟩*list-rel* ×*r B*⟩*option-rel* → ⟨*B*⟩*infty-rel*
⟨*proof*⟩

lemmas [*sepref-import-param*] = *param-mpath-weight'*

context *Dijkstra* **begin**
lemmas *impl-aux* = *mdijkstra-def*[*unfolded mdinit-def mpop-min-def mupdate-def*]

lemma *mdijkstra-correct*:
(*mdijkstra*, *SPEC* (*is-shortest-path-map v0*)) ∈ ⟨*br* α*r res-invarm*⟩*nres-rel*
⟨*proof*⟩

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* ≡ *id-assn* :: $'W \Rightarrow$ -

lemma *w-plus-param*: ((+), (+)::$'W \Rightarrow$-) ∈ *Id* → *Id* → *Id* ⟨*proof*⟩
lemma *w-less-param*: ((<), (<)::$'W \Rightarrow$-) ∈ *Id* → *Id* → *Id* ⟨*proof*⟩
lemmas [*sepref-import-param*] = *w-plus-param w-less-param*
lemma [*sepref-gen-algo-rules*]:
GEN-ALGO (*return oo* (+)) (λ*f*. (*uncurry f*, *uncurry* (*RETURN* ∘∘ (+))) ∈ *id-assn*^{k} ∗$_a$ *id-assn*^{k} →$_a$ *id-assn*)
GEN-ALGO (*return oo* (<)) (λ*f*. (*uncurry f*, *uncurry* (*RETURN* ∘∘ (<))) ∈ *id-assn*^{k} ∗$_a$ *id-assn*^{k} →$_a$ *id-assn*)
⟨*proof*⟩

lemma *conv-prio-pop-min*: *prio-pop-min m* = *do* {
ASSERT (*dom m* ≠ {});
((*k*,*v*),*m*) ← *mop-pm-pop-min id m*;
RETURN (*k*,*v*,*m*)
}
⟨*proof*⟩
end

context fixes *N* :: *nat* **and** *w-dummy*::$'W$::{*heap*,*weight*} **begin**

interpretation *Dijkstra-Impl w-dummy* ⟨*proof*⟩

definition *drmap-assn2* ≡ *IICF-Sepl-Binding.iam.assn*
(*pure* (*node-rel N*))
(*prod-assn*
(*list-assn* (*prod-assn* (*pure* (*node-rel N*)) (*prod-assn weight-assn* (*pure* (*node-rel N*)))))
weight-assn)

concrete-definition *mdijkstra'* **uses** *Dijkstra.impl-aux*

sepref-definition *dijkstra-imp* **is** *uncurry mdijkstra'*
:: (*is-graph N* (*Id*::($'W \times 'W$) *set*))^k ∗$_a$ (*pure* (*node-rel N*))^k →$_a$ *drmap-assn2*
⟨*proof*⟩
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*))

$\in [\lambda(G,v0).\ Dijkstra\ G\ v0]_f\ Id\times_r Id \to \langle br\ Dijkstra.\alpha r\ Dijkstra.res\text{-}invarm\rangle nres\text{-}rel$
⟨*proof*⟩

definition *drmap-assn N* ≡ *hr-comp* (*drmap-assn2 N*) (*br Dijkstra.αr Dijkstra.res-invarm*)

context notes [*fcomp-norm-unfold*] = *drmap-assn-def*[*symmetric*] **begin**

theorem *dijkstra-imp-correct*: (*uncurry* (*dijkstra-imp N*), *uncurry* (*SPEC* ∘∘ *weighted-graph.is-shortest-path-m*
$\in [\lambda(G,\ v0).\ v0 \in nodes\ G \wedge (\forall (v,\ w,\ v') \in edges\ G.\ 0 \le w)]_a\ (is\text{-}graph\ N\ Id)^k$
$*_a\ (node\text{-}assn\ N)^k \to drmap\text{-}assn\ N$
⟨*proof*⟩

end

corollary *dijkstra-imp-rule*:
<*is-graph n Id G Gi* ∗ ↑(*v0* ∈ *nodes G* ∧ (∀ (*v*, *w*, *v'*) ∈ *edges G*. 0 ≤ *w*))>
dijkstra-imp n Gi v0
<λ*mi*. (*is-graph n Id*) *G Gi*
∗ (∃$_A$*m*. *drmap-assn n m mi* ∗ ↑(*weighted-graph.is-shortest-path-map G v0 m*)) >$_t$
⟨*proof*⟩

end

5.4 Imperative Implementation of of Nested DFS (HPY-Improvement)

theory *Sepref-NDFS*
imports
../Sepref
Collections-Examples.Nested-DFS
Sepref-Graph
HOL−Library.Code-Target-Numeral
begin

sepref-decl-intf *'v i-red-witness* **is** *'v list* ∗ *'v*

lemma *id-red-witness*[*id-rules*]:
red-init-witness ::$_i$ *TYPE*('*v* ⇒ '*v* ⇒ '*v i-red-witness option*)
prep-wit-red ::$_i$ *TYPE*('*v* ⇒ '*v i-red-witness option* ⇒ '*v i-red-witness option*)
⟨*proof*⟩

definition
red-witness-rel-def-internal: *red-witness-rel R* ≡ ⟨⟨*R*⟩*list-rel*,*R*⟩*prod-rel*

lemma *red-witness-rel-def*: ⟨*R*⟩*red-witness-rel* ≡ ⟨⟨*R*⟩*list-rel*,*R*⟩*prod-rel*
⟨*proof*⟩

lemma *red-witness-rel-sv*[*constraint-rules*]:
single-valued R $\Longrightarrow$ *single-valued* (⟨*R*⟩*red-witness-rel*)
⟨*proof*⟩

lemma [*sepref-fr-rules*]: *hn-refine*
(*hn-val R u u'* $*$ *hn-val R v v'*)
(*return* (*red-init-witness u' v'*))
(*hn-val R u u'* $*$ *hn-val R v v'*)
(*option-assn* (*pure* (⟨*R*⟩*red-witness-rel*)))
(*RETURN*$(*red-init-witness*$*u*$*v*))
⟨*proof*⟩

lemma [*sepref-fr-rules*]: *hn-refine*
(*hn-val R u u'* $*$ *hn-ctxt* (*option-assn* (*pure* (⟨*R*⟩*red-witness-rel*))) *w w'*)
(*return* (*prep-wit-red u' w'*))
(*hn-val R u u'* $*$ *hn-ctxt* (*option-assn* (*pure* (⟨*R*⟩*red-witness-rel*))) *w w'*)
(*option-assn* (*pure* (⟨*R*⟩*red-witness-rel*)))
(*RETURN*$(*prep-wit-red*$*u*$*w*))
⟨*proof*⟩

term *red-dfs*

sepref-definition *red-dfs-impl* **is**
(*uncurry2* (*uncurry red-dfs*))
:: (*adjg-assn nat-assn*)^k $*_a$ (*ias.assn nat-assn*)^k $*_a$ (*ias.assn nat-assn*)^d $*_a$ *nat-assn*^{k}
$\rightarrow_a$ *UNSPEC*
⟨*proof*⟩
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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *hn-init-wit-blue*[*sepref-fr-rules*]: *hn-refine*
(*hn-val nat-rel v v'* $*$ *hn-ctxt* (*option-assn* (*pure* (⟨*nat-rel*⟩*red-witness-rel*))) *w w'*)

(*return* (*init-wit-blue* v' w'))
(*hn-val nat-rel v* v' $*$ *hn-ctxt* (*option-assn* (*pure* (⟨*nat-rel*⟩*red-witness-rel*))) *w* w')
(*pure blue-wit-rel*)
(*RETURN*$(*init-wit-blue*$*v*$*w*))
⟨*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*)^k$*_a$(*ias.assn nat-assn*)^k$*_a$*nat-assn*^{k}$\to_a$*id-assn*)
⟨*proof*⟩
export-code *blue-dfs-impl* **checking** *SML-imp*

definition *blue-dfs-spec E A v0* ≡ *SPEC* (λ*r*. *case r of None* ⇒ ¬ *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*) ∈ [λ((*E*,*A*),*v0*). *finite* (*E** ‘‘{*v0*})]$_f$ ((*Id*×$_r$*Id*)×$_r$*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
<λ*r*. *ias.assn id-assn A A-impl* $*$ *adjg-assn id-assn E succ-impl*
$*$ ↑(
case r of None ⇒ ¬*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* :: $((adjg\text{-}assn\ nat\text{-}assn)^k *_a (ias.assn\ nat\text{-}assn)^k *_a nat\text{-}assn^k \rightarrow_a id\text{-}assn)$
⟨*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
<λ*r*. *ias.assn id-assn A A-impl* ∗ *adjg-assn id-assn E succ-impl*
∗ ↑(
case r of None ⇒ ¬*has-acc-cycle E A v0*
| *Some* (*v*,*pc*,*pv*) ⇒ *is-acc-cycle E A v0 v pv pc*
)>$_t$
⟨*proof*⟩

end
⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*ML*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*pro*

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* ≠ {}) ≫ *SPEC* (λ (*x*, *s'*). *x* ∈ *s* ∧ *s'* = *s* − {*x*})

lemma *take-from-set-correct*:
assumes *s* ≠ {}
shows *take-from-set s* ≤ *SPEC* (λ (*x*, *s'*). *x* ∈ *s* ∧ *s'* = *s* − {*x*})
⟨*proof*⟩

lemmas [*refine-vcg*] = *take-from-set-correct*[*THEN order.trans*]

definition *take-from-mset* **where**
take-from-mset s = *ASSERT* (*s* ≠ {#}) ≫ *SPEC* (λ (*x*, *s'*). *x* ∈# *s* ∧ *s'* = *s* − {#*x*#})

lemma *take-from-mset-correct*:
assumes *s* ≠ {#}
shows *take-from-mset s* ≤ *SPEC* (λ (*x*, *s'*). *x* ∈# *s* ∧ *s'* = *s* − {#*x*#})
⟨*proof*⟩

lemmas [*refine-vcg*] = *take-from-mset-correct*[*THEN order.trans*]

lemma *set-mset-mp*: *set-mset m* ⊆ *s* ⟹ *n* < *count m x* ⟹ *x*∈*s*
⟨*proof*⟩

lemma *pred-not-lt-is-zero*: (¬ *n* − *Suc 0* < *n*) ⟷ *n*=*0* ⟨*proof*⟩

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* ⇒ *'a* ⇒ *bool* — Step relation
and a_0 :: *'a* — Start state
and *F* :: *'a* ⇒ *bool* — Final states
and *subsumes* :: *'a* ⇒ *'a* ⇒ *bool* (**infix** ⪯ *50*) — Subsumption preorder
begin
definition *reachable* **where**
reachable = E^{**} a_0

definition *F-reachable* ≡ ∃ *a*. *reachable a* ∧ *F a*

end

The set of reachable states must be finite, subsumption must be a preorder, and be compatible with steps and final states.

locale *Search-Space* = *Search-Space-Defs* +
assumes *finite-reachable*: *finite* {*a*. *reachable a*}

assumes *refl*[*intro*!, *simp*]: *a* ⪯ *a*
and *trans*[*trans*]: *a* ⪯ *b* ⟹ *b* ⪯ *c* ⟹ *a* ⪯ *c*

assumes *mono*: *a* ⪯ *b* ⟹ *E a a'* ⟹ *reachable a* ⟹ *reachable b* ⟹ ∃ *b'*. *E b b'* ∧ *a'* ⪯ *b'*
and *F-mono*: *a* ⪯ *a'* ⟹ *F a* ⟹ *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*) (λ (*a*, *b*,*c*). (*a*,*b*))

definition *worklist-inv-frontier passed wait* =
(∀ *a* ∈ *passed*. ∀ *a′*. *E a a′* ⟶ (∃ *b′* ∈ *passed* ∪ *set-mset wait*. *a′* ≼ *b′*))

definition *start-subsumed passed wait* = (∃ *a* ∈ *passed* ∪ *set-mset wait*. a_0 ≼ *a*)

definition *worklist-inv* ≡ λ (*passed*, *wait*, *brk*).
passed ⊆ *Collect reachable* ∧
(*brk* ⟶ (∃ *f*. *reachable f* ∧ *F f*)) ∧
(¬ *brk* ⟶
worklist-inv-frontier passed wait
∧ (∀ *a* ∈ *passed* ∪ *set-mset wait*. ¬ *F a*)
∧ *start-subsumed passed wait*
∧ *set-mset wait* ⊆ *Collect reachable*)

definition *add-succ-spec wait a* ≡ *SPEC* (λ(*wait′*,*brk*).
if ∃ *a′*. *E a a′* ∧ *F a′ then*
brk
else set-mset wait′ = *set-mset wait* ∪ {*a′* . *E a a′*} ∧ ¬*brk*

```
  )

  definition worklist-algo where
    worklist-algo = do
      {
        if F a₀ then RETURN True
        else do {
          let passed = {};
          let wait = {#a₀#};
         (passed, wait, brk) ← WHILEIT worklist-inv (λ (passed, wait, brk). ¬ brk
∧ wait ≠ {#})
            (λ (passed, wait, brk). do
              {
                (a, wait) ← take-from-mset wait;
                ASSERT (reachable a);
                if (∃ a' ∈ passed. a ≼ a') then RETURN (passed, wait, brk) else
                do
                  {
                    (wait,brk) ← add-succ-spec wait a;
                    let passed = insert a passed;
                    RETURN (passed, wait, brk)
                  }
              }
            )
            (passed, wait, False);
            RETURN brk
        }
      }


end
```

Correctness Proof

```
context Search-Space begin

  lemma wf-worklist-var:
    wf worklist-var
  ⟨proof⟩

  context
  begin

  private lemma aux1:
    assumes ∀ x∈passed. ¬ a ≼ x
        and passed ⊆ Collect reachable
        and reachable a
    shows
    ((insert a passed, wait', brk'),
```

passed, *wait*, *brk*)
∈ *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-mset wait ⊆ *Collect reachable*
$a \in\#$ *wait*
set-mset wait' = *set-mset* (*wait* − {#*a*#}) ∪ *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-mset wait' = *set-mset* (*wait* − {#*a*#}) ∪ *Collect* (*E a*)
shows *start-subsumed* (*insert a passed*) *wait'*
⟨*proof*⟩

lemma *aux4*:
assumes *worklist-inv-frontier passed* {#} *reachable x start-subsumed passed* {#}
passed ⊆ *Collect reachable*
shows ∃ $x' \in$ *passed*. $x \preceq x'$
⟨*proof*⟩

theorem *worklist-algo-correct*:
worklist-algo ≤ *SPEC* (λ *brk*. *brk* ⟷ *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*) (λ(-,*brk*). $\neg$*brk*) (λ*a* (*wait*,*brk*). *if F a then RETURN* (*wait*,*True*) *else RETURN* (*wait* + {#*a*#},*False*)) (*wait*, *False*)

  lemma *add-succ1-ref*[*refine*]: $[\![$(*wait*,*wait'*)$\in$*Id*; (*a*,*a'*)$\in$*b-rel Id reachable*$]\!] \Longrightarrow$ *add-succ1 wait a* $\leq \Downarrow$(*Id* $\times_r$ *bool-rel*) (*add-succ-spec wait' a'*)
    ⟨*proof*⟩

  definition *worklist-algo1* **where**
    worklist-algo1 = *do*
      {
        if F a_0 *then RETURN True*
        else do {
          let passed = {};
          let wait = {#a_0#};
          (*passed*, *wait*, *brk*) ← *WHILEIT worklist-inv* (λ (*passed*, *wait*, *brk*). $\neg$ *brk* $\wedge$ *wait* $\neq$ {#})
            (λ (*passed*, *wait*, *brk*). *do*
              {
                (*a*, *wait*) ← *take-from-mset wait*;
                if ($\exists$ *a'* $\in$ *passed*. *a* $\preceq$ *a'*) *then RETURN* (*passed*, *wait*, *brk*) *else*
                do
                  {
                    (*wait*,*brk*) ← *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*
    ⟨*proof*⟩

end

end — Theory
theory *Worklist-Subsumption-Impl*
imports *../IICF/IICF Worklist-Subsumption*
begin

locale *Worklist2-Defs* = *Worklist1-Defs* +
fixes $A :: 'a \Rightarrow 'ai \Rightarrow assn$
fixes *succsi* :: $'ai \Rightarrow 'ai\ list\ Heap$
fixes $a_0 i :: 'ai\ Heap$
fixes *Fi* :: $'ai \Rightarrow bool\ Heap$
fixes *Lei* :: $'ai \Rightarrow 'ai \Rightarrow bool\ Heap$

locale *Worklist2* = *Worklist2-Defs* + *Worklist1* +

assumes [*sepref-fr-rules*]: (*uncurry0* $a_0 i$, *uncurry0* (*RETURN* (*PR-CONST* a_0))) $\in$ *unit-assn*$^k \rightarrow_a A$
assumes [*sepref-fr-rules*]: (*Fi*,*RETURN o PR-CONST F*) $\in A^k \rightarrow_a$ *bool-assn*
assumes [*sepref-fr-rules*]: (*uncurry Lei*,*uncurry* (*RETURN oo PR-CONST* ($\preceq$))) $\in A^k *_a A^k \rightarrow_a$ *bool-assn*
assumes [*sepref-fr-rules*]: (*succsi*,*RETURN o PR-CONST succs*) $\in A^k \rightarrow_a$ *list-assn A*
begin
sepref-register *PR-CONST* a_0 *PR-CONST F PR-CONST* ($\preceq$) *PR-CONST succs*

lemma [*def-pat-rules*]:
$a_0 \equiv$ *UNPROTECT* a_0 $F \equiv$ *UNPROTECT F* ($\preceq$) $\equiv$ *UNPROTECT* ($\preceq$) *succs* $\equiv$ *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* $\Longrightarrow$ *is-pure A* ⟨*proof*⟩

sepref-thm *worklist-algo2* **is** *uncurry0 worklist-algo1* :: *unit-assn*$^k \rightarrow_a$ *bool-assn*
⟨*proof*⟩

end

concrete-definition *worklist-algo2*
for *Lei* $a_0 i$ *Fi succsi*
uses *Worklist2.worklist-algo2.refine-raw* **is** (*uncurry0 ?f*,-)$\in$-
thm *worklist-algo2-def*

context *Worklist2* **begin**
lemma *Worklist2-this*: *Worklist2 E* a_0 *F* ($\preceq$) *succs A succsi* $a_0 i$ *Fi Lei*
⟨*proof*⟩

lemma *hnr-F-reachable*: (*uncurry0* (*worklist-algo2 Lei* $a_0 i$ *Fi succsi*), *uncurry0* (*RETURN F-reachable*))
$\in$ *unit-assn*^{k} $\rightarrow_a$ *bool-assn*
⟨*proof*⟩

end

context *Worklist1* **begin**
sepref-decl-op *F-reachable* :: *bool-rel* ⟨*proof*⟩
lemma [*def-pat-rules*]: *F-reachable* $\equiv$ *op-F-reachable* ⟨*proof*⟩

lemma *hnr-op-F-reachable*:
assumes *GEN-ALGO* $a_0 i$ ($\lambda a_0 i$. (*uncurry0* $a_0 i$, *uncurry0* (*RETURN* a_0)) $\in$ *unit-assn*^{k} $\rightarrow_a$ *A*)
assumes *GEN-ALGO Fi* (λ*Fi*. (*Fi*,*RETURN o F*) $\in$ A^k $\rightarrow_a$ *bool-assn*)
assumes *GEN-ALGO Lei* (λ*Lei*. (*uncurry Lei*,*uncurry* (*RETURN oo* ($\preceq$))) $\in$ A^k $*_a$ A^k $\rightarrow_a$ *bool-assn*)
assumes *GEN-ALGO succsi* (λ*succsi*. (*succsi*,*RETURN o succs*) $\in$ A^k $\rightarrow_a$ *list-assn A*)
shows (*uncurry0* (*worklist-algo2 Lei* $a_0 i$ *Fi succsi*), *uncurry0* (*RETURN* (*PR-CONST op-F-reachable*)))
$\in$ *unit-assn*^{k} $\rightarrow_a$ *bool-assn*
⟨*proof*⟩

sepref-decl-impl *hnr-op-F-reachable* ⟨*proof*⟩
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-pure A* $\Longrightarrow$ *is-pure* (*enum-assn A*)
⟨*proof*⟩

An identitiy rule is required to easily prove trivial refinement theorems

lemma *enum-assn-id*[*simp*]: *enum-assn id-assn* = *id-assn*
⟨*proof*⟩

Structural rules.

Without congruence condition

lemma *enum-match-nocong*: $[\![\bigwedge x\ y.$ *hn-ctxt A x y* $\Longrightarrow_t$ *hn-ctxt A' x y*$]\!] \Longrightarrow$ *hn-ctxt* (*enum-assn A*) *e e'* $\Longrightarrow_t$ *hn-ctxt* (*enum-assn A'*) *e e'*
⟨*proof*⟩

lemma *enum-merge-nocong*:
assumes $\bigwedge x\ y.$ *hn-ctxt A x y* $\vee_A$ *hn-ctxt A' x y* $\Longrightarrow_A$ *hn-ctxt Am x y*
shows *hn-ctxt* (*enum-assn A*) *e e'* $\vee_A$ *hn-ctxt* (*enum-assn A'*) *e e'* $\Longrightarrow_A$ *hn-ctxt* (*enum-assn Am*) *e e'*
⟨*proof*⟩

With congruence condition

lemma *enum-match-cong*[*sepref-frame-match-rules*]:
$[\![\bigwedge x\ y.\ [\![x \in$ *set-enum e*; $y \in$ *set-enum e'*$]\!] \Longrightarrow$ *hn-ctxt A x y* $\Longrightarrow_t$ *hn-ctxt A' x y*$]\!]$
$\Longrightarrow$ *hn-ctxt* (*enum-assn A*) *e e'* $\Longrightarrow_t$ *hn-ctxt* (*enum-assn A'*) *e e'*
⟨*proof*⟩

lemma *enum-merge-cong*[*sepref-frame-merge-rules*]:
assumes $\bigwedge x\ y.\ [\![x \in$ *set-enum e*; $y \in$ *set-enum e'*$]\!] \Longrightarrow$ *hn-ctxt A x y* $\vee_A$ *hn-ctxt A' x y* $\Longrightarrow_t$ *hn-ctxt Am x y*
shows *hn-ctxt* (*enum-assn A*) *e e'* $\vee_A$ *hn-ctxt* (*enum-assn A'*) *e e'* $\Longrightarrow_t$ *hn-ctxt* (*enum-assn Am*) *e e'*
⟨*proof*⟩

Propagating invalid

lemma *entt-invalid-enum*: *hn-invalid* (*enum-assn A*) *e e'* $\Longrightarrow_t$ *hn-ctxt* (*enum-assn* (*invalid-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*]: (*return o E1*,*RETURN o E1*) ∈ $A^d \to_a$ *enum-assn A*
⟨*proof*⟩
lemma [*sepref-fr-rules*]: (*return o E2*,*RETURN o E2*) ∈ $A^d \to_a$ *enum-assn A*
⟨*proof*⟩
lemma [*sepref-fr-rules*]: (*uncurry0* (*return E3*),*uncurry0* (*RETURN E3*)) ∈ *unit-assn*$^k \to_a$ *enum-assn A*
⟨*proof*⟩
lemma [*sepref-fr-rules*]: (*uncurry* (*return oo E4*),*uncurry* (*RETURN oo E4*)) ∈ $A^d *_a A^d \to_a$ *enum-assn A*
⟨*proof*⟩
lemma [*sepref-fr-rules*]: (*uncurry* (*return oo E5*),*uncurry* (*RETURN oo E5*)) ∈ *bool-assn*$^k *_a A^d \to_a$ *enum-assn A*
⟨*proof*⟩

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*]: *case-enum* ≡ λ$_2$*f1 f2 f3 f4 f5 x*. *SP case-enum*$(λ$_2$*x*. *f1*$*x*)$(λ$_2$*x*. *f2*$*x*)$*f3*$(λ$_2$*x y*. *f4*$*x*$*y*)$(λ$_2$*x y*. *f5*$*x*$*y*)$*x*
⟨*proof*⟩

This determines an evaluation order for the first-order operands

lemma [*sepref-monadify-comb*]: *case-enum*$*f1*$*f2*$*f3*$*f4*$*f5*$*x* ≡ (≫=)$(*EVAL*$*x*)$(λ$_2$*x*. *SP case-enum*$*f1*$*f2*$*f3*$*f4*$*f5*$*x*) ⟨*proof*⟩

This enables translation of the case-distinction in a non-monadic context.

lemma [*sepref-monadify-comb*]: *EVAL*$(*case-enum*$(λ$_2$*x*. *f1 x*)$(λ$_2$*x*. *f2 x*)$*f3*$(λ$_2$*x y*. *f4 x y*)$(λ$_2$*x y*. *f5 x y*)$*x*)
≡ (≫=)$(*EVAL*$*x*)$(λ$_2$*x*. *SP case-enum*$(λ$_2$*x*. *EVAL* $ *f1 x*)$(λ$_2$*x*. *EVAL* $ *f2 x*)$(*EVAL* $ *f3*)$(λ$_2$*x y*. *EVAL* $ *f4 x y*)$(λ$_2$*x y*. *EVAL* $ *f5 x y*)$*x*)
⟨*proof*⟩

Auxiliary lemma, to lift simp-rule over *hn-ctxt*

lemma *enum-assn-ctxt*: *enum-assn A x y* = *z* ⟹ *hn-ctxt* (*enum-assn A*) *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 *enum-cases-hnr*:
fixes *A e e'*
defines [*simp*]: *INVe* ≡ *hn-invalid* (*enum-assn A*) *e e'*
assumes *FR*: Γ ⟹$_t$ *hn-ctxt* (*enum-assn A*) *e e'* ∗ *F*
assumes *E1*: ⋀*x1 x1a*. ⟦*e* = *E1 x1*; *e'* = *E1 x1a*⟧ ⟹ *hn-refine* (*hn-ctxt A x1 x1a* ∗ *INVe* ∗ *F*) (*f1' x1a*) (*hn-ctxt A1' x1 x1a* ∗ *hn-ctxt XX1 e e'* ∗ Γ*1'*) *R* (*f1 x1*)
assumes *E2*: ⋀*x2 x2a*. ⟦*e* = *E2 x2*; *e'* = *E2 x2a*⟧ ⟹ *hn-refine* (*hn-ctxt A x2 x2a* ∗ *INVe* ∗ *F*) (*f2' x2a*) (*hn-ctxt A2' x2 x2a* ∗ *hn-ctxt XX2 e e'* ∗ Γ*2'*) *R* (*f2 x2*)
assumes *E3*: ⟦*e* = *E3*; *e'* = *E3*⟧ ⟹ *hn-refine* (*hn-ctxt* (*enum-assn A*) *e e'* ∗ *F*) *f3'* (*hn-ctxt XX3 e e'* ∗ Γ*3'*) *R f3*
assumes *E4*: ⋀*x41 x42 x41a x42a*.
⟦*e* = *E4 x41 x42*; *e'* = *E4 x41a x42a*⟧
⟹ *hn-refine* (*hn-ctxt A x41 x41a* ∗ *hn-ctxt A x42 x42a* ∗ *INVe* ∗ *F*) (*f4' x41a x42a*) (*hn-ctxt A4a' x41 x41a* ∗ *hn-ctxt A4b' x42 x42a* ∗ *hn-ctxt XX4 e e'* ∗ Γ*4'*) *R*
(*f4 x41 x42*)
assumes *E5*: ⋀*x51 x52 x51a x52a*.
⟦*e* = *E5 x51 x52*; *e'* = *E5 x51a x52a*⟧
⟹ *hn-refine* (*hn-ctxt bool-assn x51 x51a* ∗ *hn-ctxt A x52 x52a* ∗ *INVe* ∗ *F*) (*f5' x51a x52a*)
(*hn-ctxt bool-assn x51 x51a* ∗ *hn-ctxt A5' x52 x52a* ∗ *hn-ctxt XX5 e e'* ∗ Γ*5'*) *R* (*f5 x51 x52*)
assumes *MERGE1*[*unfolded hn-ctxt-def*]: ⋀*x x'*. *hn-ctxt A1' x x'* ∨$_A$ *hn-ctxt A2' x x'* ∨$_A$ *hn-ctxt A3' x x'* ∨$_A$ *hn-ctxt A4a' x x'* ∨$_A$ *hn-ctxt A4b' x x'* ∨$_A$ *hn-ctxt A5' x x'* ⟹$_t$ *hn-ctxt A' x x'*
assumes *MERGE2*[*unfolded hn-ctxt-def*]: Γ*1'* ∨$_A$ Γ*2'* ∨$_A$ Γ*3'* ∨$_A$ Γ*4'* ∨$_A$ Γ*5'* ⟹$_t$ Γ'
shows *hn-refine* Γ (*case-enum f1' f2' f3' f4' f5' e'*) (*hn-ctxt* (*enum-assn A'*) *e e'* ∗ Γ') *R* (*case-enum*$(λ$_2$x. *f1 x*)$(λ$_2$x. *f2 x*)$*f3*$(λ$_2$x *y*. *f4 x y*)$(λ$_2$x *y*. *f5 x y*)$*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*)
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*) ≡ *RETURN e*
sepref-definition *test1-impl* **is** *test1* :: (*enum-assn bool-assn*)^d →$_a$ *enum-assn bool-assn*
⟨*proof*⟩
sepref-register *test1*
lemmas [*sepref-fr-rules*] = *test1-impl.refine*

definition *test* ≡ *do* {
let x = *E1 True*;

- ← *case x of*
E1 a ⇒ *RETURN* (*Some a*) — Access and invalidate compound inside case
| - ⇒ *RETURN* (*Some True*);

- ← *test1 x*; — Rely on structure being there, with valid compound

— Same thing again, with merge
- ← *if True then*
case x of
E1 a ⇒ *RETURN* (*Some a*) — Access and invalidate compound inside case
| - ⇒ *RETURN* (*Some True*)
else RETURN None;
- ← *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*;

- ← *case x of*
E1 - ⇒ *RETURN* (*0*::*nat*)
| *E2* - ⇒ *RETURN 1*
| *E3* ⇒ *RETURN 0*
| *E4* - - ⇒ *RETURN 0*
| *E5* - *a* ⇒ *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 ⇒ *do* {*mop-list-set a 0 0*; *RETURN* (*0*::*nat*)}
| *E2* - ⇒ *RETURN 1*
| *E3* ⇒ *RETURN 0*
| *E4* - - ⇒ *RETURN 0*
| *E5* - - ⇒ *RETURN 0*
}

lemmas [*safe-constraint-rules*] = *CN-FALSEI*[*of is-pure invalid-assn A* **for** *A*]

sepref-definition *foo* **is** *uncurry0 test* :: $unit\text{-}assn^k \rightarrow_a nat\text{-}assn$
⟨*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 $f :: 'a \Rightarrow 'b\ nres$
fixes $l :: 'a\ list$

fixes $fi :: 'ai \Rightarrow 'bi\ Heap$
fixes $li :: 'ai\ list$

fixes $A\ A' :: 'a \Rightarrow 'ai \Rightarrow assn$ — Refinement for list elements before and after map-function. Different, as map function may invalidate list elements!
fixes $B :: 'b \Rightarrow 'bi \Rightarrow assn$

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* $(hn\text{-}ctxt\ A\ x\ xi * F)\ (fi\ xi)\ (hn\text{-}ctxt\ A'\ x\ xi * F)\ B\ (f\$x)$
— Refinement for f

begin

We implement our combinator using the monadic refinement framework.

definition $mmap \equiv RECT\ (\lambda mmap.$
$\lambda[] \Rightarrow RETURN\ []$
$| x\#xs \Rightarrow do\ \{\ x \leftarrow f\ x;\ xs \leftarrow mmap\ xs;\ RETURN\ (x\#xs)\ \})\ l$

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-assn* (*x*::*unit*) (*y*::*unit*) ≡ *F*
lemma *F-unf*: *hn-ctxt F-assn x y* = *F*
⟨*proof*⟩

We build a combinator rule to refine *f*. We need a combinator rule here, because *f* does not only depend on its formal arguments, but also on the frame (represented as dummy argument).

lemma *f-rl'*: *hn-refine* (*hn-ctxt A x xi* ∗ *hn-ctxt* (*F-assn*) *dx dxi*) (*fi xi*) (*hn-ctxt A' x xi* ∗ *hn-ctxt* (*F-assn*) *dx dxi*) *B* (*f*$*x*)
⟨*proof*⟩

Then we use the Sepref tool to synthesize an implementation of *mmap*.

schematic-goal *mmap-impl*:
notes [*sepref-comb-rules*] = *hn-refine-frame*[*OF f-rl'*]
shows *hn-refine* (*hn-ctxt* (*list-assn A*) *l li* ∗ *hn-ctxt* (*F-assn*) *dx dxi*) (*?c*::*?'c Heap*) *?*Γ*' ?R mmap*
⟨*proof*⟩

We unfold the wrapped frame

lemmas *mmap-impl'* = *mmap-impl*[*unfolded F-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-impl* **uses** *mmap-impl'*
prepare-code-thms *mmap-impl-def*

Moreover, we have to manually declare arity and monadify theorems. The arity theorem ensures that we always have a constant number of operators, and the monadify theorem determines an execution order: The list-argument is evaluated first.

lemma *mmap-arity*[*sepref-monadify-arity*]: *mmap* ≡ λ$_2$*f l*. *SP mmap*$(λ$_2$*x*. *f*$*x*)$*l*
⟨*proof*⟩
lemma *mmap-mcomb*[*sepref-monadify-comb*]: *mmap*$*f*$*x* ≡ (≫=)$(*EVAL*$*x*)$(λ$_2$*x*. *SP mmap*$*f*$*x*) ⟨*proof*⟩

We can massage the refinement theorem (⋀*x xi*. *hn-refine* (*hn-ctxt ?A x xi* ∗ *?F*) (*?fi xi*) (*hn-ctxt ?A' x xi* ∗ *?F*) *?B* (*?f* $ *x*)) ⟹ *hn-refine* (*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 ⋀*x xi*. *hn-refine* (*hn-ctxt A x xi* ∗ *F*) (*fi xi*) (*Q x xi*) *B* (*f x*)
— Refinement of map-function
and ⋀*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*$(λ$_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 λ*l*. *do* { *let a* = *op-array-of-list* [*True*,*True*,*False*]; *mmap* (λ*x*. *do* { *mop-list-get a* (*x mod 3*) }) *l* }
:: (*list-assn nat-assn*)^k →$_a$ *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

definition rrand :: uint32 ⇒ uint32
  where rrand s ≡ (s * 1103515245 + 12345) AND 0x7FFFFFFF

definition rand :: uint32 ⇒ nat ⇒ (uint32 * nat) where
  rand s m ≡ let
    s = rrand s;
    r = nat-of-uint32 s;
    r = (r * m) div 0x80000000
  in (s,r)

partial-function (heap) rep where rep i N f s = (
  if i<N then do {
    s ← f s i;
    rep (i+1) N f s
  } else return s
)

declare rep.simps[code]

term hm-insert-op-impl

definition testsuite N ≡ do {
  let s=0;
  let N2=efficient-nat-div2 N;
  hm ← hm-empty-op-impl N;
```

```
  (hm,s) ← rep 0 N (λ(hm,s) i. do {
    let (s,v) = rand s N2;
    hm ← hm-insert-op-impl N id i v hm;
    return (hm,s)
  }) (hm,s);


  (hm,s) ← rep 0 N (λ(hm,s) i. do {
    let (s,v) = rand s N2;
    hm ← hm-change-key-op-impl id i v hm;
    return (hm,s)
  }) (hm,s);


  hm ← rep 0 N (λhm i. do {
    (-,hm) ← hm-pop-min-op-impl id hm;
    return hm
  }) hm;


  return ()
}

export-code rep in SML-imp

partial-function (tailrec) drep where drep i N f s = (
  if i<N then drep (i+1) N f (f s i)
  else s
)

declare drep.simps[code]


term aluprioi.insert
term aluprioi.empty
term aluprioi.pop

definition ftestsuite N ≡ do {
  let s=0;
  let N2=efficient-nat-div2 N;
  let hm= aluprioi.empty ();

  let (hm,s) = drep 0 N (λ(hm,s) i. do {
    let (s,v) = rand s N2;
    let hm = aluprioi.insert hm i v;
    (hm,s)
  }) (hm,s);

  let (hm,s) = drep 0 N (λ(hm,s) i. do {
```

```
    let (s,v) = rand s N2;
    let hm = aluprioi.insert hm i v;
    (hm,s)
  }) (hm,s);

  let hm = drep 0 N (λhm i. do {
    let (-,-,hm) = aluprioi.pop hm;
    hm
  }) hm;

  ()
}
```

export-code
testsuite ftestsuite
nat-of-integer integer-of-nat
in *SML-imp* **module-name** *Heapmap*
file ‹*heapmap-export.sml*›

end
theory *Dijkstra-Benchmark*
imports *../../../Examples/Sepref-Dijkstra*
Dijkstra-Shortest-Path.Test
begin

definition *nat-cr-graph-imp*
:: *nat* ⇒ (*nat* × *nat* × *nat*) *list* ⇒ *nat graph-impl Heap*
where *nat-cr-graph-imp* ≡ *cr-graph*

concrete-definition *nat-dijkstra-imp* **uses** *dijkstra-imp-def*[**where** *'W*=*nat*]
prepare-code-thms *nat-dijkstra-imp-def*

lemma *nat-dijkstra-imp-eq*: *nat-dijkstra-imp* = *dijkstra-imp*
⟨*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 ‹*dijkstra-export.sml*›

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*⇀*nat set* **and** *R*=⟨*nat-rel*,*R*⟩*dflt-rm-rel* **for** *R*]
ty-REL[**where** *'a*=*nat set* **and** *R*=⟨*nat-rel*⟩*list-set-rel*]

shows (*?f*::*?'c*,*succ-of-list*) ∈ *?R*
⟨*proof*⟩

concrete-definition *succ-of-list-impl* **uses** *succ-of-list-impl*

schematic-goal *acc-of-list-impl*:
notes [*autoref-tyrel*] =
ty-REL[**where** *'a*=*nat set* **and** *R*=⟨*nat-rel*⟩*dflt-rs-rel* **for** *R*]

shows (*?f*::*?'c*,*acc-of-list*) ∈ *?R*
⟨*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*⟩*dflt-rs-rel*]
assumes [*autoref-rules*]:
(*u*,*u'*)∈*nat-rel*
(*V*,*V'*)∈⟨*nat-rel*⟩*dflt-rs-rel*
(*onstack*,*onstack'*)∈⟨*nat-rel*⟩*dflt-rs-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*⟩*dflt-rs-rel*]
assumes [*autoref-rules*]:
(*succi*,*E*)∈⟨*nat-rel*⟩*slg-rel*
(*Ai*,*A*)∈⟨*nat-rel*⟩*dflt-rs-rel*
notes [*autoref-rules*] = *IdI*[*of s*]
shows (*RETURN* (*?f*::*?'c*), *blue-dfs E A s*) ∈ ⟨*?R*⟩*nres-rel*
⟨*proof*⟩

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* ≡
succ-of-list-impl o map (λ(*u*,*v*). (*nat-of-integer u*, *nat-of-integer v*))

definition *fun-acc-of-list* ≡
acc-of-list-impl o map nat-of-integer

end

interpretation *fun*: *bm-fun* ⟨*proof*⟩

locale *bm-funs* **begin**

schematic-goal *succ-of-list-impl*:
notes [*autoref-tyrel*] =
ty-REL[**where** *'a*=*nat*⇀*nat set* **and** *R*=⟨*nat-rel*,*R*⟩*iam-map-rel* **for** *R*]
ty-REL[**where** *'a*=*nat set* **and** *R*=⟨*nat-rel*⟩*list-set-rel*]

shows (*?f*::*?'c*,*succ-of-list*) ∈ *?R*
⟨*proof*⟩

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 (λ(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* ≡ *blue-dfs-impl*
definition *imp-ndfs-sz-impl* ≡ *blue-dfs-impl-sz*
definition *imp-acc-of-list l* ≡ *From-List-GA.ias-from-list* (*map nat-of-integer l*)
definition *imp-graph-of-list n l* ≡ *cr-graph* (*nat-of-integer n*) (*map* (*pairself nat-of-integer*) *l*)

export-code
nat-of-integer integer-of-nat
fun.fun-ndfs-impl fun.fun-succ-of-list fun.fun-acc-of-list
funs.funs-ndfs-impl funs.funs-succ-of-list funs.funs-acc-of-list
imp-ndfs-impl imp-ndfs-sz-impl imp-acc-of-list imp-graph-of-list
in *SML-imp* **module-name** *NDFS-Benchmark* **file** ‹*NDFS-Benchmark-export.sml*›

⟨*ML*⟩
end