The Imperative Refinement Framework

Peter Lammich

February 4, 2026

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

1 The Sepref Tool **2**
1.1 Operation Identification Phase . . . 2
1.1.1 Proper Protection of Term . . . 2
1.1.2 Operation Identification . . . 3
1.1.3 ML-Level code . . . 4
1.1.4 Default Setup . . . 9
1.2 Basic Definitions . . . 9
1.2.1 Values on Heap . . . 9
1.2.2 Constraints in Refinement Relations . . . 12
1.2.3 Heap-Nres Refinement Calculus . . . 12
1.2.4 Product Types . . . 15
1.2.5 Convenience Lemmas . . . 15
1.2.6 ML-Level Utilities . . . 22
1.3 Monadify . . . 33
1.4 Frame Inference . . . 48
1.5 Refinement Rule Management . . . 60
1.5.1 Assertion Interface Binding . . . 60
1.5.2 Function Refinement with Precondition . . . 60
1.5.3 Heap-Function Refinement . . . 62
1.5.4 Automation . . . 74
1.6 Setup for Combinators . . . 97
1.6.1 Interface Types . . . 97
1.6.2 Rewriting Inferred Interface Types . . . 100
1.6.3 ML-Code . . . 100
1.6.4 Obsolete Manual Setup Rules . . . 109
1.7 Translation . . . 109
1.7.1 Import of Parametricity Theorems . . . 124
1.7.2 Purity . . . 127
1.8 Sepref-Definition Command . . . 128
1.8.1 Setup of Extraction-Tools . . . 128
1.8.2 Synthesis setup for sepref-definition goals . . . 129
1.9 Utilities for Interface Specifications and Implementations . . . 132
1.9.1 Relation Interface Binding . . . 132

1.9.2 Operations with Precondition . . . 133
1.9.3 Constraints . . . 134
1.9.4 Composition . . . 134
1.9.5 Protected Constants . . . 137
1.9.6 Rule Collections . . . 137
1.9.7 ML-Level Declarations . . . 138
1.9.8 Obsolete Manual Specification Helpers . . . 159
1.10 Sepref Tool . . . 162
1.10.1 Sepref Method . . . 162
1.10.2 Debugging Methods . . . 166
1.10.3 Utilities . . . 168

2 Basic Setup **173**
2.1 HOL Setup . . . 173
2.1.1 Assertion Annotation . . . 173
2.1.2 Shortcuts . . . 173
2.1.3 Identity Relations . . . 174
2.1.4 Inverse Relation . . . 174
2.1.5 Single Valued and Total Relations . . . 175
2.1.6 Bounded Assertions . . . 178
2.1.7 Tool Setup . . . 183
2.1.8 HOL Combinators . . . 183
2.1.9 Basic HOL types . . . 184
2.1.10 Product . . . 185
2.1.11 Option . . . 188
2.1.12 Lists . . . 193
2.1.13 Sum-Type . . . 199
2.1.14 String Literals . . . 203
2.2 Setup for Foreach Combinator . . . 203
2.2.1 Foreach Loops . . . 203
2.2.2 For Loops . . . 219
2.3 Ad-Hoc Solutions . . . 223
2.3.1 Pure Higher-Order Functions . . . 223

3 The Imperative Isabelle Collection Framework **225**
3.1 Set Interface . . . 225
3.1.1 Operations . . . 225
3.1.2 Patterns . . . 226
3.2 Sets by Lists that Own their Elements . . . 227
3.3 Multiset Interface . . . 229
3.3.1 Additions to Multiset Theory . . . 229
3.3.2 Parametricity Setup . . . 231
3.3.3 Operations . . . 232
3.3.4 Patterns . . . 233

3.4 Priority Bag Interface . . . 234
3.4.1 Operations . . . 234
3.4.2 Patterns . . . 237
3.5 Multisets by Lists . . . 237
3.5.1 Abstract Operations . . . 237
3.5.2 Declaration of Implementations . . . 238
3.5.3 Swap two elements of a list, by index . . . 241
3.5.4 Operations . . . 242
3.5.5 Patterns . . . 243
3.6 Heap Implementation On Lists . . . 244
3.6.1 Basic Definitions . . . 245
3.6.2 Basic Operations . . . 248
3.6.3 Auxiliary operations . . . 251
3.6.4 Operations . . . 259
3.6.5 Operations as Relator-Style Refinement . . . 262
3.7 Implementation of Heaps with Arrays . . . 272
3.7.1 Setup of the Sepref-Tool . . . 272
3.7.2 Synthesis of operations . . . 274
3.7.3 Regression Test . . . 275
3.8 Map Interface . . . 276
3.8.1 Parametricity for Maps . . . 276
3.8.2 Interface Type . . . 277
3.8.3 Operations . . . 278
3.8.4 Patterns . . . 279
3.8.5 Parametricity . . . 279
3.9 Priority Maps . . . 280
3.9.1 Additional Operations . . . 281
3.10 Priority Maps implemented with List and Map . . . 283
3.10.1 Basic Setup . . . 283
3.10.2 Basic Operations . . . 284
3.10.3 Auxiliary Operations . . . 290
3.10.4 Operations . . . 294
3.11 Plain Arrays Implementing List Interface . . . 309
3.11.1 Empty . . . 317
3.11.2 Swap . . . 318
3.11.3 Length . . . 319
3.11.4 Index . . . 319
3.11.5 Butlast . . . 320
3.11.6 Append . . . 321
3.11.7 Get . . . 323
3.11.8 Contains . . . 323
3.12 Implementation of Heaps by Arrays . . . 324
3.12.1 Implement Basic Operations . . . 330
3.12.2 Auxiliary Operations . . . 332

3.12.3 Interface Operations . . . 333
3.12.4 Manual fine-tuning of code-lemmas . . . 336
3.13 Matrices . . . 340
3.13.1 Relator and Interface . . . 341
3.13.2 Operations . . . 341
3.13.3 Patterns . . . 342
3.13.4 Pointwise Operations . . . 342
3.14 Matrices by Array (Row-Major) . . . 350
3.14.1 Pointwise Operations . . . 356
3.14.2 Regression Test and Usage Example . . . 360
3.15 Sepref Bindings for Imp/HOL Collections . . . 362
3.15.1 Binding Locales . . . 367
3.15.2 Array Map (iam) . . . 376
3.15.3 Array Set (ias) . . . 377
3.15.4 Hash Map (hm) . . . 377
3.15.5 Hash Set (hs) . . . 378
3.15.6 Open Singly Linked List (osll) . . . 379
3.15.7 Circular Singly Linked List (csll) . . . 379
3.16 The Imperative Isabelle Collection Framework . . . 380

4 User Guides **382**
4.1 Quickstart Guide . . . 382
4.1.1 Introduction . . . 382
4.1.2 First Example . . . 383
4.1.3 Binary Search Example . . . 388
4.1.4 Basic Troubleshooting . . . 390
4.1.5 The Isabelle Imperative Collection Framework (IICF) 394
4.1.6 Specification of Preconditions . . . 395
4.1.7 Linearity and Copying . . . 397
4.1.8 Nesting of Data Structures . . . 399
4.1.9 Fixed-Size Data Structures . . . 400
4.1.10 Type Classes . . . 411
4.1.11 Higher-Order . . . 411
4.1.12 A-Posteriori Optimizations . . . 411
4.1.13 Short-Circuit Evaluation . . . 411
4.2 Reference Guide . . . 411
4.2.1 The Sepref Method . . . 412
4.2.2 Refinement Rules . . . 428
4.2.3 Composition . . . 430
4.2.4 Registration of Interface Types . . . 432
4.2.5 Registration of Abstract Operations . . . 432
4.2.6 High-Level tools for Interface/Implementation Declaration . . . 434
4.2.7 Defining synthesized Constants . . . 436

4.3 General Purpose Utilities 436
4.3.1 Methods 436
4.3.2 Structured Apply Scripts (experimental) 437
4.3.3 Extracting Definitions from Theorems 438

5 Examples **439**
5.1 Imperative Graph Representation 439
5.2 Simple DFS Algorithm 441
5.2.1 Definition 441
5.2.2 Refinement to Imperative/HOL 445
5.3 Imperative Implementation of Dijkstra's Shortest Paths Algorithm 446
5.4 Imperative Implementation of of Nested DFS (HPY-Improvement) 453
5.5 Generic Worklist Algorithm with Subsumption 456
5.5.1 Utilities 456
5.5.2 Search Spaces 457
5.5.3 Worklist Algorithm 458
5.5.4 Towards an Implementation 463
5.6 Non-Recursive Algebraic Datatype 466
5.6.1 Refinement Assertion 466
5.6.2 Constructors 467
5.6.3 Destructor 468
5.6.4 Regression Test 471
5.7 Snippet to Define Custom Combinators 472
5.7.1 Definition of the Combinator 472
5.7.2 Synthesis of Implementation 473
5.7.3 Setup for Sepref 474
5.7.4 Example 474
5.7.5 Limitations 475

6 Benchmarks **476**

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** ‹λ₂› *10*)
where *ABS2 f* ≡ (λ*x*. *PROTECT2* (*f x*) *DUMMY*)

lemma *beta*: (λ₂*x*. *f x*)$*x* ≡ *f x* **by** *simp*

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

definition *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* **by** *simp*
lemma *itypeI'*: *intf-type c TYPE*('*T*) **by** (*rule itypeI*)

lemma *itype-self*: (*c*::'*a*) ::ᵢ *TYPE*('*a*) **by** *simp*

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* **by** *simp*

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

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

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

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

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

lemma *id-rule*: $c{::}_i I \Longrightarrow ID\ c\ c\ I$ **by** *simp*

lemma *annot-rule*: $ID\ t\ t'\ I \Longrightarrow ID\ (t{:::}_i I)\ t'\ I$
by *simp*

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

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

1.1.3 ML-Level code

ML ‹
infix 0 THEN-ELSE-COMB'

signature ID-OP-TACTICAL = sig
val SOLVE-FWD: tactic' -> tactic'
val DF-SOLVE-FWD: bool -> tactic' -> tactic'
end

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

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

(Search for solution with DFS−strategy. If dbg−flag is given,*

```
  return sequence of stuck states if no solution is found.
*)
fun DF-SOLVE-FWD dbg tac = let
  val stuck-list-ref = Unsynchronized.ref []

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

  fun rec-tac i st = (
      (tac THEN-ALL-NEW-FWD (SOLVED' rec-tac))
      ORELSE' stuck-tac
    ) i st

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

end
```
›

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)*

ML ‹

```
structure Id-Op = struct

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

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

```
      val t1 = protect env t1
      val t2 = protect env t2
    in
      @{mk-term env: ?t1.0 $ ?t2.0}
    end
  | protect env (Abs (x,T,t)) = let
      val t = protect (T::env) t
    in
      @{mk-term env: λv-x::?'v-T. PROTECT2 ?t DUMMY}
    end
  | protect - t = t

  fun protect-conv ctxt = Refine-Util.f-tac-conv ctxt
    (protect [])
    (fn goal-ctxt =>
        simp-tac (put-simpset HOL-basic-ss goal-ctxt addsimps @{thms PRO-
TECT2-def APP-def}) 1)

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

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

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

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

  fun dest-id-rl thm = case Thm.concl-of thm of
    @{mpat (typs) Trueprop (?c::ᵢTYPE(?'v-T))} => (c,T)
  | - => raise THM(dest-id-rl,~1,[thm])


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

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

  fun id-tac ss ctxt = let
    open Id-Op-Tactical
    val certT = Thm.ctyp-of ctxt
    val cert = Thm.cterm-of ctxt

    val thy = Proof-Context.theory-of ctxt
```

```
  val id-rules = Named-Theorems-Rev.get ctxt @{named-theorems-rev id-rules}
  val pat-rules = Named-Theorems-Rev.get ctxt @{named-theorems-rev pat-rules}
    val def-pat-rules = Named-Theorems-Rev.get ctxt @{named-theorems-rev
def-pat-rules}

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

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

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

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

  val has-type = Symtab.defined ityping

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

  fun trace-fallback thm =
```

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

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

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

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

    val solve-tac = DF-SOLVE-FWD (Config.get ctxt cfg-id-debug) step-tac

  in
    case ss of
      Init => init-tac
    | Step => step-tac
    | Normal => init-tac THEN' solve-tac
    | Solve => solve-tac

  end

end
```

›

1.1.4 Default Setup

Numerals

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

lemma *id-nat-const*[*id-rules*]: $(PR\text{-}CONST\ (a{::}nat)) ::_i TYPE(nat)$ **by** *simp*
lemma *id-int-const*[*id-rules*]: $(PR\text{-}CONST\ (a{::}int)) ::_i TYPE(int)$ **by** *simp*

end

1.2 Basic Definitions

theory *Sepref-Basic*
imports
HOL−Eisbach.Eisbach
Separation-Logic-Imperative-HOL.Sep-Main
Refine-Monadic.Refine-Monadic
Lib/Sepref-Misc
Lib/Structured-Apply
Sepref-Id-Op
begin
no-notation *i-ANNOT* (**infixr** ‹:::$_i$› *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* :: $('a \Rightarrow 'c \Rightarrow assn) \Rightarrow 'a \Rightarrow 'c \Rightarrow assn$
— Tag for refinement assertion
where
$hn\text{-}ctxt\ P\ a\ c \equiv P\ a\ c$

definition *pure* :: $('b \times 'a)\ set \Rightarrow 'a \Rightarrow 'b \Rightarrow assn$
— Pure binding, not involving the heap
where $pure\ R \equiv (\lambda a\ c.\ \uparrow((c,a)\in R))$

lemma *pure-app-eq*: *pure R a c* = ↑((*c*,*a*)∈*R*) **by** (*auto simp*: *pure-def*)

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

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

definition *is-pure P* ≡ ∃ *P′*. ∀ *x x′*. *P x x′*=↑(*P′ x x′*)
lemma *is-pureI*[*intro?*]:
assumes ⋀*x x′*. *P x x′* = ↑(*P′ x x′*)
shows *is-pure P*
using *assms* **unfolding** *is-pure-def* **by** *blast*

lemma *is-pureE*:
assumes *is-pure P*
obtains *P′* **where** ⋀*x x′*. *P x x′* = ↑(*P′ x x′*)
using *assms* **unfolding** *is-pure-def* **by** *blast*

lemma *pure-pure*[*simp*]: *is-pure* (*pure P*)
unfolding *pure-def* **by** *rule blast*
lemma *pure-hn-ctxt*[*intro*!]: *is-pure P* ⟹ *is-pure* (*hn-ctxt P*)
unfolding *hn-ctxt-def*[*abs-def*] **.**

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

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

lemma *is-pure-alt-def*: *is-pure R* ⟷ (∃ *Ri*. ∀ *x y*. *R x y* = ↑((*y*,*x*)∈*Ri*))
unfolding *is-pure-def*
apply *auto*
apply (*rename-tac P′*)
apply (*rule-tac x*={(*x*,*y*). *P′ y x*} **in** *exI*)
apply *auto*
done

lemma *pure-the-pure*[*simp*]: *is-pure R* ⟹ *pure* (*the-pure R*) = *R*
unfolding *is-pure-alt-def pure-def the-pure-def*
apply (*intro ext*)
apply *clarsimp*

apply (*rename-tac a c Ri*)
apply (*rule-tac a=Ri* **in** *theI2*)
apply *auto*
done

lemma *is-pure-conv*: *is-pure R* $\longleftrightarrow$ ($\exists$ *R'*. *R* = *pure R'*)
unfolding *pure-def is-pure-alt-def* **by** *force*

lemma *is-pure-the-pure-id-eq*[*simp*]: *is-pure R* $\Longrightarrow$ *the-pure R* = *Id* $\longleftrightarrow$ *R=pure Id*
by (*auto simp*: *is-pure-conv*)

lemma *is-pure-iff-pure-assn*: *is-pure P* = ($\forall$ *x x'*. *is-pure-assn* (*P x x'*))
unfolding *is-pure-def is-pure-assn-def* **by** *metis*

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

lemma *hn-val-unfold*: *hn-val R a b* = $\uparrow$((*b*,*a*)$\in$*R*)
by (*simp add*: *hn-ctxt-def pure-def*)

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*
apply (*rule entailsI*)
unfolding *invalid-assn-def*
apply (*auto simp*: *models-in-range mod-star-trueI*)
done

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

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

lemma *invalid-pure-recover*: *invalid-assn* (*pure R*) *x y* = *pure R x y* $*$ *true*
apply (*rule ent-iffI*)
subgoal
apply (*rule entailsI*)

unfolding *invalid-assn-def*
by (*auto simp*: *pure-def*)
subgoal
unfolding *invalid-assn-def*
by (*auto simp*: *pure-def*)
done

lemma *hn-invalidI*: $h \models$*hn-ctxt P x y* $\Longrightarrow$ *hn-invalid P x y* = *true*
apply (*cases h*)
apply (*rule ent-iffI*)
apply (*auto simp*: *invalid-assn-def hn-ctxt-def*)
done

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

1.2.2 Constraints in Refinement Relations

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

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*
by (*simp add*: *hn-ctxt-def*[*abs-def*])

lemma *rdomp-pure*[*simp*]: *rdomp* (*pure R*) *a* $\longleftrightarrow$ *a*$\in$*Range R*
unfolding *rdomp-def pure-def* **by** *auto*

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

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

1.2.3 Heap-Nres Refinement Calculus

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

definition *hn-refine* Γ *c* Γ' *R m* $\equiv$ *nofail m* $\longrightarrow$

$<\Gamma>\ c\ <\lambda r.\ \Gamma' * (\exists_A x.\ R\ x\ r * \uparrow(RETURN\ x \leq m))>_t$

simproc-setup *assn-simproc-hnr* (*hn-refine* $\Gamma\ c\ \Gamma'$)
= ‹*K Seplogic-Auto.assn-simproc-fun*›

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* $\Gamma\ c\ \Gamma'\ R\ m$
using *assms* **unfolding** *hn-refine-def* **by** *blast*

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

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

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

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

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

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

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*
using *R* **unfolding** *hn-refine-def*
apply *clarify*
apply (*rule cons-pre-rulet*[*OF I*])
apply (*rule cons-post-rulet*)
apply *assumption*
apply (*sep-auto simp*: *entailst-def*)
apply (*rule enttD*)
apply (*intro entt-star-mono I' R'*)
done

lemma *hn-refine-cons-pre*:
assumes *I*: $P \Longrightarrow_t P'$
assumes *R*: *hn-refine* P' *c* *Q* *R* *m*
shows *hn-refine* *P* *c* *Q* *R* *m*
by (*rule hn-refine-cons*[*OF I R*]) *sep-auto+*

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*
using *assms*
by (*rule hn-refine-cons*[*OF entt-refl - - entt-refl*])

lemma *hn-refine-cons-res*:
$[\![$ *hn-refine* Γ *f* Γ' *R* *g*; $\bigwedge a\ c.\ R\ a\ c \Longrightarrow_t R'\ a\ c\ ]\!] \Longrightarrow$ *hn-refine* Γ *f* Γ' R' *g*
by (*erule hn-refine-cons*[*OF entt-refl*]) *sep-auto+*

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'
apply *rule*
apply (*rule cons-post-rule*)
apply (*rule hn-refineD*[*OF R*])
using *LE* **apply** (*simp add*: *pw-le-iff*)
apply (*sep-auto intro*: *order-trans*[*OF - LE*])
done

lemma *hn-refine-cons-complete*:
assumes *I*: $P \Longrightarrow_t P'$
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* $P\ c\ Q'\ R'\ m'$
apply (*rule hn-refine-ref*[*OF LE*])
apply (*rule hn-refine-cons*[*OF I R I' R'*])
done

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

1.2.4 Product Types

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

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

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

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

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

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

1.2.5 Convenience Lemmas

lemma *hn-refine-guessI*:
assumes *hn-refine P f P' R f'*
assumes *f*=*f-conc*
shows *hn-refine P f-conc P' R f'*
— To prove a refinement, first synthesize one, and then prove equality
using *assms* **by** *simp*

lemma *imp-correctI*:
assumes *R*: *hn-refine* Γ *c* Γ′ *R a*
assumes *C*: $a \leq SPEC\ \Phi$
shows $<\Gamma>\ c\ <\lambda r'.\ \exists_A r.\ \Gamma' * R\ r\ r' * \uparrow(\Phi\ r)>_t$
apply (*rule cons-post-rule*)
apply (*rule hn-refineD*[*OF R*])
apply (*rule le-RES-nofailI*[*OF C*])
apply (*sep-auto dest*: *order-trans*[*OF* - *C*])
done

lemma *hnr-pre-ex-conv*:
shows *hn-refine* $(\exists_A x.\ \Gamma\ x)\ c\ \Gamma'\ R\ a \longleftrightarrow (\forall x.$ *hn-refine* $(\Gamma\ x)\ c\ \Gamma'\ R\ a)$
unfolding *hn-refine-def*
apply *safe*
apply (*erule cons-pre-rule*[*rotated*])
apply (*rule ent-ex-postI*)
apply (*rule ent-refl*)
apply *sep-auto*
done

lemma *hnr-pre-pure-conv*:
shows *hn-refine* $(\Gamma * \uparrow P)\ c\ \Gamma'\ R\ a \longleftrightarrow (P \longrightarrow$ *hn-refine* Γ *c* Γ′ *R a*)
unfolding *hn-refine-def*
by *auto*

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

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

Return

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

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

shows *hn-refine emp* (*return c*) *emp* (*pure R*) (*RETURN a*)
— Return pure value
unfolding *hn-refine-def* **using** *assms*
by (*sep-auto simp*: *pure-def*)

Assertion

lemma *hnr-FAIL*[*simp*, *intro*!]: *hn-refine* Γ *c* Γ′ *R FAIL*
unfolding *hn-refine-def*
by *simp*

lemma *hnr-ASSERT*:
assumes Φ ⟹ *hn-refine* Γ *c* Γ′ *R c*′
shows *hn-refine* Γ *c* Γ′ *R* (*do* { *ASSERT* Φ; *c*′})
using *assms*
apply (*cases* Φ)
by *auto*

Bind

lemma *bind-det-aux*: ⟦ *RETURN x* ≤ *m*; *RETURN y* ≤ *f x* ⟧ ⟹ *RETURN y* ≤ *m* ⤜ *f*
apply (*rule order-trans*[*rotated*])
apply (*rule Refine-Basic.bind-mono*)
apply *assumption*
apply (*rule order-refl*)
apply *simp*
done

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*)
using *assms*
unfolding *hn-refine-def*
apply (*clarsimp simp add*: *pw-bind-nofail*)
apply (*rule Hoare-Triple.bind-rule*)
apply *assumption*
apply (*clarsimp intro*!: *normalize-rules simp*: *hn-ctxt-def*)
proof −
fix *x*′ *x*
assume *1*: *RETURN x* ≤ *m*
and *nofail m* ∀ *x*. *inres m x* ⟶ *nofail* (*f x*)
hence *nofail* (*f x*) **by** (*auto simp*: *pw-le-iff*)
moreover assume ⋀*x x*′. *RETURN x* ≤ *m* ⟹
nofail (*f x*) ⟶ <Γ*1* ∗ *Rh x x*′> *f*′ *x*′
<λ*r*′. ∃$_A$*r*. Γ*2 x x*′ ∗ *R r r*′ ∗ *true* ∗ ↑ (*RETURN r* ≤ *f x*)>

ultimately have $\bigwedge x'.\ <\Gamma 1 * Rh\ x\ x'> f'\ x'$
$<\lambda r'.\ \exists_A r.\ \Gamma 2\ x\ x' * R\ r\ r' * true * \uparrow (RETURN\ r \leq f\ x)>$
using *1* **by** *simp*
also have $\bigwedge r'.\ \exists_A r.\ \Gamma 2\ x\ x' * R\ r\ r' * true * \uparrow (RETURN\ r \leq f\ x) \Longrightarrow_A$
$\exists_A r.\ \Gamma' * R\ r\ r' * true * \uparrow (RETURN\ r \leq f\ x)$
apply (*sep-auto*)
apply (*rule ent-frame-fwd*[*OF IMP*[*THEN enttD*]])
apply *frame-inference*
apply (*solve-entails*)
done
finally (*cons-post-rule*) **have**
R: $<\Gamma 1 * Rh\ x\ x'> f'\ x'$
$<\lambda r'.\ \exists_A r.\ \Gamma' * R\ r\ r' * true * \uparrow (RETURN\ r \leq f\ x)>$
.
show $<\Gamma 1 * Rh\ x\ x' * true> f'\ x'$
$<\lambda r'.\ \exists_A r.\ \Gamma' * R\ r\ r' * true * \uparrow (RETURN\ r \leq m \gg= f)>$
by (*sep-auto heap*: *R intro*: *bind-det-aux*[*OF 1*])
qed

Recursion

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

lemma *hn-refine-alt*: *hn-refine Fpre c Fpost P m* $\equiv$ *nofail m* $\longrightarrow$
$<Fpre>\ c\ <\lambda r.\ hn\text{-}rel\ P\ m\ r * Fpost>_t$
apply (*rule eq-reflection*)
unfolding *hn-refine-def hn-rel-def*
apply (*simp add*: *hn-ctxt-def*)
apply (*simp only*: *star-aci*)
done

lemma *wit-swap-forall*:
assumes *W*: $<P>\ c\ <\lambda\text{-}.\ true>$
assumes *T*: $(\forall x.\ A\ x \longrightarrow <P>\ c\ <Q\ x>)$
shows $<P>\ c\ <\lambda r.\ \neg_A\ (\exists_A x.\ \uparrow (A\ x) * \neg_A\ Q\ x\ r)>$
unfolding *hoare-triple-def Let-def*
apply (*intro conjI impI allI*)
subgoal by (*elim conjE*) (*rule hoare-tripleD*[*OF W*], *assumption*+) []

subgoal
apply (*clarsimp*, *intro conjI allI*)
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*))
by *assumption*

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

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

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

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

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

  thus ?thesis
    apply (rule cons-post-rule)
    unfolding entails-def hn-rel-def
    apply clarsimp
  proof −
    fix h as p
    assume A: ∀f. f∈A ⟶ (∃ a.
      ((h, as) ⊨ Ry a p * F ∧ RETURN a ≤ f x)) ∨ ¬ nofail (f x)
    with ‹f∈A› and ‹nofail (f x)› obtain a where
      1: (h, as) ⊨ Ry a p * F and RETURN a ≤ f x
      by blast
    have
      ∀f∈A. nofail (f x) ⟶ (h, as) ⊨ Ry a p * F ∧ RETURN a ≤ f x
    proof clarsimp
      fix f'
      assume f'∈A and nofail (f' x)
      with A obtain a' where
        2: (h, as) ⊨ Ry a' p * F and RETURN a' ≤ f' x
        by blast

      moreover note preciseD'[OF PREC 1 2]
      ultimately show (h, as) ⊨ Ry a p * F ∧ RETURN a ≤ f' x by simp
    qed
    hence RETURN a ≤ (INF f∈A. f x)
      by (metis (mono-tags) le-INF-iff le-nofailI)
```

```
    with 1 show ∃a. (h, as) ⊨ Ry a p * F ∧ RETURN a ≤ (INF f∈A. f x)
      by blast
  qed
qed

lemma hn-admissible':
  assumes PREC: precise Ry
  assumes E: ∀f∈A. nofail (f x) ⟶ <P> c <λr. hn-rel Ry (f x) r * F>t
  assumes NF: nofail (INF f∈A. f x)
  shows <P> c <λr. hn-rel Ry (INF f∈A. f x) r * F>t
  apply (rule hn-admissible[OF PREC, where F=F*true, simplified])
  apply simp
  by fact+

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 (λf. 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)
  unfolding RECT-gfp-def
proof (simp, intro conjI impI)
  assume trimono aB
  hence mono aB by (simp add: trimonoD)
  have ∀ax px.
    hn-refine (hn-ctxt Rx ax px * F) (heap.fixp-fun cB px) (F' ax px) Ry
      (gfp aB ax)
    apply (rule gfp-cadm-induct[OF - - ‹mono aB›])

    apply rule
    apply (auto simp: hn-refine-alt intro: hn-admissible'[OF PREC]) []

    apply (auto simp: hn-refine-alt) []

    apply clarsimp
    apply (subst heap.mono-body-fixp[of cB, OF M])
    apply (rule S)
    apply blast
    done
  thus hn-refine (hn-ctxt Rx ax px * F)
     (ccpo.fixp (fun-lub Heap-lub) (fun-ord Heap-ord) cB px) (F' ax px) Ry
     (gfp aB ax) by simp
qed

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 (λf. cB f x))
  shows hn-refine
    (hn-ctxt Rx ax px ∗ F) (heap.fixp-fun cB px) (F' ax px) Ry (RECT aB ax)
  unfolding RECT-def
proof (simp, intro conjI impI)
  assume trimono aB
  hence flatf-mono-ge aB by (simp add: trimonoD)
  have ∀ ax px.
    hn-refine (hn-ctxt Rx ax px ∗ F) (heap.fixp-fun cB px) (F' ax px) Ry
      (flatf-gfp aB ax)

    apply (rule flatf-ord.fixp-induct[OF - ‹flatf-mono-ge aB›])

    apply (rule flatf-admissible-pointwise)
    apply simp

    apply (auto simp: hn-refine-alt) []

    apply clarsimp
    apply (subst heap.mono-body-fixp[of cB, OF M])
    apply (rule S)
    apply blast
    done
  thus hn-refine (hn-ctxt Rx ax px ∗ F)
     (ccpo.fixp (fun-lub Heap-lub) (fun-ord Heap-ord) cB px) (F' ax px) Ry
     (flatf-gfp aB ax) by simp
qed

lemma hnr-If:
  assumes P: Γ ⟹_t Γ1 ∗ hn-val bool-rel a a'
  assumes RT: a ⟹ hn-refine (Γ1 ∗ hn-val bool-rel a a') b' Γ2b R b
  assumes RE: ¬a ⟹ hn-refine (Γ1 ∗ hn-val bool-rel a a') c' Γ2c R c
  assumes IMP: Γ2b ∨_A Γ2c ⟹_t Γ'
  shows hn-refine Γ (if a' then b' else c') Γ' R (if a then b else c)
  apply (rule hn-refine-cons[OF P])
  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*)
done

1.2.6 ML-Level Utilities

ML ‹
signature SEPREF-BASIC = sig
(∗ *Destroy lambda term, return function to reconstruct. Bound var is replaced by free.* ∗)
val dest-lambda-rc: *Proof.context −> term −> ((term ∗ (term −> term)) ∗ Proof.context)*
(∗ *Apply function under lambda. Bound var is replaced by free.* ∗)
val apply-under-lambda: *(Proof.context −> term −> term) −> Proof.context −> term −> term*

(∗ *'a nres type* ∗)
val is-nresT: *typ −> bool*
val mk-nresT: *typ −> typ*
val dest-nresT: *typ −> typ*

(∗ *Make certified ==* ∗)
val mk-cequals: *cterm ∗ cterm −> cterm*
(∗ *Make* $\Longrightarrow_A$ ∗)
val mk-entails: *term ∗ term −> term*

(∗ *Operations on pre−terms* ∗)
val constrain-type-pre: *typ −> term −> term* (∗ *t::T* ∗)

val mk-pair-in-pre: *term −> term −> term −> term* (∗ *(c,a) ∈ R* ∗)

val mk-compN-pre: *int −> term −> term −> term* (∗ *f o...o g*∗)

val mk-curry0-pre: *term −> term* (∗ *curry0 f* ∗)
val mk-curry-pre: *term −> term* (∗ *curry f* ∗)
val mk-curryN-pre: *int −> term −> term* (∗ *curry (...(curry f)...)* ∗)

val mk-uncurry0-pre: *term −> term* (∗ *uncurry0 f* ∗)
val mk-uncurry-pre: *term −> term* (∗ *uncurry f* ∗)
val mk-uncurryN-pre: *int −> term −> term* (∗ *uncurry (...(uncurry f)...)* ∗)

(∗ *Conversion for hn-refine − term*∗)
val hn-refine-conv: *conv −> conv −> conv −> conv −> conv −> conv*

```
(* Conversion on abstract value (last argument) of hn-refine - term *)
val hn-refine-conv-a: conv -> conv

(* Conversion on abstract value of hn-refine term in conclusion of theorem *)
val hn-refine-concl-conv-a: (Proof.context -> conv) -> Proof.context -> conv

(* Destruct hn-refine term *)
val dest-hn-refine: term -> term * term * term * term * term
(* Make hn-refine term *)
val mk-hn-refine: term * term * term * term * term -> term
(* Check if given term is Trueprop (hn-refine ...). Use with CONCL-COND'. *)
val is-hn-refine-concl: term -> bool

(* Destruct abs-fun, returns RETURN-flag, (f, args) *)
val dest-hnr-absfun: term -> bool * (term * term list)
(* Make abs-fun. *)
val mk-hnr-absfun: bool * (term * term list) -> term
(* Make abs-fun. Guess RETURN-flag from type. *)
val mk-hnr-absfun': (term * term list) -> term

(* Prove permutation of *. To be used with f-tac-conv. *)
val star-permute-tac: Proof.context -> tactic

(* Make separation conjunction *)
val mk-star: term * term -> term
(* Make separation conjunction from list. [] yields emp. *)
val list-star: term list -> term
(* Decompose separation conjunction. emp yields []. *)
val strip-star: term -> term list

(* Check if true-assertion *)
val is-true: term -> bool

(* Check if term is hn-ctxt-assertion *)
val is-hn-ctxt: term -> bool
(* Decompose hn-ctxt-assertion *)
val dest-hn-ctxt: term -> term * term * term
(* Decompose hn-ctxt-assertion, NONE if term has wrong format *)
val dest-hn-ctxt-opt: term -> (term * term * term) option


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

```
  (* No tracing or intermediate result, all phases *)
  val dflt-phases-ctrl: phases-ctrl
  (* Tracing, intermediate result, all phases *)
  val dbg-phases-ctrl: phases-ctrl
  val flag-phases-ctrl: bool -> phases-ctrl

  (* Name, tactic, expected number of created goals (may be negative for solved
goals) *)
  type phase = string * (Proof.context -> tactic') * int

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

end

structure Sepref-Basic: SEPREF-BASIC = struct

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


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

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


  (* Functions on pre-terms *)
  fun mk-pair-in-pre x y r = Const (@{const-name Set.member}, dummyT) $
    (Const (@{const-name Product-Type.Pair}, dummyT) $ x $ y) $ r
```

```
  fun mk-uncurry-pre t = Const(@{const-name uncurry}, dummyT)$t
  fun mk-uncurry0-pre t = Const(@{const-name uncurry0}, dummyT)$t
  fun mk-uncurryN-pre 0 = mk-uncurry0-pre
    | mk-uncurryN-pre 1 = I
    | mk-uncurryN-pre n = mk-uncurry-pre o mk-uncurryN-pre (n-1)

  fun mk-curry-pre t = Const(@{const-name curry}, dummyT)$t
  fun mk-curry0-pre t = Const(@{const-name curry0}, dummyT)$t
  fun mk-curryN-pre 0 = mk-curry0-pre
    | mk-curryN-pre 1 = I
    | mk-curryN-pre n = mk-curry-pre o mk-curryN-pre (n-1)


  fun mk-compN-pre 0 f g = f $ g
    | mk-compN-pre n f g = let
       val g = fold (fn i => fn t => t$Bound i) (n-2 downto 0) g
       val t = Const(@{const-name Fun.comp},dummyT) $ f $ g

        val t = fold (fn i => fn t => Abs (x^string-of-int i,dummyT,t)) (n-1
downto 1) t
      in
        t
      end

  fun constrain-type-pre T t = Const(@{syntax-const -type-constraint-},T-->T)
$ t




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

    val hn-refine-conv-a = hn-refine-conv all-conv all-conv all-conv all-conv

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

  end

  (* FIXME: Strange dependency! *)
  val mk-cequals = uncurry SMT-Util.mk-cequals
```

```
  val mk-entails = HOLogic.mk-binrel @{const-name entails}

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

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

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

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

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

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

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

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

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

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

  end

  fun dest-hn-refine @{mpat hn-refine ?P ?c ?Q ?R ?a} = (P,c,Q,R,a)
    | dest-hn-refine t = raise TERM(dest-hn-refine,[t])

  fun mk-hn-refine (P,c,Q,R,a) = @{mk-term hn-refine ?P ?c ?Q ?R ?a}

  val is-hn-refine-concl = can (HOLogic.dest-Trueprop #> dest-hn-refine)
```

```
fun star-permute-tac ctxt = ALLGOALS (simp-tac (put-simpset HOL-basic-ss
ctxt addsimps @{thms star-aci}))


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

val dflt-phases-ctrl = {trace=false,int-res=false,start=NONE,stop=NONE}
val dbg-phases-ctrl = {trace=true,int-res=true,start=NONE,stop=NONE}
fun flag-phases-ctrl dbg = if dbg then dbg-phases-ctrl else dflt-phases-ctrl

type phase = string * (Proof.context -> tactic') * int

local
  fun ph-range phases start stop = let
    fun find-phase name = let
      val i = find-index (fn (n,-,-) => n=name) phases
      val - = if i<0 then error (No such phase: ^ name) else ()
    in
      i
    end

    val i = case start of NONE => 0 | SOME n => find-phase n
    val j = case stop of NONE => length phases - 1 | SOME n => find-phase
n

    val phases = take (j+1) phases |> drop i

    val - = case phases of [] => error No phases selected, range is empty | - =>
()
  in
    phases
  end
in

  fun PHASES' phases ctrl ctxt = let
    val phases = ph-range phases (#start ctrl) (#stop ctrl)
    val phases = map (fn (n,tac,d) => (n,tac ctxt,d)) phases

    fun r [] - st = Seq.single st
      | r ((name,tac,d)::tacs) i st = let
          val n = Thm.nprems-of st
          val bailout-tac = if #int-res ctrl then all-tac else no-tac
        fun trace-tac msg st = (if #trace ctrl then tracing msg else (); Seq.single
```

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

    in
      r phases
    end


  end

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



(*

  fun xPHASES' dbg tacs ctxt = let
    val tacs = map (fn (tac,d) => (tac ctxt,d)) tacs

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

```
            (r tacs i st)
          else
            bailout-tac st
      ,
        K bailout-tac)
    end i st

  in
    r tacs
  end
*)
end


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

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

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

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

  (* Tracing message and current subgoal *)
  val tracing-tac': string -> Proof.context -> tactic'
  (* Warning message and current subgoal *)
  val warning-tac': string -> Proof.context -> tactic'
  (* Error message and current subgoal *)
  val error-tac': string -> Proof.context -> tactic'

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

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

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

```
  val msg-allgoals: string -> Proof.context -> int -> thm -> string

end

structure Sepref-Debugging: SEPREF-DEBUGGING = struct

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

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

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

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

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

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

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


  local
    fun gen-subgoal-msg-tac do-msg msg ctxt = IF-EXGOAL (fn i => fn st =>
let
```

```
      val t = nth (Thm.prems-of st) (i-1)
      val - = Pretty.block [Pretty.str msg, Pretty.fbrk, Syntax.pretty-term ctxt t]
        |> Pretty.string-of |> do-msg

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


  fun dbg-msg-tac cfg msg ctxt =
    if is-debug cfg ctxt then (fn i => fn st => (tracing (msg ctxt i st); Seq.single
st))
    else K all-tac
  fun dbg-msg-tac' msg ctxt =
     if is-debug' ctxt then (fn i => fn st => (tracing (msg ctxt i st); Seq.single
st))
    else K all-tac

  fun msg-text msg - - - = msg

  fun msg-from-subgoal msg sgmsg ctxt i st =
    case try (nth (Thm.prems-of st)) (i-1) of
      NONE => msg ^ \n ^ Subgoal out of range
    | SOME t => msg ^ \n ^ sgmsg t ctxt

  fun msg-subgoal msg = msg-from-subgoal msg (fn t => fn ctxt =>
    Syntax.pretty-term ctxt t |> Pretty.string-of
  )

  fun msg-allgoals msg ctxt - st =
    Goal-Display.print-goal ctxt msg st

end
›


ML ‹
  (* Tactics for produced subgoals *)
  infix 1 THEN-NEXT THEN-ALL-NEW-LIST THEN-ALL-NEW-LIST'
  signature STACTICAL = sig
    (* Apply first tactic on this subgoal, and then second tactic on next subgoal *)
    val THEN-NEXT: tactic' * tactic' -> tactic'
    (* Apply tactics to the current and following subgoals *)
    val APPLY-LIST: tactic' list -> tactic'
```

```
    (* Apply list of tactics on subgoals emerging from tactic.
      Requires exactly one tactic per emerging subgoal.*)
    val THEN-ALL-NEW-LIST: tactic' * tactic' list -> tactic'
   (* Apply list of tactics to subgoals emerging from tactic, use fallback for additional
subgoals. *)
    val THEN-ALL-NEW-LIST': tactic' * (tactic' list * tactic') -> tactic'

  end

  structure STactical : STACTICAL = struct
    infix 1 THEN-WITH-GOALDIFF
    fun (tac1 THEN-WITH-GOALDIFF tac2) st = let
      val n1 = Thm.nprems-of st
    in
      st |> (tac1 THEN (fn st => tac2 (Thm.nprems-of st - n1) st ))
    end

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

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

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

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


  end
```

open STactical
›

end

1.3 Monadify

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

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

The monadify process is separated into 2 steps.

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

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

definition *SP* — Tag to protect content from further application of arity and combinator equations
  where [*simp*]: $SP\ x \equiv x$
lemma *SP-cong*[*cong*]: $SP\ x \equiv SP\ x$ **by** *simp*
lemma *PR-CONST-cong*[*cong*]: $PR\text{-}CONST\ x \equiv PR\text{-}CONST\ x$ **by** *simp*

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

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

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

lemma *monadify-simps*:
  Refine-Basic.bind$(*RETURN*$x)$(λ$_2$x. f x) = f x$

```
  EVAL$x ≡ RETURN$x
  by simp-all

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

lemma remove-pass-simps:
  Refine-Basic.bind$(PASS$x)$(λ2x. f x) ≡ f x
  Refine-Basic.bind$m$(λ2x. PASS$x) ≡ m
  by simp-all


definition COPY :: 'a ⇒ 'a
  — Marks required copying of parameter
  where [simp]: COPY x ≡ x
lemma RET-COPY-PASS-eq: RETURN$(COPY$p) = PASS$p by simp


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

ML ‹
  structure Sepref-Monadify = struct
    local
      fun cr-var (i,T) = (v^string-of-int i, Free (--v^string-of-int i,T))

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


      fun
        bind-args exp0 [] = exp0
      | bind-args exp0 ((x,m)::xms) = let
          val lr = bind-args exp0 xms
            |> incr-boundvars 1
            |> lambda2-name x
        in @{mk-term Refine-Basic.bind$?m$?lr} end

      fun monadify t = let
        val (f,args) = Autoref-Tagging.strip-app t
        val - = not (is-Abs f) orelse
          raise TERM (monadify: higher−order,[t])

        val argTs = map fastype-of args
```

```
    (*val args = map monadify args*)
    val args = map (fn a => @{mk-term EVAL$?a}) args

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

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

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

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

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

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

  val monadify-simproc =
  simproc-setup ‹passive monadify (EVAL$a) = ‹K (try o monadify-conv-aux)››;

end

local
```

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

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

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

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

  end

  local

    open Sepref-Basic

   fun dp ctxt (@{mpat Refine-Basic.bind$(PASS$?p)$(?t' AS_p (λ-. PROTECT2
- DUMMY))}) =
        let
          val (t',ps) = let
              val ((t',rc),ctxt) = dest-lambda-rc ctxt t'
             val f = case t' of @{mpat PROTECT2 ?f -} => f | - => raise Match

              val (f,ps) = dp ctxt f
              val t' = @{mk-term PROTECT2 ?f DUMMY}
              val t' = rc t'
            in
              (t',ps)
            end

          val dup = member (aconv) ps p
          val t = if dup then
            @{mk-term Refine-Basic.bind$(RETURN$(COPY$?p))$?t'}
          else
            @{mk-term Refine-Basic.bind$(PASS$?p)$?t'}
```

```
        in
          (t,p::ps)
        end
      | dp ctxt (t1$t2) = (#1 (dp ctxt t1) $ #1 (dp ctxt t2),[])
      | dp ctxt (t as (Abs -)) = (apply-under-lambda (#1 oo dp) ctxt t,[])
      | dp - t = (t,[])

    fun dp-conv ctxt = Refine-Util.f-tac-conv ctxt
      (#1 o dp ctxt)
      (fn goal-ctxt =>
       ALLGOALS (simp-tac (put-simpset HOL-basic-ss goal-ctxt addsimps @{thms
RET-COPY-PASS-eq})))


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


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

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

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

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

  (*fun ops-tac ctxt = CONVERSION (
```

```
    Sepref-Basic.hn-refine-concl-conv-a monadify-conv ctxt)∗)

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

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

  fun remove-pass-tac ctxt =
   simp-tac (put-simpset HOL-basic-ss ctxt addsimps @{thms remove-pass-simps})

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

end
›

lemma dflt-arity[sepref-monadify-arity]:
  RETURN ≡ λ2x. SP RETURN$x
  RECT ≡ λ2B x. SP RECT$(λ2D x. B$(λ2x. RCALL$D$x)$x)$x
  case-list ≡ λ2fn fc l. SP case-list$fn$(λ2x xs. fc$x$xs)$l
  case-prod ≡ λ2fp p. SP case-prod$(λ2a b. fp$a$b)$p
  case-option ≡ λ2fn fs ov. SP case-option$fn$(λ2x. fs$x)$ov
  If ≡ λ2b t e. SP If$b$t$e
  Let ≡ λ2x f. SP Let$x$(λ2x. f$x)
  by (simp-all only: SP-def APP-def PROTECT2-def RCALL-def)


lemma dflt-comb[sepref-monadify-comb]:
  ⋀B x. RECT$B$x ≡ Refine-Basic.bind$(EVAL$x)$(λ2x. SP (RECT$B$x))
  ⋀D x. RCALL$D$x ≡ Refine-Basic.bind$(EVAL$x)$(λ2x. SP (RCALL$D$x))
  ⋀fn fc l. case-list$fn$fc$l ≡ Refine-Basic.bind$(EVAL$l)$(λ2l. (SP case-list$fn$fc$l))
  ⋀fp p. case-prod$fp$p ≡ Refine-Basic.bind$(EVAL$p)$(λ2p. (SP case-prod$fp$p))
  ⋀fn fs ov. case-option$fn$fs$ov
    ≡ Refine-Basic.bind$(EVAL$ov)$(λ2ov. (SP case-option$fn$fs$ov))
  ⋀b t e. If$b$t$e ≡ Refine-Basic.bind$(EVAL$b)$(λ2b. (SP If$b$t$e))
  ⋀x. RETURN$x ≡ Refine-Basic.bind$(EVAL$x)$(λ2x. SP (RETURN$x))
```

$\bigwedge x\ f.\ Let\$x\$f \equiv Refine\text{-}Basic.bind\$(EVAL\$x)\$(\lambda_2 x.\ (SP\ Let\$x\$f))$
by (*simp-all*)

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

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

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 $PROP\ (CONSTRAINT\text{-}SLOT\ (PROP\ P)) \Longrightarrow PROP\ Q$
using *assms* **unfolding** *CONSTRAINT-SLOT-def* **by** *simp*

lemma *insert-slot-rl2*:
assumes $PROP\ P \Longrightarrow PROP\ (CONSTRAINT\text{-}SLOT\ S) \Longrightarrow PROP\ Q$
shows $PROP\ (CONSTRAINT\text{-}SLOT\ (PROP\ S\ \&\&\&\ PROP\ P)) \Longrightarrow PROP\ Q$
using *assms* **unfolding** *CONSTRAINT-SLOT-def conjunction-def* **.**

lemma *remove-slot*: $PROP\ (CONSTRAINT\text{-}SLOT\ (Trueprop\ True))$
unfolding *CONSTRAINT-SLOT-def* **by** (*rule TrueI*)

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

lemma *CONSTRAINT-D*:
assumes $CONSTRAINT\ (P::'a \Rightarrow bool)\ x$

shows *P x*
using *assms* **unfolding** *CONSTRAINT-def* **by** *simp*

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

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

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

named-theorems *constraint-simps* ‹*Simplification of constraints*›

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

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

(* *Focus on goals in slot* *)
val focus: *tactic*
(* *Unfocus goals in slot* *)

```
val unfocus: tactic
(* Unfocus goals, and insert them as first subgoals *)
val unfocus-ins:tactic

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


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

(******** Constraint solving *)

val add-constraint-rule: thm -> Context.generic -> Context.generic
val del-constraint-rule: thm -> Context.generic -> Context.generic
val get-constraint-rules: Proof.context -> thm list

val add-safe-constraint-rule: thm -> Context.generic -> Context.generic
val del-safe-constraint-rule: thm -> Context.generic -> Context.generic
val get-safe-constraint-rules: Proof.context -> thm list

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

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

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

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

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

```
  val setup: theory -> theory

end


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

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

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

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

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

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

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


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

  val dest-slot-tac = SELECT-GOAL (
    ALLGOALS (
      CONVERSION (Conv.rewr-conv @{thm CONSTRAINT-SLOT-def})
      THEN' Goal.conjunction-tac
```

```
    THEN' TRY o resolve0-tac @{thms TrueI})
  THEN distinct-subgoals-tac
)

val remove-slot-tac = WITH-SLOT (resolve0-tac @{thms remove-slot})

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

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

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

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

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


fun ON-SLOT tac = focus THEN tac THEN unfocus

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

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

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

```
end*)

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

fun check-constraint-rl thm = let
  fun ck (t as @{mpat Trueprop (?C -)}) =
        if is-Var (Term.head-of C) then
          raise TERM (Schematic head in constraint rule,[t,Thm.prop-of thm])
        else ()
    | ck @{mpat ⋀-. PROP ?t} = ck t
    | ck @{mpat PROP ?s ⟹ PROP ?t} = (ck s; ck t)
    | ck t = raise TERM (Invalid part of constraint rule,[t,Thm.prop-of thm])

in
  ck (Thm.prop-of thm); thm
end

fun check-unsafe-constraint-rl thm = let
  val - = Thm.nprems-of thm = 0
    andalso raise TERM(Unconditional constraint rule must be safe (register
this as safe rule),[Thm.prop-of thm])
in
  thm
end

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

  structure safe-constraint-rules = Named-Sorted-Thms (
    val name = @{binding safe-constraint-rules}
    val description = Safe Constraint rules
    val sort = K I
    fun transform context = let
      open Conv
```

```
      val ctxt = Context.proof-of context
    in
      unfold-abbrevs ctxt #> map check-constraint-rl
    end
  )

end

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

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

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

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

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

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

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

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

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

fun SAFE tac = SEQ-CASES tac (all-tac, no-tac)
```

```
fun SAFE' tac = SAFE o tac

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

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

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

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

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

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

 fun solve-tac cn-nets ctxt = SOLVED' (wrap-tac (solve-step-tac cn-nets) ctxt)
  fun safe-tac cn-nets ctxt =
    simp-constraints-tac ctxt
    THEN-ALL-NEW unfold-abbrevs-tac ctxt
        THEN-ALL-NEW (solve-tac cn-nets ctxt ORELSE' TRY o wrap-tac
```

```
(safe-step-tac cn-nets) ctxt)

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

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

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

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


    end

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

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

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


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

  end
```

›

setup *Sepref-Constraints.setup*

method-setup *print-slot* = ‹*Scan.succeed* (*fn ctxt => SIMPLE-METHOD* (*Sepref-Constraints.print-slot-tac ctxt*))›

method-setup *solve-constraint* = ‹*Scan.succeed* (*fn ctxt => SIMPLE-METHOD'* (*Sepref-Constraints.solve-constraint'-tac ctxt*))›
method-setup *safe-constraint* = ‹*Scan.succeed* (*fn ctxt => SIMPLE-METHOD'* (*Sepref-Constraints.safe-constraint'-tac ctxt*))›

end

1.4 Frame Inference

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

In this theory, we provide a specific frame inference tactic for Sepref.

The first tactic, *frame-tac*, is a standard frame inference tactic, based on the assumption that only *hn-ctxt*-assertions need to be matched.

The second tactic, *merge-tac*, resolves entailments of the form $F1 \vee_A F2 \Longrightarrow_t ?F$ that occur during translation of if and case statements. It synthesizes a new frame ?F, where refinements of variables with equal refinements in *F1* and *F2* are preserved, and the others are set to *hn-invalid*.

definition *mismatch-assn* :: $('a \Rightarrow 'c \Rightarrow assn) \Rightarrow ('a \Rightarrow 'c \Rightarrow assn) \Rightarrow 'a \Rightarrow 'c \Rightarrow assn$
  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* $\Longrightarrow_t$ *hn-ctxt R x y*
  by (*auto simp*: *is-pure-conv invalid-pure-recover hn-ctxt-def*)

lemma *frame-thms*:
  $P \Longrightarrow_t P$
  $P \Longrightarrow_t P' \Longrightarrow F \Longrightarrow_t F' \Longrightarrow F*P \Longrightarrow_t F'*P'$
  hn-ctxt R x y $\Longrightarrow_t$ *hn-invalid R x y*
  hn-ctxt R x y $\Longrightarrow_t$ *hn-ctxt* (λ- -. *true*) *x y*
  CONSTRAINT is-pure R ⟹ *hn-invalid R x y* $\Longrightarrow_t$ *hn-ctxt R x y*
  apply −

applyS *simp*
applyS (*rule entt-star-mono*; *assumption*)
subgoal
apply (*simp add*: *hn-ctxt-def*)
apply (*rule enttI*)
apply (*rule ent-trans*[*OF invalidate*[*of R*]])
by *solve-entails*
applyS (*sep-auto simp*: *hn-ctxt-def*)
applyS (*erule recover-pure-aux*)
done

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

Rules to discharge unmatched stuff

lemma *frame-rem1*: $P \Longrightarrow_t P$ **by** *simp*

lemma *frame-rem2*: $F \Longrightarrow_t F' \Longrightarrow F * \textit{hn-ctxt}\ A\ x\ y \Longrightarrow_t F' * \textit{hn-ctxt}\ A\ x\ y$
apply (*rule entt-star-mono*) **by** *auto*

lemma *frame-rem3*: $F \Longrightarrow_t F' \Longrightarrow F * \textit{hn-ctxt}\ A\ x\ y \Longrightarrow_t F'$
using *frame-thms*(*2*) **by** *fastforce*

lemma *frame-rem4*: $P \Longrightarrow_t \textit{emp}$ **by** *simp*

lemmas *frame-rem-thms* = *frame-rem1 frame-rem2 frame-rem3 frame-rem4*

named-theorems-rev *sepref-frame-rem-rules*
‹*Sepref*: *Additional rules to resolve remainder of frame−pairing*›

lemma *ent-disj-star-mono*:
$[\![A \vee_A C \Longrightarrow_A E;\ B \vee_A D \Longrightarrow_A F]\!] \Longrightarrow A*B \vee_A C*D \Longrightarrow_A E*F$
by (*metis ent-disjI1 ent-disjI2 ent-disjE ent-star-mono*)

lemma *entt-disj-star-mono*:
$[\![A \vee_A C \Longrightarrow_t E;\ B \vee_A D \Longrightarrow_t F]\!] \Longrightarrow A*B \vee_A C*D \Longrightarrow_t E*F$
proof −
assume *a1*: $A \vee_A C \Longrightarrow_t E$
assume $B \vee_A D \Longrightarrow_t F$
then have $A * B \vee_A C * D \Longrightarrow_A \textit{true} * E * (\textit{true} * F)$
using *a1* **by** (*simp add*: *ent-disj-star-mono enttD*)
then show *?thesis*
by (*metis* (*no-types*) *assn-times-comm enttI merge-true-star-ctx star-aci*(*3*))
qed

lemma *hn-merge1*:

$F \vee_A F \Longrightarrow_t F$

⟦ *hn-ctxt R1 x x′* ∨$_A$ *hn-ctxt R2 x x′* ⟹$_t$ *hn-ctxt R x x′*; *Fl* ∨$_A$ *Fr* ⟹$_t$ *F* ⟧
⟹ *Fl* ∗ *hn-ctxt R1 x x′* ∨$_A$ *Fr* ∗ *hn-ctxt R2 x x′* ⟹$_t$ *F* ∗ *hn-ctxt R x x′*
apply *simp*
by (*rule entt-disj-star-mono*; *simp*)

lemma *hn-merge2*:
hn-invalid R x x′ ∨$_A$ *hn-ctxt R x x′* ⟹$_t$ *hn-invalid R x x′*
hn-ctxt R x x′ ∨$_A$ *hn-invalid R x x′* ⟹$_t$ *hn-invalid R x x′*
by (*sep-auto eintros*: *invalidate ent-disjE intro*!: *ent-imp-entt simp*: *hn-ctxt-def*)+

lemma *invalid-assn-mono*: *hn-ctxt A x y* ⟹$_t$ *hn-ctxt B x y*
⟹ *hn-invalid A x y* ⟹$_t$ *hn-invalid B x y*
by (*clarsimp simp*: *invalid-assn-def entailst-def entails-def hn-ctxt-def*)
(*force simp*: *mod-star-conv*)

lemma *hn-merge3*:
⟦*NO-MATCH* (*hn-invalid XX*) *R2*; *hn-ctxt R1 x x′* ∨$_A$ *hn-ctxt R2 x x′* ⟹$_t$ *hn-ctxt Rm x x′*⟧ ⟹ *hn-invalid R1 x x′* ∨$_A$ *hn-ctxt R2 x x′* ⟹$_t$ *hn-invalid Rm x x′*
⟦*NO-MATCH* (*hn-invalid XX*) *R1*; *hn-ctxt R1 x x′* ∨$_A$ *hn-ctxt R2 x x′* ⟹$_t$ *hn-ctxt Rm x x′*⟧ ⟹ *hn-ctxt R1 x x′* ∨$_A$ *hn-invalid R2 x x′* ⟹$_t$ *hn-invalid Rm x x′*
apply (*meson entt-disjD1 entt-disjD2 entt-disjE entt-trans frame-thms*(*3*) *invalid-assn-mono*)
apply (*meson entt-disjD1 entt-disjD2 entt-disjE entt-trans frame-thms*(*3*) *invalid-assn-mono*)
done

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

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

lemma *hn-merge-mismatch*: *hn-ctxt R1 x x′* ∨$_A$ *hn-ctxt R2 x x′* ⟹$_t$ *hn-mismatch R1 R2 x x′*
by (*sep-auto simp*: *hn-ctxt-def mismatch-assn-def*)

lemma *is-merge*: *P1*∨$_A$*P2*⟹$_t$*P* ⟹ *P1*∨$_A$*P2*⟹$_t$*P* .

lemma *merge-mono*: ⟦*A*⟹$_t$*A′*; *B*⟹$_t$*B′*; *A′*∨$_A$*B′* ⟹$_t$ *C*⟧ ⟹ *A*∨$_A$*B* ⟹$_t$ *C*
by (*meson entt-disjE entt-disjI1-direct entt-disjI2-direct entt-trans*)

Apply forward rule on left or right side of merge

lemma *gen-merge-cons1*: ⟦*A*⟹$_t$*A′*; *A′*∨$_A$*B* ⟹$_t$ *C*⟧ ⟹ *A*∨$_A$*B* ⟹$_t$ *C*
by (*meson merge-mono entt-refl*)

lemma *gen-merge-cons2*: ⟦*B*⟹$_t$*B′*; *A*∨$_A$*B′* ⟹$_t$ *C*⟧ ⟹ *A*∨$_A$*B* ⟹$_t$ *C*
by (*meson merge-mono entt-refl*)

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

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

definition *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*)
unfolding *RECOVER-PURE-def*
subgoal by *sep-auto*
subgoal by (*drule* (*1*) *entt-star-mono*)
subgoal by (*rule recover-pure-aux*)
subgoal by *sep-auto*
done

lemma *recover-pure-triv*:
RECOVER-PURE P P
unfolding *RECOVER-PURE-def* **by** *sep-auto*

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*
apply (*rule hn-refine-preI*)
apply (*rule hn-refine-cons-post*)
apply (*rule assms*)
using *assms*(*1*) **unfolding** *WEAKEN-HNR-POST-def* **by** *blast*

lemma *weaken-hnr-post-triv*: *WEAKEN-HNR-POST* Γ *P P*
unfolding *WEAKEN-HNR-POST-def*
by *sep-auto*

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 (*goal-cases*)
case *1* **thus** *?case*
unfolding *WEAKEN-HNR-POST-def*

apply *clarsimp*
apply (*rule entt-star-mono*)
by (*auto simp*: *mod-star-conv*)
next
case *2* **thus** *?case* **by** (*rule weaken-hnr-post-triv*)
next
case *3* **thus** *?case*
unfolding *WEAKEN-HNR-POST-def*
by (*sep-auto simp*: *invalid-assn-def hn-ctxt-def*)
qed

lemma *reorder-enttI*:
assumes $A{*}true = C{*}true$
assumes $B{*}true = D{*}true$
shows $(A \Longrightarrow_t B) \equiv (C \Longrightarrow_t D)$
apply (*intro eq-reflection*)
unfolding *entt-def-true*
by (*simp add*: *assms*)

lemma *merge-sat1*: $(A \vee_A A' \Longrightarrow_t Am) \Longrightarrow (A \vee_A Am \Longrightarrow_t Am)$
using *entt-disjD1 entt-disjE* **by** *blast*
lemma *merge-sat2*: $(A \vee_A A' \Longrightarrow_t Am) \Longrightarrow (Am \vee_A A' \Longrightarrow_t Am)$
using *entt-disjD2 entt-disjE* **by** *blast*

ML ‹
signature SEPREF-FRAME = sig

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

val frame-step-tac: (*Proof.context* −> *tactic′*) −> *bool* −> *Proof.context* −> *tactic′*

(∗ *Reorder frame* ∗)

```
  val prepare-frame-tac : Proof.context -> tactic'
  (* Solve a RECOVER-PURE goal, inserting constraints as necessary *)
  val recover-pure-tac: Proof.context -> tactic'

  (* Split precondition of hnr-goal into frame and arguments *)
  val align-goal-tac: Proof.context -> tactic'
  (* Normalize goal's precondition *)
  val norm-goal-pre-tac: Proof.context -> tactic'
  (* Rearrange precondition of hnr-term according to parameter order, normalize
all relations *)
  val align-rl-conv: Proof.context -> conv

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

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

  val cfg-debug: bool Config.T

  val setup: theory -> theory
end


structure Sepref-Frame : SEPREF-FRAME = struct

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

  val DCONVERSION = Sepref-Debugging.DBG-CONVERSION cfg-debug
  val dbg-msg-tac = Sepref-Debugging.dbg-msg-tac cfg-debug


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

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

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


  local
    open Sepref-Basic Refine-Util Conv
```

```
  fun assn-ord p = case apply2 dest-hn-ctxt-opt p of
      (NONE,NONE) => EQUAL
    | (SOME -, NONE) => LESS
    | (NONE, SOME -) => GREATER
    | (SOME (-,a,-), SOME (-,a',-)) => Term-Ord.fast-term-ord (a,a')

in
  fun reorder-ctxt-conv ctxt ct = let
    val cert = Thm.cterm-of ctxt

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

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

  in
    thm
  end

  fun prepare-fi-conv ctxt ct = case Thm.term-of ct of
    (t as @{mpat ?P ⟹\<^sub>t ?Q}) => let

      (* Build table from abs-vars to ctxt *)
     val (Qm, Qum) = strip-star Q |> filter-out is-true |> List.partition is-hn-ctxt

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

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

      val Pum = filter-out is-true Pum
```

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

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

      val new-t = mk-entailst (P', Q')
      val goal-t = Logic.mk-equals (t,new-t)

      val goal-ctxt = Variable.declare-term goal-t ctxt

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

      val goal-ct = Thm.cterm-of ctxt goal-t

      val thm = Goal.prove-internal ctxt [] goal-ct (fn - => tac)

    in
      thm
    end
  | - => no-conv ct

end

fun is-merge @{mpat Trueprop (- ∨_A - ⟹_t -)} = true | is-merge - = false
fun is-gen-frame @{mpat Trueprop (- ⟹_t -)} = true | is-gen-frame - = false


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

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

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

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

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

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

  fun frame-loop-tac side-tac ctxt = let

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


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

```
    frame-loop-tac side-tac ctxt,
    solve-remainder-tac
  ]
end

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

val setup = normrel-eqs.setup

local
  open Sepref-Basic
  fun is-invalid @{mpat hn-invalid - - - :: assn} = true | is-invalid - = false
    fun contains-invalid @{mpat Trueprop (RECOVER-PURE ?Q -)} = exists
is-invalid (strip-star Q)
    | contains-invalid - = false

in
  fun recover-pure-tac ctxt =
    CONCL-COND' contains-invalid THEN-ELSE' (
      REPEAT-ALL-NEW (DETERM o (resolve-tac ctxt @{thms recover-pure}
ORELSE' Sepref-Constraints.constraint-tac ctxt)),
      resolve-tac ctxt @{thms recover-pure-triv}
    )
end

local
  open Sepref-Basic Refine-Util
  datatype cte = Other of term | Hn of term * term * term
  fun dest-ctxt-elem @{mpat hn-ctxt ?R ?a ?c} = Hn (R,a,c)
    | dest-ctxt-elem t = Other t

  fun mk-ctxt-elem (Other t) = t
    | mk-ctxt-elem (Hn (R,a,c)) = @{mk-term hn-ctxt ?R ?a ?c}

  fun match x (Hn (-,y,-)) = x aconv y
    | match - - = false

  fun dest-with-frame (*ctxt*) - t = let
    val (P,c,Q,R,a) = dest-hn-refine t
```

```
    val (-,(-,args)) = dest-hnr-absfun a
    val pre-ctes = strip-star P |> map dest-ctxt-elem

    val (pre-args,frame) =
      (case split-matching match args pre-ctes of
        NONE => raise TERM(align-conv: Could not match all arguments,[P,a])
      | SOME x => x)

  in
    ((frame,pre-args),c,Q,R,a)
  end

  fun align-goal-conv-aux ctxt t = let
    val ((frame,pre-args),c,Q,R,a) = dest-with-frame ctxt t
    val P' = apply2 (list-star o map mk-ctxt-elem) (frame,pre-args) |> mk-star
    val t' = mk-hn-refine (P',c,Q,R,a)
  in t' end

  fun align-rl-conv-aux ctxt t = let
    val ((frame,pre-args),c,Q,R,a) = dest-with-frame ctxt t

     val - = frame = [] orelse raise TERM (align-rl-conv: Extra preconditions in
rule,[t,list-star (map mk-ctxt-elem frame)])

    val P' = list-star (map mk-ctxt-elem pre-args)
    val t' = mk-hn-refine (P',c,Q,R,a)
  in t' end


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

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

  fun norm-goal-pre-conv ctxt = let
    open Conv

    fun conv ctxt = let
      val nr-conv = normrel-conv ctxt
    in
      hn-refine-conv nr-conv all-conv all-conv all-conv all-conv
    end
  in
    HOL-concl-conv conv ctxt
  end
```

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

    fun align-rl-conv ctxt = let
      open Conv
      fun conv ctxt = let
        val nr-conv = normrel-conv ctxt
      in
        hn-refine-conv nr-conv all-conv nr-conv nr-conv all-conv
      end
    in
     HOL-concl-conv (fn ctxt => f-tac-conv ctxt (align-rl-conv-aux ctxt) star-permute-tac)
ctxt
      then-conv HOL-concl-conv conv ctxt
    end

    fun align-goal-tac ctxt =
      CONCL-COND' is-hn-refine-concl
      THEN' DCONVERSION ctxt (HOL-concl-conv align-goal-conv ctxt)
  end


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

end
›

setup Sepref-Frame.setup

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


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



lemmas [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*($'a$) **by** *simp*

named-theorems-rev *intf-of-assn* ‹*Links between refinement assertions and interface types*›

lemma *intf-of-assn-fallback*: *intf-of-assn* (R :: $'a \Rightarrow - \Rightarrow assn$) *TYPE*($'a$) **by** *simp*

1.5.2 Function Refinement with Precondition

definition *fref* :: $('c \Rightarrow bool) \Rightarrow ('a \times 'c)\ set \Rightarrow ('b \times 'd)\ set$
$\Rightarrow (('a \Rightarrow 'b) \times ('c \Rightarrow 'd))\ set$
(‹[-]$_f$ - → -› [0,60,60] 60)
where $[P]_f\ R \to S \equiv \{(f,g).\ \forall x\ y.\ P\ y \wedge (x,y) \in R \longrightarrow (f\ x,\ g\ y) \in S\}$

abbreviation *freft* (‹- →$_f$ -› [60,60] 60) **where** $R \to_f S \equiv ([\lambda\text{-}.\ True]_f\ R \to S)$

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

lemma *fref-cons*:
assumes $(f,g) \in [P]_f\ R \to S$
assumes $\bigwedge c\ a.\ (c,a) \in R' \Longrightarrow Q\ a \Longrightarrow P\ a$
assumes $R' \subseteq R$
assumes $S \subseteq S'$
shows $(f,g) \in [Q]_f\ R' \to S'$
using *assms*
unfolding *fref-def*
by *fastforce*

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*
using *assms*
unfolding *fref-def*
by *auto*

lemma *fref-ncI*: $(f,g)\in R\to S \Longrightarrow (f,g)\in R\to_f S$
apply (*rule frefI*)
apply *parametricity*
done

lemma *frefD*:
assumes $(f,g)\in$*fref P R S*
shows $[\![P\ y;\ (x,y)\in R]\!] \Longrightarrow (f\ x,\ g\ y)\in S$
using *assms*
unfolding *fref-def*
by *auto*

lemma *fref-ncD*: $(f,g)\in R\to_f S \Longrightarrow (f,g)\in R\to S$
apply (*rule fun-relI*)
apply (*drule frefD*)
apply *simp*
apply *assumption+*
done

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)*
unfolding *fref-def*
apply (*auto*)
apply *blast*
done

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)*
using *fref-compI*[*of P R1 R2 Q S1 S2*]
by *auto*

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

lemma *fref-uncurry-conv*:
$(uncurry\ c,\ uncurry\ a) \in$ *fref P* $(R1\times_r R2)$ *S*

$\longleftrightarrow$ ($\forall$ *x1 y1 x2 y2*. *P* (*y1*,*y2*) $\longrightarrow$ (*x1*,*y1*)$\in$*R1* $\longrightarrow$ (*x2*,*y2*)$\in$*R2* $\longrightarrow$ (*c x1 x2*, *a y1 y2*) $\in$ *S*)
by (*auto simp*: *fref-def*)

lemma *fref-mono*: $[\![\bigwedge x.\ P'\ x \Longrightarrow P\ x;\ R' \subseteq R;\ S \subseteq S']\!]$
$\Longrightarrow$ *fref P R S* $\subseteq$ *fref P' R' S'*
unfolding *fref-def*
by *auto blast*

lemma *fref-composeI*:
assumes *FR1*: (*f*,*g*)$\in$*fref P R1 R2*
assumes *FR2*: (*g*,*h*)$\in$*fref Q S1 S2*
assumes *C1*: $\bigwedge x.\ P'\ x \Longrightarrow Q\ x$
assumes *C2*: $\bigwedge x\ y.\ [\![P'\ x;\ (y,x)\in S1]\!] \Longrightarrow P\ y$
assumes *R1*: *R'* $\subseteq$ *R1 O S1*
assumes *R2*: *R2 O S2* $\subseteq$ *S'*
assumes *FH*: *f'*=*f h'*=*h*
shows (*f'*,*h'*) $\in$ *fref P' R' S'*
unfolding *FH*
apply (*rule subsetD*[*OF fref-mono fref-compI'*[*OF FR1 FR2*]])
using *C1 C2* **apply** *blast*
using *R1* **apply** *blast*
using *R2* **apply** *blast*
done

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

1.5.3 Heap-Function Refinement

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

definition *hfref*
::
(*'a* $\Rightarrow$ *bool*)
$\Rightarrow$ ((*'a* $\Rightarrow$ *'ai* $\Rightarrow$ *assn*) $\times$ (*'a* $\Rightarrow$ *'ai* $\Rightarrow$ *assn*))
$\Rightarrow$ (*'b* $\Rightarrow$ *'bi* $\Rightarrow$ *assn*)
$\Rightarrow$ ((*'ai* $\Rightarrow$ *'bi Heap*) $\times$ (*'a*$\Rightarrow$*'b nres*)) *set*
($\langle$[-]$_a$ - $\rightarrow$ -$\rangle$ [*0*,*60*,*60*] *60*)
where
[*P*]$_a$ *RS* $\rightarrow$ *T* $\equiv$ { (*f*,*g*) . $\forall$ *c a*. *P a* $\longrightarrow$ *hn-refine* (*fst RS a c*) (*f c*) (*snd RS a c*) *T* (*g a*)}

abbreviation *hfreft* ($\langle$- $\rightarrow_a$ -$\rangle$ [*60*,*60*] *60*) **where** *RS* $\rightarrow_a$ *T* $\equiv$ ([λ-. *True*]$_a$ *RS* $\rightarrow$ *T*)

lemma *hfrefI*[*intro?*]:
assumes $\bigwedge$*c a*. *P a* $\Longrightarrow$ *hn-refine* (*fst RS a c*) (*f c*) (*snd RS a c*) *T* (*g a*)

shows $(f,g)\in$*hfref P RS T*
using *assms* **unfolding** *hfref-def* **by** *blast*

lemma *hfrefD*:
assumes $(f,g)\in$*hfref P RS T*
shows $\bigwedge c\ a.\ P\ a \Longrightarrow$ *hn-refine* (*fst RS a c*) (*f c*) (*snd RS a c*) *T* (*g a*)
using *assms* **unfolding** *hfref-def* **by** *blast*

lemma *hfref-to-ASSERT-conv*:
NO-MATCH (λ-. *True*) *P* $\Longrightarrow$ $(a,b)\in[P]_a\ R \to S \longleftrightarrow (a,\lambda x.\ ASSERT\ (P\ x) \gg b\ x) \in R \to_a S$
unfolding *hfref-def*
apply (*clarsimp*; *safe*; *clarsimp?*)
apply (*rule hn-refine-nofailI*)
apply (*simp add*: *refine-pw-simps*)
subgoal for *xc xa*
apply (*drule spec*[*of* - *xc*])
apply (*drule spec*[*of* - *xa*])
by *simp*
done

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

primrec *hf-pres*
:: $('a \Rightarrow 'b \Rightarrow assn) \Rightarrow bool \Rightarrow ('a \Rightarrow 'b \Rightarrow assn)\times('a \Rightarrow 'b \Rightarrow assn)$
where
hf-pres R True = (*R*,*R*) | *hf-pres R False* = (*R*,*invalid-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-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-pres R False*

abbreviation *hn-kede R kd* ≡ *hn-ctxt* (*snd* (*hf-pres R kd*))
abbreviation *hn-keep R* ≡ *hn-kede R True*
abbreviation *hn-dest R* ≡ *hn-kede R False*

lemma *keep-drop-sels*[*simp*]:
fst (R^k) = *R*
snd (R^k) = *R*
fst (R^d) = *R*
snd (R^d) = *invalid-assn R*
by *auto*

lemma *hf-pres-fst*[*simp*]: *fst* (*hf-pres R k*) = *R* **by** (*cases k*) *auto*

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

definition *hfprod* ::
(('*a* ⇒ '*b* ⇒ *assn*)×('*a* ⇒ '*b* ⇒ *assn*))
⇒ (('*c* ⇒ '*d* ⇒ *assn*)×('*c* ⇒ '*d* ⇒ *assn*))
⇒ ((('*a*×'*c*) ⇒ ('*b* × '*d*) ⇒ *assn*) × (('*a*×'*c*) ⇒ ('*b* × '*d*) ⇒ *assn*))
(**infixl** ‹*$_a$› 65)
where *RR* *$_a$ *SS* ≡ (*prod-assn* (*fst RR*) (*fst SS*), *prod-assn* (*snd RR*) (*snd SS*))

lemma *hfprod-fst-snd*[*simp*]:
fst (*A* *$_a$ *B*) = *prod-assn* (*fst A*) (*fst B*)
snd (*A* *$_a$ *B*) = *prod-assn* (*snd A*) (*snd B*)
unfolding *hfprod-def* **by** *auto*

Conversion from fref to hfref

lemma *fref-to-pure-hfref'*:
assumes (*f*,*g*) ∈ [*P*]$_f$ *R*→⟨*S*⟩*nres-rel*
assumes ⋀*x*. *x*∈*Domain R* ∩ R^{-1} ‘‘*Collect P* ⟹ *f x* = *RETURN* (*f' x*)
shows (*return o f'*, *g*) ∈ [*P*]$_a$ (*pure R*)^k→*pure S*
apply (*rule hfrefI*) **apply** (*rule hn-refineI*)
using *assms*
apply ((*sep-auto simp*: *fref-def pure-def pw-le-iff pw-nres-rel-iff*
refine-pw-simps eintros del: *exI*))
apply *force*
done

Conversion from hfref to hnr

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

lemma *hf2hnr*:
assumes (*f*,*g*) ∈ [*P*]$_a$ *R* → *S*
shows ∀*x xi*. *P x* ⟶ *hn-refine* (*emp* ∗ *hn-ctxt* (*fst R*) *x xi*) (*f*$*xi*) (*emp* ∗ *hn-ctxt* (*snd R*) *x xi*) *S* (*g*$*x*)
using *assms*
unfolding *hfref-def*
by (*auto simp*: *hn-ctxt-def*)

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

lemma *to-hnr-prod-fst-snd*:

fst (*A* $*_a$ *B*) = *to-hnr-prod* (*fst A*) (*fst B*)
snd (*A* $*_a$ *B*) = *to-hnr-prod* (*snd A*) (*snd B*)
unfolding *hfprod-def* **by** *auto*

lemma *hnr-uncurry-unfold*:
(∀ *x xi*. *P x* ⟶
hn-refine
(Γ ∗ *hn-ctxt* (*to-hnr-prod A B*) *x xi*)
(*fi xi*)
(Γ′ ∗ *hn-ctxt* (*to-hnr-prod A′ B′*) *x xi*)
R
(*f x*))
⟷ (∀ *b bi a ai*. *P* (*a*,*b*) ⟶
hn-refine
(Γ ∗ *hn-ctxt B b bi* ∗ *hn-ctxt A a ai*)
(*fi* (*ai*,*bi*))
(Γ′ ∗ *hn-ctxt B′ b bi* ∗ *hn-ctxt A′ a ai*)
R
(*f* (*a*,*b*))
)
by (*auto simp*: *hn-ctxt-def prod-assn-def star-aci*)

lemma *hnr-intro-dummy*:
∀ *x xi*. *P x* ⟶ *hn-refine* (Γ *x xi*) (*c xi*) (Γ′ *x xi*) *R* (*a x*) ⟹ ∀ *x xi*. *P x* ⟶ *hn-refine* (*emp*∗Γ *x xi*) (*c xi*) (*emp*∗Γ′ *x xi*) *R* (*a x*)
by *simp*

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

lemma *uncurry-APP*: *uncurry f*$(*a*,*b*) = *f*$*a*$*b* **by** *auto*

lemma *norm-RETURN-o*:
⋀*f*. (*RETURN o f*)$*x* = (*RETURN*$(*f*$*x*))
⋀*f*. (*RETURN oo f*)$*x*$*y* = (*RETURN*$(*f*$*x*$*y*))
⋀*f*. (*RETURN ooo f*)$*x*$*y*$*z* = (*RETURN*$(*f*$*x*$*y*$*z*))
⋀*f*. (λ*x*. *RETURN ooo f x*)$*x*$*y*$*z*$*a* = (*RETURN*$(*f*$*x*$*y*$*z*$*a*))
⋀*f*. (λ*x y*. *RETURN ooo f x y*)$*x*$*y*$*z*$*a*$*b* = (*RETURN*$(*f*$*x*$*y*$*z*$*a*$*b*))
by *auto*

lemma *norm-return-o*:
⋀*f*. (*return o f*)$*x* = (*return*$(*f*$*x*))
⋀*f*. (*return oo f*)$*x*$*y* = (*return*$(*f*$*x*$*y*))
⋀*f*. (*return ooo f*)$*x*$*y*$*z* = (*return*$(*f*$*x*$*y*$*z*))
⋀*f*. (λ*x*. *return ooo f x*)$*x*$*y*$*z*$*a* = (*return*$(*f*$*x*$*y*$*z*$*a*))
⋀*f*. (λ*x y*. *return ooo f x y*)$*x*$*y*$*z*$*a*$*b* = (*return*$(*f*$*x*$*y*$*z*$*a*$*b*))
by *auto*

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

Conversion from hnr to hfref

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

abbreviation *id-assn* ≡ *pure Id*
abbreviation (*input*) *unit-assn* ≡ *id-assn* :: *unit* ⇒ -

lemma *pure-unit-rel-eq-empty*: *unit-assn x y* = *emp*
by (*auto simp*: *pure-def*)

lemma *uc-hfprod-sel*:
fst (*A* ∗$_a$ *B*) *a c* = (*case* (*a*,*c*) *of* ((*a1*,*a2*),(*c1*,*c2*)) ⇒ *fst A a1 c1* ∗ *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*)
unfolding *hfprod-def prod-assn-def*[*abs-def*] **by** *auto*

Conversion from relation to fref

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

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

lemma *fref-param1*: *R*→*S* = *fref* (λ-. *True*) *R S*
by (*auto simp*: *fref-def fun-relD*)

lemma *fref-nest*: *fref P1 R1* (*fref P2 R2 S*)
≡ *CURRY* (*fref* (λ(*a*,*b*). *P1 a* ∧ *P2 b*) (*R1*×$_r$*R2*) *S*)
apply (*rule eq-reflection*)
by (*auto simp*: *fref-def CURRY-def*)

lemma *in-CURRY-conv*: (*f*,*g*) ∈ *CURRY R* ⟷ (*uncurry f*, *uncurry g*) ∈ *R*
unfolding *CURRY-def* **by** *auto*

lemma *uncurry0-APP*[*simp*]: *uncurry0 c* $ *x* = *c* **by** *auto*

lemma *fref-param0I*: (*c*,*a*)∈*R* ⟹ (*uncurry0 c*, *uncurry0 a*) ∈ *fref* (λ-. *True*) *unit-rel R*
by (*auto simp*: *fref-def*)

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

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

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

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

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

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

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

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

lemma *hr-comp-the-pure*: *is-pure A* $\Longrightarrow$ *the-pure* (*hr-comp A B*) = *the-pure A O B*
unfolding *is-pure-conv*
by (*clarsimp simp*: *hr-comp-pure*)

lemma *rdomp-hrcomp-conv*: *rdomp* (*hr-comp A R*) *x* $\longleftrightarrow$ ($\exists$ *y*. *rdomp A y* $\wedge$ (*y*,*x*)$\in$*R*)
by (*auto simp*: *rdomp-def hr-comp-def*)

lemma *hn-rel-compI*:
⟦*nofail a*; (*b*,*a*)$\in$⟨*R2*⟩*nres-rel*⟧ $\Longrightarrow$ *hn-rel R1 b c* $\Longrightarrow_A$ *hn-rel* (*hr-comp R1 R2*) *a c*
unfolding *hr-comp-def hn-rel-def nres-rel-def*
apply (*clarsimp intro*!: *ent-ex-preI*)
apply (*drule* (*1*) *order-trans*)
apply (*simp add*: *ret-le-down-conv*)
by *sep-auto*

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

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

lemma *hnr-comp*:
assumes *R*: $\bigwedge$*b1 c1*. *P b1* $\Longrightarrow$ *hn-refine* (*R1 b1 c1* $*$ Γ) (*c c1*) (*R1p b1 c1* $*$ Γ') *R* (*b b1*)
assumes *S*: $\bigwedge$*a1 b1*. ⟦*Q a1*; (*b1*,*a1*)$\in$*R1'*⟧ $\Longrightarrow$ (*b b1*,*a a1*)$\in$⟨*R'*⟩*nres-rel*
assumes *PQ*: $\bigwedge$*a1 b1*. ⟦*Q a1*; (*b1*,*a1*)$\in$*R1'*⟧ $\Longrightarrow$ *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)
   unfolding hn-refine-alt
  proof clarsimp
   assume NF: nofail (a a1)
   show
    <hr-comp R1 R1' a1 c1 * Γ>
     c c1
    <λr. hn-rel (hr-comp R R') (a a1) r * (hr-comp R1p R1' a1 c1 * Γ')>t
    apply (subst hr-comp-def)
    apply (clarsimp intro!: norm-pre-ex-rule)
   proof −
    fix b1
    assume R1: (b1, a1) ∈ R1'

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

    from PQ R1 Q have P: P b1 by blast
    with NFB R have <R1 b1 c1 * Γ> c c1 <λr. hn-rel R (b b1) r * (R1p b1
c1 * Γ')>t
     unfolding hn-refine-alt by auto
    thus <R1 b1 c1 * Γ>
     c c1
     <λr. hn-rel (hr-comp R R') (a a1) r * (hr-comp R1p R1' a1 c1 * Γ')>t
     apply (rule cons-post-rule)
     apply (solve-entails)
    by (intro ent-star-mono hn-rel-compI[OF NF R'] hr-compI[OF R1] ent-refl)
   qed
  qed

  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)
   using assms hnr-comp[where Γ=emp and Γ'=emp and a=a and b=b and
c=c and P=P and Q=Q]
   unfolding hn-ctxt-def
   by auto

  lemma hfcomp:
```

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 $(f,h) \in [\lambda a.\ Q\ a \wedge (\forall a'.\ (a',a) \in T \longrightarrow P\ a')]_a$
hrp-comp RR' T $\to$ *hr-comp S U*
using *assms*
unfolding *fref-def hfref-def hrp-comp-def*
apply *clarsimp*
apply (*rule hnr-comp1-aux*[*of*
P fst RR' f snd RR' S g $\lambda a.\ Q\ a \wedge (\forall a'.\ (a',a) \in T \longrightarrow P\ a')\ T\ h\ U$])
apply (*auto simp*: *hn-ctxt-def*)
done

lemma *hfref-weaken-pre-nofail*:
assumes $(f,g) \in [P]_a\ R \to S$
shows $(f,g) \in [\lambda x.\ nofail\ (g\ x) \longrightarrow P\ x]_a\ R \to S$
using *assms*
unfolding *hfref-def hn-refine-def*
by *auto*

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

Composition Automation

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

lemma *prod-hrp-comp*:
hrp-comp $(A *_a B)\ (C \times_r D) =$ *hrp-comp A C* $*_a$ *hrp-comp B D*
unfolding *hrp-comp-def hfprod-def* **by** *simp*

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

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

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

definition *hrp-imp RR RR'* ≡
∀ *a b*. (*fst RR' a b* ⟹$_t$ *fst RR a b*) ∧ (*snd RR a b* ⟹$_t$ *snd RR' a b*)

lemma *hfref-imp*: *hrp-imp RR RR'* ⟹ [*P*]$_a$ *RR* → *S* ⊆ [*P*]$_a$ *RR'* → *S*
apply *clarsimp*
apply (*erule hfref-cons*)
apply (*simp-all add*: *hrp-imp-def*)
done

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

lemma *hrp-imp-reflI*: *RR* = *RR'* ⟹ *hrp-imp RR RR'*
unfolding *hrp-imp-def* **by** *auto*

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

lemma *hrp-prod-cong*: *hrp-imp A A'* ⟹ *hrp-imp B B'* ⟹ *hrp-imp* (*A*∗$_a$*B*) (*A'*∗$_a$*B'*)
by (*sep-auto simp*: *hrp-imp-def prod-assn-def intro*: *entt-star-mono*)

lemma *hrp-imp-trans*: *hrp-imp A B* ⟹ *hrp-imp B C* ⟹ *hrp-imp A C*
unfolding *hrp-imp-def*
by (*fastforce intro*: *entt-trans*)

lemma *fcomp-norm-dflt-init*: *x*∈[*P*]$_a$ *R* → *T* ⟹ *hrp-imp R S* ⟹ *x*∈[*P*]$_a$ *S* → *T*
apply (*erule rev-subsetD*)
by (*rule hfref-imp*)

definition *comp-PRE R P Q S* ≡ λ*x*. *S x* ⟶ (*P x* ∧ (∀ *y*. (*y*,*x*)∈*R* ⟶ *Q x y*))

lemma *comp-PRE-cong*[*cong*]:
assumes *R*≡*R'*
assumes ⋀*x*. *P x* ≡ *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'*
using *assms*
by (*fastforce simp*: *comp-PRE-def intro*!: *eq-reflection ext*)

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

lemma *PRE-D1*: $(Q\ x \wedge P\ x) \longrightarrow comp\text{-}PRE\ S1\ Q\ (\lambda x\ \text{-}.\ P\ x)\ S\ x$
by (*auto simp*: *comp-PRE-def*)

lemma *PRE-D2*: $(Q\ x \wedge (\forall y.\ (y,x)\in S1 \longrightarrow S\ x \longrightarrow P\ x\ y)) \longrightarrow comp\text{-}PRE\ S1\ Q\ P\ S\ x$
by (*auto simp*: *comp-PRE-def*)

lemma *fref-weaken-pre*:
assumes $\bigwedge x.\ P\ x \longrightarrow P'\ x$
assumes $(f,h) \in fref\ P'\ R\ S$
shows $(f,h) \in fref\ P\ R\ S$
apply (*rule rev-subsetD*[*OF assms*(*2*) *fref-mono*])
using *assms*(*1*) **by** *auto*

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

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

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 $(f,h) \in hfref\ P\ R\ S$
using *assms*
by (*auto simp*: *hfref-def*)

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 $(f,h) \in hfref\ P\ R\ S$
apply (*rule hfrefI*)
apply (*rule hn-refine-preI*)
using *assms*
by (*auto simp*: *hfref-def rdomp-def*)

lemma *hfref-weaken-pre-nofail'*:
assumes $(f,g) \in [P]_a\ R \to S$
assumes $\bigwedge x.\ [\![nofail\ (g\ x);\ Q\ x]\!] \Longrightarrow P\ x$
shows $(f,g) \in [Q]_a\ R \to S$
apply (*rule hfref-weaken-pre*[*OF - assms*(*1*)[*THEN hfref-weaken-pre-nofail*]])
using *assms*(*2*)
by *blast*

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

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

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

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

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

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

1.5.4 Automation

Purity configuration for constraint solver

lemmas [*safe-constraint-rules*] = *pure-pure*

Configuration for hfref to hnr conversion

named-theorems *to-hnr-post* ‹*to-hnr converter*: *Postprocessing unfold rules*›

lemma *uncurry0-add-app-tag*: *uncurry0* (*RETURN c*) = *uncurry0* (*RETURN*$*c*)
by *simp*

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*) **by** *auto*
lemma *uncurry0-hfref-post*[*to-hfref-post*]: *hfref* (*uncurry0 True*) *R S* = *hfref* (λ-. *True*) *R S*
apply (*fo-rule arg-cong fun-cong*)+ **by** *auto*

Configuration for relation normalization after composition

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

Default Setup

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

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

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

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

lemma *ensure-fref-nresI*: $(f,g)\in[P]_f\ R\to S \Longrightarrow (RETURN\ o\ f,\ RETURN\ o\ g)\in[P]_f\ R\to\langle S\rangle nres\text{-}rel$
by (*auto intro*: *nres-relI simp*: *fref-def*)

lemma *ensure-fref-nres-unfold*:
$\bigwedge f.\ RETURN\ o\ (uncurry0\ f) = uncurry0\ (RETURN\ f)$
$\bigwedge f.\ RETURN\ o\ (uncurry\ f) = uncurry\ (RETURN\ oo\ f)$
$\bigwedge f.\ (RETURN\ ooo\ uncurry)\ f = uncurry\ (RETURN\ ooo\ f)$
by *auto*

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 $\bigwedge x.\ PROTECT\ P\ x \longrightarrow P'\ x$
assumes $(f,h) \in hfref\ P'\ R\ S$
shows $(f,h) \in hfref\ P\ R\ S$
using *assms*
by (*auto simp*: *hfref-def*)

lemma *auto-weaken-pre-init-f*:
assumes $\bigwedge x.\ PROTECT\ P\ x \longrightarrow P'\ x$
assumes $(f,h) \in fref\ P'\ R\ S$
shows $(f,h) \in fref\ P\ R\ S$
using *assms*
by (*auto simp*: *fref-def*)

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 $PROTECT\ (\lambda(x,y).\ f\ x\ y)\ (a,b) \equiv f'\ b$
using *assms*
by (*auto simp*: *curry-def dest*!: *meta-eq-to-obj-eq intro*!: *eq-reflection*)

lemma *auto-weaken-pre-uncurry-finish*:
$PROTECT\ f\ x \equiv f\ x$ **by** (*auto*)

lemma *auto-weaken-pre-uncurry-start*:
assumes $P \equiv P'$
assumes $P' \longrightarrow Q$
shows $P \longrightarrow Q$
using *assms* **by** (*auto*)

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*
using *assms* **by** (*auto simp*: *comp-PRE-def*)

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

lemma *auto-weaken-pre-add-dummy-imp*:
$P \Longrightarrow True \longrightarrow P$ **by** *simp*

Synthesis for hfref statements

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

lemma *hfsynth-ID-R-D*:
fixes $I :: 'a\ itself$
assumes *hfsynth-ID-R R a*
assumes *intf-of-assn R I*
shows $a ::_i I$
by *simp*

lemma *hfsynth-hnr-from-hfI*:
assumes $\forall x\ xi.\ P\ x \wedge$ *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 $(f,g) \in [P]_a\ R \to S$
using *assms*
unfolding *hfref-def*
by (*auto simp*: *hn-ctxt-def*)

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

ML ‹

signature SEPREF-RULES = sig
(∗ *Analysis of relations, both fref and fun-rel* ∗)
(∗ *R1→...→Rn→- / [-]$_f$ ((R1×$_r$R2)...×$_r$Rn) ↦ [R1,...,Rn]* ∗)
val binder-rels: *term −> term list*
(∗ *-→...→-→S / [-]$_f$ - → S ↦ S* ∗)
val body-rel: *term −> term*
(∗ *Map →/fref to (precond,args,res). NONE if no/trivial precond.* ∗)
val analyze-rel: *term −> term option ∗ term list ∗ term*

```
(* Make trivial (λ-. True) precond *)
val mk-triv-precond: term list -> term
(* Make [P]_f ((R1×_rR2)...×_rRn) → S. Insert trivial precond if NONE. *)
val mk-rel: term option * term list * term -> term
(* Map relation to (args,res) *)
val strip-rel: term -> term list * term

(* Make hfprod (op *_a) *)
val mk-hfprod : term * term -> term
val mk-hfprods : term list -> term

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

(*
  Convert a parametricity theorem in higher-order form to
  uncurried fref-form. For functions without arguments,
  a unit-argument is added.

  TODO/FIXME: Currently this only works for higher-order theorems,
    i.e., theorems of the form (f,g)∈R1→...→Rn.

    First-order theorems are silently treated as refinement theorems
    for functions with zero arguments, i.e., a unit-argument is added.
*)
val to-fref : Proof.context -> thm -> thm

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

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

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

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

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


type hnr-analysis = {
  thm: thm,                    (* Original theorem, may be normalized *)
```

```
    precond: term,                (* Precondition, abstracted over abs−arguments
*)
    prems : term list,            (* Premises not depending on arguments *)
    ahead: term * bool,           (* Abstract function, has leading RETURN *)
    chead: term * bool,           (* Concrete function, has leading return *)
    argrels: (term * bool) list,  (* Argument relations, preserved (keep−flag) *)
    result-rel: term              (* Result relation *)
  }

  val analyze-hnr: Proof.context −> thm −> hnr-analysis
  val pretty-hnr-analysis: Proof.context −> hnr-analysis −> Pretty.T
  val mk-hfref-thm: Proof.context −> hnr-analysis −> thm



  (* Simplify precondition of fref/hfref−theorem *)
  val simplify-precond: Proof.context −> thm −> thm

  (* Normalize hfref−theorem after composition *)
  val norm-fcomp-rule: Proof.context −> thm −> thm

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

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

  (* FCOMP−attribute *)
  val fcomp-attrib: attribute context-parser
end

structure Sepref-Rules: SEPREF-RULES = struct

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

    local
      fun br-aux @{mpat - → ?G} = br-aux G
        | br-aux R = R
    in
      fun body-rel @{mpat fref - - ?G} = G
        | body-rel R = br-aux R
    end
```

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

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

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

fun mk-rel (P,Rs,S) = let
  val R = list-prodrel-left Rs

  val P = case P of
      SOME P => P
    | NONE => mk-triv-precond Rs

in
  @{mk-term fref ?P ?R ?S}
end
end


fun mk-hfprod (a, b) = @{mk-term ?a*_a ?b}

local
  fun mk-hfprods-rev [] = @{mk-term unit-assn^k}
    | mk-hfprods-rev [Rk] = Rk
    | mk-hfprods-rev (Rkn::Rks) = mk-hfprod (mk-hfprods-rev Rks, Rkn)
in
  val mk-hfprods = mk-hfprods-rev o rev
end


fun intf-of-assn ctxt t = let
  val orig-ctxt = ctxt
  val (t,ctxt) = yield-singleton (Variable.import-terms false) t ctxt

 val v = TVar ((T,0),Proof-Context.default-sort ctxt (T,0)) |> Logic.mk-type
  val goal = @{mk-term Trueprop (intf-of-assn ?t ?v)}

  val i-of-assn-rls =
    Named-Theorems-Rev.get ctxt @{named-theorems-rev intf-of-assn}
    @ @{thms intf-of-assn-fallback}
```

```
    fun tac ctxt = REPEAT-ALL-NEW (resolve-tac ctxt i-of-assn-rls)

    val thm = Goal.prove ctxt [] [] goal (fn {context,...} => ALLGOALS (tac
context))
    val intf = case Thm.concl-of thm of
        @{mpat Trueprop (intf-of-assn - (?v AS_p TYPE (-)))} => v
      | - => raise THM(Intf-of-assn: Proved a different theorem?,~1,[thm])

    val intf = singleton (Variable.export-terms ctxt orig-ctxt) intf
      |> Logic.dest-type

  in
    intf
  end

  datatype rthm-type =
    RT-HOPARAM    (* (-,-) ∈ - → ... → - *)
  | RT-FREF       (* (-,-) ∈ [-]_f - → - *)
  | RT-HNR        (* hn-refine - - - - - *)
  | RT-HFREF      (* (-,-) ∈ [-]_a - → - *)
  | RT-OTHER

  fun rthm-type thm =
    case Thm.concl-of thm |> HOLogic.dest-Trueprop of
      @{mpat (-,-) ∈ fref - - -} => RT-FREF
    | @{mpat (-,-) ∈ hfref - - -} => RT-HFREF
    | @{mpat hn-refine - - - - -} => RT-HNR
      | @{mpat (-,-) ∈ -} => RT-HOPARAM (* TODO: Distinction between
ho−param and fo−param *)
    | - => RT-OTHER


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

  fun to-foparam ctxt thm = let
    val unf-thms = @{thms
   split-tupled-all prod-rel-simp uncurry-apply cnv-conj-to-meta Product-Type.split}
  in
```

```
    case Thm.concl-of thm of
      @{mpat Trueprop ((-,-) ∈ fref - - -)} =>
        (@{thm frefD} OF [thm])
        |> Thm.forall-intr-vars
        |> Local-Defs.unfold0 ctxt unf-thms
        |> Variable.gen-all ctxt
    | @{mpat Trueprop ((-,-) ∈ -)} =>
        Parametricity.fo-rule thm
    | - => raise THM(Expected parametricity or fref theorem,~1,[thm])
  end

  fun to-hnr ctxt thm =
    (thm RS @{thm hf2hnr})
      |> Local-Defs.unfold0 ctxt @{thms to-hnr-prod-fst-snd keep-drop-sels} (*
Resolve fst and snd over *_a and R^k, R^d *)
      |> Local-Defs.unfold0 ctxt @{thms hnr-uncurry-unfold} (* Resolve products
for uncurried parameters *)
      |> Local-Defs.unfold0 ctxt @{thms uncurry-apply uncurry-APP assn-one-left
split} (* Remove the uncurry modifiers, the emp−dummy, and unfold product cases
*)
      |> Local-Defs.unfold0 ctxt @{thms hn-ctxt-ctxt-fix-conv} (* Remove duplicate
hn-ctxt tagging *)
    |> Local-Defs.unfold0 ctxt @{thms all-to-meta imp-to-meta HOL.True-implies-equals
HOL.implies-True-equals Pure.triv-forall-equality cnv-conj-to-meta} (* Convert to
meta−level, remove vacuous condition *)
      |> Local-Defs.unfold0 ctxt (Named-Theorems.get ctxt @{named-theorems
to-hnr-post}) (* Post−Processing *)
      |> Goal.norm-result ctxt
      |> Conv.fconv-rule Thm.eta-conversion

  (* Convert schematic hfref−goal to hn-refine goal *)
  fun prepare-hfref-synth-tac ctxt = let
    val i-of-assn-rls =
      Named-Theorems-Rev.get ctxt @{named-theorems-rev intf-of-assn}
      @ @{thms intf-of-assn-fallback}

    val to-hnr-post-rls =
      Named-Theorems.get ctxt @{named-theorems to-hnr-post}

    val i-of-assn-tac = (
      REPEAT' (
        DETERM o dresolve-tac ctxt @{thms hfsynth-ID-R-D}
         THEN' DETERM o SOLVED' (REPEAT-ALL-NEW (resolve-tac ctxt
i-of-assn-rls))
      )
    )
  in
    (* Note: To re−use the to-hnr infrastructure, we first work with
      $−tags on the abstract function, which are finally removed.
```

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


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

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


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

    (* Debug information: Stores string*term pairs, which are pretty−printed on
error *)
    val dbg = Unsynchronized.ref []
    fun add-dbg msg ts = (
      dbg := (msg,ts) :: !dbg;
      ()
```

```
)
fun pretty-dbg (msg,ts) = Pretty.block [
  Pretty.str msg,
  Pretty.str :,
  Pretty.brk 1,
  Pretty.list [ ] (map (Syntax.pretty-term ctxt) ts)
]
fun pretty-dbgs l = map pretty-dbg l |> Pretty.fbreaks |> Pretty.block

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

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


(* Heads may have a leading return/RETURN.
  The following code strips off the leading return, unless it has the form
  return x for an argument x
*)
fun check-strip-leading args t f = (* Handle the case RETURN x, where x is
an argument *)
  if Termtab.defined args f then (t,false) else (f,true)

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

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


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

  Moreover, it only strips actual arguments that occur in the
  precondition-section of the hn-refine-statement. This ensures
  that non-arguments, like maxsize, are treated correctly.
*)
fun strip-fun - (t as @{mpat PR-CONST -}) = (t,[])
  | strip-fun s (t as @{mpat ?f$?x}) = check-arg s t f x
  | strip-fun s (t as @{mpat ?f ?x}) = check-arg s t f x
  | strip-fun - f = (f,[])
and check-arg s t f x =
```

```
    if Termtab.defined s x then
      strip-fun s f |> apsnd (curry op :: x)
    else (t,[])

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


fun dest-hn-refine @{mpat (hn-refine ?G ?c ?G' ?R ?a)} = (G,c,G',R,a)
  | dest-hn-refine - = fail Conclusion is not a hn-refine statement

(*
  Strip separation conjunctions. Special case for emp, which is ignored.
*)
fun is-emp @{mpat emp} = true | is-emp - = false

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

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


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

(* Make HOL conjunction list *)
fun mk-conjs [] = Const ‹True›
  | mk-conjs [p] = p
  | mk-conjs (p::ps) = Const ‹conj for p ‹mk-conjs ps››


(**********************)
(* Start actual analysis *)

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

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

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

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

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

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

val - = add-dbg stripped abstract term [a]
val - = add-dbg stripped concrete term [c]

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

val - = add-dbg abstract head [ahead]
val - = add-dbg abstract args aargs
val - = add-dbg concrete head [chead]
val - = add-dbg concrete args cargs


val - = assert (length cargs = length aargs) Different number of abstract and
concrete arguments;

    val - = assert (not (has-duplicates op aconv aargs)) Duplicate abstract
arguments
    val - = assert (not (has-duplicates op aconv cargs)) Duplicate concrete
arguments

val argpairs = aargs ~~ cargs
val ap-set = Termtab2.make-set argpairs
val - = assert (pairs-eq pre-pairs ap-set) Arguments from pre/postcondition
do not match operation's arguments

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

val - = add-dbg pre-rels pre-rels
val - = add-dbg post-rels post-rels

fun adjust-hf-pres @{mpat snd (?R^k)} = R
  | adjust-hf-pres t = t
```

```
    val post-rels = map adjust-hf-pres post-rels

    fun is-invalid R @{mpat invalid-assn ?R'} = R aconv R'
      | is-invalid - @{mpat snd (-ᵈ)} = true
      | is-invalid - - = false

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

    val keep = map is-keep (pre-rels ~~ post-rels)

    val argrels = pre-rels ~~ keep

    val aa-set = Termtab.make-set aargs
    val ca-set = Termtab.make-set cargs

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

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

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

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

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

  fun pretty-hnr-analysis
    ctxt
    ({thm,precond,ahead,chead,argrels,result-rel,...})
    : Pretty.T =
```

```
  let
    val - = thm (* Suppress unused warning for thm *)

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

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

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

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

  end


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

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

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

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

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

  TODO: Combine with abstraction and  uncurry−procedure,
    and mark the deviation about uncurry as redundant
    intermediate step to be eliminated.
*)
fun rew-uncurry-lambda t = let
    val rr = map (Logic.dest-equals o Thm.prop-of) @{thms uncurry-def
uncurry0-def}
  val thy = Proof-Context.theory-of ctxt
in
  Pattern.rewrite-term-topdown thy rr [] t
end

(* Shortcuts for simplification tactics *)
fun gsimp-only ctxt sec = let
  val ss = put-simpset HOL-basic-ss ctxt |> sec
in asm-full-simp-tac ss end

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


(******************************)
(* Build theorem statement *)
(* ⟦prems⟧ ⟹ (chead,ahead) ∈ [precond] rels → R *)

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

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

fun mk-return (t,r) = if r then
    let
      val T = funpow num-args range-type (fastype-of (fst chead))
    val tRETURN = Const (@{const-name return}, T --> Type(@{type-name
```

```
Heap},[T]))
          in
            Refine-Util.mk-compN num-args tRETURN t
          end
        else t

      (* Hrmpf!: Gone for good from 2015→2016. Inserting ctxt−based substitute
here. *)
      fun certify-inst ctxt (instT, inst) =
       (TVars.map (K (Thm.ctyp-of ctxt)) instT,
        Vars.map (K (Thm.cterm-of ctxt)) inst);

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

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

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

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

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

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

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

      (* Tactic to prove the theorem.
       A first step uses hfrefI to get a hnr−goal.
       This is then normalized in several consecutive steps, which
        get rid of uncurrying. Finally, the original theorem is used for resolution,
```

```
      where the pre− and postcondition, and result relation are connected with
      a consequence rule, to handle unfolded hn-ctxt−tags, re−ordered relations,
      and introduced unit−parameters (TODO:
      Mark artificially introduced unit−parameter specially, it may get confused

        with intentional unit−parameter, e.g., functional empty-set ()!)

    *)
    fun tac ctxt =
          resolve-tac ctxt @{thms hfrefI}
      THEN' gsimp-only ctxt (fn c => c
        addsimps @{thms uncurry-def hn-ctxt-def uncurry0-def
                        keep-drop-sels uc-hfprod-sel o-apply
                        APP-def}
        |> Splitter.add-split @{thm prod.split}
      )

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

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

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

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

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

  in
    rthm
  end

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

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

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

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

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

     val norm = changed-rule (try-rule norm1 o try-rule norm2 o try-rule
norm3)
  in
    repeat-rule norm
  end
end

fun add-pure-constraints-rule ctxt thm = let
  val orig-ctxt = ctxt

  val t = Thm.prop-of thm

  fun
    cnv (@{mpat (typs) pure (mpaq-STRUCT (mpaq-Var ?x -) :: (?'v-c×?'v-a)
set)}) =
      let
        val T = a --> c --> @{typ assn}
        val t = Var (x,T)
        val t = @{mk-term (the-pure ?t)}
      in
```

```
      [(x,T,t)]
    end
  | cnv (t$u) = union op= (cnv t) (cnv u)
  | cnv (Abs (-,-,t)) = cnv t
  | cnv - = []

  val pvars = cnv t

  val - = (pvars |> map #1 |> has-duplicates op=)
    andalso raise TERM (Duplicate indexname with different type,[t]) (* This
should not happen *)

  val substs = map (fn (x,-,t) => (x,t)) pvars

  val t' = subst-Vars substs t

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

  val assms = map mk-asm pvars

  fun add-prems prems t = let
    val prems' = Logic.strip-imp-prems t
    val concl = Logic.strip-imp-concl t
  in
    Logic.list-implies (prems@prems', concl)
  end

  val t' = add-prems assms t'

  val (t',ctxt) = yield-singleton (Variable.import-terms true) t' ctxt

  val thm' = Goal.prove-internal ctxt [] (Thm.cterm-of ctxt t') (fn - =>
    ALLGOALS (resolve-tac ctxt [thm] THEN-ALL-NEW assume-tac ctxt))

  val thm' = norm-fcomp-rule ctxt thm'

  val thm' = singleton (Variable.export ctxt orig-ctxt) thm'
in
  thm'
end


val cfg-simp-precond =
  Attrib.setup-config-bool @{binding fcomp-simp-precond} (K true)
```

```
  local
    fun mk-simp-thm ctxt t = let
      val st = t
        |> HOLogic.mk-Trueprop
        |> Thm.cterm-of ctxt
        |> Goal.init

      val ctxt = Context-Position.set-visible false ctxt
      val ctxt = ctxt addsimps (
          refine-pw-simps.get ctxt
        @ Named-Theorems.get ctxt @{named-theorems fcomp-prenorm-simps}
        @ @{thms split-tupled-all cnv-conj-to-meta}
        )

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

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

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

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

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

      val thm = case Thm.concl-of thm of
        @{mpat Trueprop (- ⟶ -)} => thm
    | @{mpat Trueprop -} => thm RS @{thm auto-weaken-pre-add-dummy-imp}

      | - => raise THM(Simp-Precond: Generated odd theorem, expected form
'P⟶Q',~1,[thm])
```

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

      fun rec-uncurry thm =
       case try (fn () => thm OF @{thms auto-weaken-pre-uncurry-step}) () of
          NONE => thm OF @{thms auto-weaken-pre-uncurry-finish}
        | SOME thm => rec-uncurry thm

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

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

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

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

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

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

  end

  (* fref O fref *)
  fun compose-ff ctxt A B =
      (@{thm fref-compI-PRE} OF [A,B])
```

```
    |> norm-fcomp-rule ctxt
    |> simplify-precond-if-cfg ctxt
    |> Conv.fconv-rule Thm.eta-conversion

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

  fun ensure-fref ctxt thm = case rthm-type thm of
    RT-HOPARAM => to-fref ctxt thm
  | RT-FREF => thm
  | - => raise THM(Expected parametricity or fref theorem,~1,[thm])

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

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

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

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

  end
```

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

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

    end
  ›

  attribute-setup to-fref = ‹
    Scan.succeed (Thm.rule-attribute [] (Sepref-Rules.to-fref o Context.proof-of))
› Convert parametricity theorem to uncurried fref−form

  attribute-setup to-foparam = ‹
    Scan.succeed (Thm.rule-attribute [] (Sepref-Rules.to-foparam o Context.proof-of))
› ‹Convert param or fref rule to first order rule›

  attribute-setup param-fo = ‹
    Scan.succeed (Thm.rule-attribute [] (Sepref-Rules.to-foparam o Context.proof-of))
› ‹Convert param or fref rule to first order rule›

  attribute-setup to-hnr = ‹
    Scan.succeed (Thm.rule-attribute [] (Sepref-Rules.to-hnr o Context.proof-of))
› Convert hfref−rule to hnr−rule

  attribute-setup to-hfref = ‹Scan.succeed (
      Thm.rule-attribute [] (Context.proof-of #> Sepref-Rules.to-hfref)
    )› ‹Convert hnr to hfref theorem›


  attribute-setup ensure-fref-nres = ‹Scan.succeed (
      Thm.rule-attribute [] (Context.proof-of #> Sepref-Rules.ensure-fref-nres)
    )›

  attribute-setup sepref-dbg-norm-fcomp-rule = ‹Scan.succeed (
      Thm.rule-attribute [] (Context.proof-of #> Sepref-Rules.norm-fcomp-rule)
    )›

  attribute-setup sepref-simplify-precond = ‹Scan.succeed (
      Thm.rule-attribute [] (Context.proof-of #> Sepref-Rules.simplify-precond)
    )› ‹Simplify precondition of fref/hfref−theorem›

  attribute-setup FCOMP = Sepref-Rules.fcomp-attrib Composition of refine-
ment rules
```

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* ⇒ *unit list* ⇒ *unit* **where** *T - - ≡ ()*
  private lemma *unit-eq*: (*a*::*unit*) ≡ *b* **by** *simp*
  named-theorems *--itype-rewrite*

  ML ‹
    signature SEPREF-INTF-TYPES = sig
      (∗ Declare new interface type ∗)
     val decl-intf-type-cmd: ((*string list ∗ binding*) *∗ mixfix*) *∗ string −> local-theory −> local-theory*
      (∗ Register interface type rewrite rule ∗)
      val register-itype-rewrite: *typ −> typ −> Proof.context −> local-theory*

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

      (∗ Check whether interface type matches operation's type ∗)
      val check-intf-type: *Proof.context −> typ −> typ −> bool*
     (∗ Invoke msg with (normalized) non−matching types in case of no−match ∗)
      val check-intf-type-msg: (*typ ∗ typ −> unit*) *−> Proof.context −> typ −> typ −> unit*
      (∗ Trigger error message if no match ∗)
      val check-intf-type-err: *Proof.context −> typ −> typ −> unit*

    end

    structure Sepref-Intf-Types: *SEPREF-INTF-TYPES = struct*
      fun t2t (*Type*(*name*,*args*)) =
        @{*term T*}
          $*HOLogic.mk-string name*
          $*HOLogic.mk-list* @{*typ unit*} (*map t2t args*)
      | *t2t* (*TFree* (*name*,-)) = *Var* ((*F^name*,*0*),*HOLogic.unitT*)

```
  | t2t (TVar ((name,i),-)) = Var ((V^name,i),HOLogic.unitT)

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

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

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

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

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

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

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

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

  fun norm-intf-typet ctxt T = let
```

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

fun norm-intf-type ctxt T = norm-intf-typet ctxt T |> tt2

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

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

fun check-intf-type-err ctxt iT cT = let
  fun msg (iT',cT') = Pretty.block [
    Pretty.str Interface type and logical type do not match,
    Pretty.fbrk,
    Pretty.str Interface: ,Syntax.pretty-typ ctxt iT, Pretty.brk 1,
    Pretty.str   is   , Syntax.pretty-typ ctxt iT', Pretty.fbrk,
    Pretty.str Logical: ,Syntax.pretty-typ ctxt cT, Pretty.brk 1,
    Pretty.str   is   , Syntax.pretty-typ ctxt cT', Pretty.fbrk
  ] |> Pretty.string-of |> error

in
  check-intf-type-msg msg ctxt iT cT
end

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

›

end

1.6.2 Rewriting Inferred Interface Types

definition *map-type-eq* :: *'a itself* ⇒ *'b itself* ⇒ *bool*
(**infixr** ‹$\rightarrow_{nt}$› *60*)
where [*simp*]: *map-type-eq - - ≡ True*
lemma *map-type-eqI*: *map-type-eq L R* **by** *auto*

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

1.6.3 ML-Code

context begin

private lemma *start-eval*: *x ≡ SP x* **by** *auto*
private lemma *add-eval*: *f x ≡ (≫=)$(EVAL$x)$(λ2x. f x)* **by** *auto*

private lemma *init-mk-arity*: *f ≡ id (SP f)* **by** *simp*
private lemma *add-mk-arity*: *id f ≡ (λ2x. id (f$x))* **by** *auto*
private lemma *finish-mk-arity*: *id f ≡ f* **by** *simp*

ML ‹
structure Sepref-Combinator-Setup = struct

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

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

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

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

    val lhs = Autoref-Tagging.list-APP (t,vars)
      |> Thm.cterm-of ctxt

    fun add-EVAL x thm =
      case Thm.prop-of thm of
        @{mpat - ≡ ?rhs} => let
          val f = lambda x rhs |> Thm.cterm-of ctxt
          val x = Thm.cterm-of ctxt x
          val eval-thm = Drule.infer-instantiate' ctxt
            [SOME f, SOME x] @{thm add-eval}
          val thm = @{thm transitive} OF [thm,eval-thm]
        in thm end
      | - => raise THM (mk-mcomb internal: Expected lhs==rhs,~1,[thm])

    val thm = Drule.infer-instantiate' ctxt [SOME lhs] @{thm start-eval}
    val thm = fold add-EVAL (rev vars) thm
    val thm = singleton (Proof-Context.export ctxt orig-ctxt) thm
  in
    thm
  end;

  (*
    Generate arity-theorem: t = λx1...xn. SP t$x1$...$xn
  *)
  fun mk-arity ctxt t n = let
    val t = Thm.cterm-of ctxt t
    val thm = Drule.infer-instantiate' ctxt [SOME t] @{thm init-mk-arity}
    val add-mk-arity = Conv.fconv-rule (
      Refine-Util.ftop-conv (K (Conv.rewr-conv @{thm add-mk-arity})) ctxt)
    val thm = funpow n add-mk-arity thm
    val thm = Conv.fconv-rule (
      Refine-Util.ftop-conv (K (Conv.rewr-conv @{thm finish-mk-arity})) ctxt)
thm
  in
    thm
  end;

  datatype opkind = PURE | COMB


  fun analyze-decl c tyt = let
    fun add-tcons-of (Type (name,args)) l = fold add-tcons-of args (name::l)
      | add-tcons-of - l = l
```

```
  fun all-tcons-of P T = forall P (add-tcons-of T [])

  val T = Logic.dest-type tyt
  val (argsT,resT) = strip-type T

  val - = forall (all-tcons-of (fn tn => tn <> @{type-name nres})) argsT
    orelse raise TYPE (
      Arguments contain nres-type
    ^ (currently not supported by this attribute),
    argsT,[c,tyt])

  val kind = case resT of
    Type (@{type-name nres},-) => COMB
  | T => let
      val - = all-tcons-of (fn tn => tn <> @{type-name nres}) T
        orelse raise TYPE (
          Result contains inner nres-type,
        argsT,[c,tyt])
    in
      PURE
    end

in (kind,(argsT,resT)) end


fun analyze-itype-thm thm =
  case Thm.prop-of thm of
    @{mpat (typs) Trueprop (intf-type ?c (-::?'v-T itself))} => let
      val tyt = Logic.mk-type T
      val (kind,(argsT,resT)) = analyze-decl c tyt
    in (c,kind,(argsT,resT)) end
  | - => raise THM(Invalid itype-theorem,~1,[thm])


(*fun register-combinator itype-thm context = let
  val ctxt = Context.proof-of context
  val (t,kind,(argsT,-)) = analyze-itype-thm itype-thm
  val n = length argsT
in
  case kind of
    PURE => context
   |> Named-Theorems-Rev.add-thm @{named-theorems-rev id-rules} itype-thm
  | COMB => let
      val arity-thm = mk-arity ctxt t n
      (*val skel-thm = mk-skel ctxt t n*)
      val mcomb-thm = mk-mcomb ctxt t n
    in
      context
   |> Named-Theorems-Rev.add-thm @{named-theorems-rev id-rules} itype-thm
```

```
      |> Named-Theorems-Rev.add-thm @{named-theorems-rev sepref-monadify-arity}
arity-thm
      |> Named-Theorems-Rev.add-thm @{named-theorems-rev sepref-monadify-comb}
mcomb-thm
        (*|> Named-Theorems-Rev.add-thm @{named-theorems-rev sepref-la-skel}
skel-thm*)
      end
  end
  *)

  fun generate-basename ctxt t = let
    fun fail () = raise TERM (Basename generation heuristics failed. Specify a
basename.,[t])
    fun gb (Const (n,-)) =
     (* TODO: There should be a cleaner way than handling this on string level!*)
      n |> space-explode . |> List.last
     | gb (@{mpat PR-CONST ?t}) = gb t
     | gb (t as (-$-)) = let
         val h = head-of t
         val - = is-Const h orelse is-Free h orelse fail ()
       in
         gb h
       end
     | gb (Free (n,-)) =
         if Variable.is-fixed ctxt n then n
         else fail ()
     | gb - = fail ()
  in
    gb t
  end

  fun map-type-raw ctxt rls T = let
    val thy = Proof-Context.theory-of ctxt

    fun rewr-this (lhs,rhs) T = let
      val env = Sign.typ-match thy (lhs,T) Vartab.empty
    in
      Envir.norm-type env rhs
    end

    fun map-Targs f (Type (name,args)) = Type (name,map f args)
      | map-Targs - T = T

    fun
      rewr-thiss (r::rls) T =
       (SOME (rewr-this r T) handle Type.TYPE-MATCH => rewr-thiss rls T)
    | rewr-thiss [] - = NONE

    fun
```

```
    map-type-aux T =
      let
        val T = map-Targs map-type-aux T
      in
        case rewr-thiss rls T of
          SOME T => map-type-aux T
        | NONE => T
      end
in
  map-type-aux T
end

fun get-nt-rule thm = case Thm.prop-of thm of
  @{mpat (typs) Trueprop (map-type-eq (-::?'v-L itself) (-::?'v-R itself))} =>
  let
    val Lvars = Term.add-tvar-namesT L []
    val Rvars = Term.add-tvar-namesT R []

    val - = subset (=) (Rvars, Lvars) orelse (
      let
        val frees = subtract (=) Lvars Rvars
          |> map (Term.string-of-vname)
          |> Pretty.str-list [ ]
          |> Pretty.string-of
      in
        raise THM (Free variables on RHS: ^frees,~1,[thm])
      end)

  in
    (L,R)
  end

| - => raise THM(No map-type-eq theorem,~1,[thm])

fun map-type ctxt T = let
  val rls =
      Named-Theorems-Rev.get ctxt @{named-theorems-rev map-type-eqs}
   |> map get-nt-rule
in map-type-raw ctxt rls T end

fun read-term-type ts tys lthy = case tys of
  SOME ty => let
    val ty = Syntax.read-typ lthy ty
    val ctxt = Variable.declare-typ ty lthy
    val t = Syntax.read-term ctxt ts
    val ctxt = Variable.declare-term t ctxt
  in
    ((t,ty),ctxt)
  end
```

```
  | NONE => let
      val t = Syntax.read-term lthy ts
      val ctxt = Variable.declare-term t lthy

      val tyt = fastype-of t |> map-type ctxt |> Logic.mk-type

      val tyt = tyt |> singleton (Variable.export-terms ctxt lthy)
      val (tyt,ctxt) = yield-singleton (Variable.import-terms true) tyt ctxt
      val ty = Logic.dest-type tyt
    in
      ((t,ty),ctxt)
    end

  fun check-type-intf ctxt Tc Ti = let
    fun type2term (TFree (name,-)) = Var ((F^name,0),HOLogic.unitT)
      | type2term (TVar ((name,i),-)) = Var ((V^name,i),HOLogic.unitT)
      | type2term (Type (@{type-name fun},[T1,T2])) =
          Free (F,HOLogic.unitT --> HOLogic.unitT --> HOLogic.unitT)
            $type2term T1$type2term T2
      | type2term (Type (name,argsT)) = let
          val args = map type2term argsT
          val n = length args
          val T = replicate n HOLogic.unitT --->  HOLogic.unitT
          val v = Var ((T^name,0),T)
        in list-comb (v, args) end

    val c = type2term Tc
    val i = type2term Ti
    val thy = Proof-Context.theory-of ctxt
  in
    Pattern.matches thy (i,c)
  end

  (* Import all terms into context, with disjoint free variables *)
  fun import-terms-disj ts ctxt = let
    fun exp ctxt t = let
      val new-ctxt = Variable.declare-term t ctxt
      val t = singleton (Variable.export-terms new-ctxt ctxt) t
    in t end

    val ts = map (exp ctxt) ts

    fun cons-fst f a (l,b) = let val (a,b) = f a b in (a::l,b) end

   val (ts,ctxt) = fold-rev (cons-fst (yield-singleton (Variable.import-terms true)))
ts ([],ctxt)
  in
    (ts,ctxt)
  end
```

```
type reg-thms = {
  itype-thm: thm,
  arity-thm: thm option,
  mcomb-thm: thm option
}

fun cr-reg-thms t ty ctxt = let
  val orig-ctxt = ctxt
  val tyt = Logic.mk-type ty
  val ([t,tyt],ctxt) = import-terms-disj [t,tyt] ctxt

  val (kind,(argsT,-)) = analyze-decl t tyt
  val n = length argsT

  val - = Sepref-Intf-Types.check-intf-type-err ctxt ty (fastype-of t)

  val - = is-valid-abs-op ctxt t
      orelse raise TERM(Malformed abstract operation. Use PR-CONST for
complex terms.,[t])

  val itype-thm = mk-itype ctxt t tyt
    |> singleton (Variable.export ctxt orig-ctxt)
in
  case kind of
    PURE => {itype-thm = itype-thm, arity-thm = NONE, mcomb-thm =
NONE}
  | COMB => let
      val arity-thm = mk-arity ctxt t n
        |> singleton (Variable.export ctxt orig-ctxt)
      val mcomb-thm = mk-mcomb ctxt t n
        |> singleton (Variable.export ctxt orig-ctxt)
    in
      {itype-thm = itype-thm, arity-thm = SOME arity-thm, mcomb-thm =
SOME mcomb-thm}
    end
end

fun gen-pr-const-pat ctxt t =
  if is-valid-abs-op ctxt t then (NONE,t)
  else
    let
      val ct = Thm.cterm-of ctxt t
        val thm = Drule.infer-instantiate' ctxt [SOME ct] @{thm UNPRO-
TECT-def[symmetric]}
        |> Conv.fconv-rule (Conv.arg1-conv (Id-Op.protect-conv ctxt))
    in
      (SOME thm,@{mk-term PR-CONST ?t})
    end
```

```
  fun sepref-register-single basename t ty lthy = let
    fun mk-qualified basename q = Binding.qualify true basename (Binding.name
q);
    fun
      do-note - - NONE = I
    | do-note q attrs (SOME thm) =
        Local-Theory.note ((mk-qualified basename q,attrs),[thm]) #> snd

    val (pat-thm,t) = gen-pr-const-pat lthy t

    val {itype-thm, arity-thm, mcomb-thm} = cr-reg-thms t ty lthy

    val lthy = lthy
        |> do-note pat @{attributes [def-pat-rules]} pat-thm
        |> do-note itype @{attributes [id-rules]} (SOME itype-thm)
        |> do-note arity @{attributes [sepref-monadify-arity]} arity-thm
        |> do-note mcomb @{attributes [sepref-monadify-comb]} mcomb-thm

  in
    (((arity-thm,mcomb-thm),itype-thm),lthy)
  end

  fun sepref-register-single-cmd ((basename,ts),tys) lthy = let
    val t = Syntax.read-term lthy ts
    val ty = map-option (Syntax.read-typ lthy) tys

    val ty = case ty of SOME ty => ty | NONE => fastype-of t |> map-type lthy

    val basename = case basename of
      NONE => generate-basename lthy t
    | SOME n => n

    val ((-,itype-thm),lthy) = sepref-register-single basename t ty lthy
      val - = Document-Output.pretty-thm lthy itype-thm |> Pretty.string-of |>
writeln

  in
    lthy
  end

  val sepref-register-cmd = fold sepref-register-single-cmd

  val sepref-register-parser = Scan.repeat1 (
      Scan.option (Parse.name --| @{keyword :})
      -- Parse.term
      -- Scan.option (@{keyword ::} |-- Parse.typ)
    )
```

```
  val - =
    Outer-Syntax.local-theory
      @{command-keyword sepref-register}
      Register operation for sepref
      ( sepref-register-parser
        >> sepref-register-cmd);

  val sepref-register-adhoc-parser = Scan.repeat1 (
    Args.term -- Scan.option (Scan.lift (Args.$$$ ::) |-- Args.typ)
  )

  fun sepref-register-adhoc-single (t,ty) context = let
    val ctxt = Context.proof-of context

      (* TODO: Map-type probably not clean, as it draws info from (current)
context,
       which may have changed if registered elsewhere ...
    *)
    val ty = case ty of SOME ty => ty | NONE => fastype-of t |> map-type ctxt

    val (pat-thm,t) = gen-pr-const-pat ctxt t

    val {itype-thm, arity-thm, mcomb-thm} = cr-reg-thms t ty ctxt

    fun app - NONE = I
      | app attr (SOME thm) = Thm.apply-attribute attr thm #> snd

  in
    context
      |> app (Named-Theorems-Rev.add @{named-theorems-rev def-pat-rules})
pat-thm
    |> app (Named-Theorems-Rev.add @{named-theorems-rev id-rules}) (SOME
itype-thm)
  |> app (Named-Theorems-Rev.add @{named-theorems-rev sepref-monadify-arity})
arity-thm
  |> app (Named-Theorems-Rev.add @{named-theorems-rev sepref-monadify-comb})
mcomb-thm
  end

  val sepref-register-adhoc = fold sepref-register-adhoc-single

 fun sepref-register-adhoc-attr ttys = Thm.declaration-attribute (K (sepref-register-adhoc
ttys))

 val sepref-register-adhoc-attr-decl = sepref-register-adhoc-parser >> sepref-register-adhoc-attr

end
›
```

end

attribute-setup *sepref-register-adhoc = Sepref-Combinator-Setup.sepref-register-adhoc-attr-decl*
‹*Register operations in ad−hoc manner. Improper if this gets exported*!›

1.6.4 Obsolete Manual Setup Rules

lemma
mk-mcomb1: $\bigwedge c.\ c\$x1 \equiv (\gg\!=)\$(EVAL\$x1)\$(\lambda_2 x1.\ SP\ (c\$x1))$
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*: $\bigwedge c.\ c\$x1\$x2\$x3 \equiv (\gg\!=)\$(EVAL\$x1)\$(\lambda_2 x1.\ (\gg\!=)\$(EVAL\$x2)\$(\lambda_2 x2.\ (\gg\!=)\$(EVAL\$x3)\$(\lambda_2 x3.\ SP\ (c\$x1\$x2\$x3))))$
by *auto*

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
unfolding *vassn-tag-def* **..**

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
unfolding *vassn-tag-def rdomp-def*[*abs-def*]
by (*auto simp*: *mod-star-conv hn-ctxt-def*)

lemma *entails-preI*:
assumes *vassn-tag* $A \Longrightarrow A \Longrightarrow_A B$
shows $A \Longrightarrow_A B$
using *assms*
by (*auto simp*: *entails-def vassn-tag-def*)

lemma *invalid-assn-const*:
invalid-assn $(\lambda\text{- -}.\ P)\ x\ y = \uparrow$(*vassn-tag* P) $*$ *true*
by (*simp-all add*: *invalid-assn-def vassn-tag-def*)

lemma *vassn-tag-simps*[*simp*]:
vassn-tag emp
vassn-tag true
by (*sep-auto simp*: *vassn-tag-def mod-emp*)+

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

lemma *is-GEN-ALGO*: *GEN-ALGO* $f\ \Phi \Longrightarrow$ *GEN-ALGO* $f\ \Phi$ **.**

Tag for side-condition solver to discharge by assumption

definition *RPREM* :: *bool* $\Rightarrow$ *bool* **where** [*simp*]: *RPREM* $P = P$
lemma *RPREMI*: $P \Longrightarrow$ *RPREM* P **by** *simp*

lemma *trans-frame-rule*:
assumes *RECOVER-PURE* $\Gamma\ \Gamma'$

assumes *vassn-tag* $\Gamma' \Longrightarrow$ *hn-refine* Γ' *c* Γ'' *R a*
shows *hn-refine* $(F{*}\Gamma)$ *c* $(F{*}\Gamma'')$ *R a*
apply (*rule hn-refine-frame*[*OF - entt-refl*])
applyF (*rule hn-refine-cons-pre*)
focus using *assms*(*1*) **unfolding** *RECOVER-PURE-def* **apply** *assumption*
solved

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

lemma *recover-pure-cons*: — Used for debugging
assumes *RECOVER-PURE* Γ Γ'
assumes *hn-refine* Γ' *c* Γ'' *R a*
shows *hn-refine* (Γ) *c* (Γ'') *R a*
using *trans-frame-rule*[**where** *F*=*emp*, *OF assms*] **by** *simp*

— Tag to align structure of refinement assertions for consequence rule
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*)
by *simp*
lemma *CPR-tag-ctxtI*: *CPR-TAG* (*hn-ctxt R x xi*) (*hn-ctxt R′ x xi*) **by** *simp*
lemma *CPR-tag-fallbackI*: *CPR-TAG P Q* **by** *simp*

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

lemma *cons-pre-rule*: — Consequence rule to be applied if no direct operation rule matches
assumes *CPR-TAG P P′*
assumes $P \Longrightarrow_t P'$
assumes *hn-refine P′ c Q R m*
shows *hn-refine P c Q R m*
using *assms*(*2*−) **by** (*rule hn-refine-cons-pre*)

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

ML ‹

structure Sepref-Translate = *struct*

val cfg-debug =
Attrib.setup-config-bool @{*binding sepref-debug-translate*} (*K false*)

```
val dbg-msg-tac = Sepref-Debugging.dbg-msg-tac cfg-debug

fun gen-msg-analyze t ctxt = let
  val t = Logic.strip-assums-concl t
in
  case t of
    @{mpat Trueprop ?t} => (case t of
          @{mpat - ∨_A - ⟹_t -} => t-merge
        | @{mpat - ⟹_t -} => t-frame
        | @{mpat INDEP -} => t-indep
        | @{mpat CONSTRAINT - -} => t-constraint
        | @{mpat mono-Heap -} => t-mono
        | @{mpat PREFER-tag -} => t-prefer
        | @{mpat DEFER-tag -} => t-defer
        | @{mpat RPREM -} => t-rprem
      | @{mpat hn-refine - - - - ?a} => Pretty.block [Pretty.str t-hnr: ,Pretty.brk
1, Syntax.pretty-term ctxt a] |> Pretty.string-of
        | - => Unknown goal type
      )
  | - => Non−Trueprop goal
end

fun msg-analyze msg = Sepref-Debugging.msg-from-subgoal msg gen-msg-analyze

fun check-side-conds thm = let
  open Sepref-Basic
  (* Check that term is no binary operator on assertions *)
  fun is-atomic (Const (-,@{typ assn⇒assn⇒assn})$-$-) = false
    | is-atomic - = true

   val is-atomic-star-list = (Expected atoms separated by star,forall is-atomic o
strip-star)

  val is-trueprop = (Expected Trueprop conclusion,can HOLogic.dest-Trueprop)

  fun assert t' (msg,p) t = if p t then () else raise TERM(msg,[t',t])

  fun chk-prem t = let
    val assert = assert t

    fun chk @{mpat ?l ∨_A ?r ⟹_t ?m} = (
          assert is-atomic-star-list l;
          assert is-atomic-star-list r;
          assert is-atomic-star-list m
        )
      | chk (t as @{mpat - ⟹_A -}) = raise TERM(Invalid frame side condition
(old−style ent),[t])
      | chk @{mpat ?l ⟹_t ?r} = (
```

```
        assert is-atomic-star-list l;
        assert is-atomic-star-list r
      )
    | chk - = ()

    val t = Logic.strip-assums-concl t
  in
    assert is-trueprop t;
    chk (HOLogic.dest-Trueprop t)
  end

in
  map chk-prem (Thm.prems-of thm)
end

structure sepref-comb-rules = Named-Sorted-Thms (
  val name = @{binding sepref-comb-rules}
  val description = Sepref: Combinator rules
  val sort = K I
  fun transform - thm = let
    val - = check-side-conds thm
  in
    [thm]
  end
)

structure sepref-fr-rules = Named-Sorted-Thms (
  val name = @{binding sepref-fr-rules}
  val description = Sepref: Frame-based rules
  val sort = K I
  fun transform context thm = let
    val ctxt = Context.proof-of context
    val thm = Sepref-Rules.ensure-hnr ctxt thm
      |> Conv.fconv-rule (Sepref-Frame.align-rl-conv ctxt)

    val - = check-side-conds thm
    val - = case try (Sepref-Rules.analyze-hnr ctxt) thm of
        NONE =>
          (Pretty.block [
            Pretty.str hnr-analysis failed,
            Pretty.str :,
            Pretty.brk 1,
            Thm.pretty-thm ctxt thm])
          |> Pretty.string-of |> error
      | SOME ana => let
          val - = Sepref-Combinator-Setup.is-valid-abs-op ctxt (fst (#ahead ana))
            orelse Pretty.block [
              Pretty.str Invalid abstract head:,
              Pretty.brk 1,
```

```
            Pretty.enclose ( ) [Syntax.pretty-term ctxt (fst (#ahead ana))],
            Pretty.brk 1,
            Pretty.str in thm,
            Pretty.brk 1,
            Thm.pretty-thm ctxt thm
          ]
        |> Pretty.string-of |> error
      in () end
  in
    [thm]
  end
)

(***** Side Condition Solving *)
local
  open Sepref-Basic
in

  fun side-unfold-tac ctxt = let
    (*val ctxt = put-simpset HOL-basic-ss ctxt
      addsimps sepref-prep-side-simps.get ctxt*)
  in
    CONVERSION (Id-Op.unprotect-conv ctxt)
   THEN' SELECT-GOAL (Local-Defs.unfold0-tac ctxt @{thms bind-ref-tag-def})
    (*THEN' asm-full-simp-tac ctxt*)
  end

  fun side-fallback-tac ctxt = side-unfold-tac ctxt THEN' TRADE (SELECT-GOAL
o auto-tac) ctxt

  val side-frame-tac = Sepref-Frame.frame-tac side-fallback-tac
  val side-merge-tac = Sepref-Frame.merge-tac side-fallback-tac
  fun side-constraint-tac ctxt = Sepref-Constraints.constraint-tac ctxt
  fun side-mono-tac ctxt = side-unfold-tac ctxt THEN' TRADE Pf-Mono-Prover.mono-tac
ctxt

  fun side-gen-algo-tac ctxt =
    side-unfold-tac ctxt
   THEN' resolve-tac ctxt (Named-Theorems-Rev.get ctxt @{named-theorems-rev
sepref-gen-algo-rules})

  fun side-pref-def-tac ctxt =
    side-unfold-tac ctxt THEN'
    TRADE (fn ctxt =>
      resolve-tac ctxt @{thms PREFER-tagI DEFER-tagI}
        THEN' (Sepref-Debugging.warning-tac' Obsolete PREFER/DEFER side
condition ctxt THEN' Tagged-Solver.solve-tac ctxt)
    ) ctxt
```

```
    fun side-rprem-tac ctxt =
      resolve-tac ctxt @{thms RPREMI} THEN' Refine-Util.rprems-tac ctxt
      THEN' (K (smash-new-rule ctxt))

    (*
      Solve side condition, or invoke hnr-tac on hn-refine goal.

      In debug mode, side-condition solvers are allowed to not completely solve
      the side condition, but must change the goal.
    *)
    fun side-cond-dispatch-tac dbg hnr-tac ctxt = let
      fun MK tac = if dbg then CHANGED o tac ctxt else SOLVED' (tac ctxt)

      val t-merge = MK side-merge-tac
      val t-frame = MK side-frame-tac
      val t-indep = MK Indep-Vars.indep-tac
      val t-constraint = MK side-constraint-tac
      val t-mono = MK side-mono-tac
      val t-pref-def = MK side-pref-def-tac
      val t-rprem = MK side-rprem-tac
      val t-gen-algo = side-gen-algo-tac ctxt
      val t-fallback = MK side-fallback-tac
    in
      WITH-concl
        (fn @{mpat Trueprop ?t} => (case t of
              @{mpat - ∨_A - ⟹_t -} => t-merge
            | @{mpat - ⟹_t -} => t-frame
            | @{mpat - ⟹_A -} => Sepref-Debugging.warning-tac' Old-style frame
side condition ctxt THEN' (K no-tac)
            | @{mpat INDEP -} => t-indep      (* TODO: Get rid of this!? *)
            | @{mpat CONSTRAINT - -} => t-constraint
            | @{mpat mono-Heap -} => t-mono
            | @{mpat PREFER-tag -} => t-pref-def
            | @{mpat DEFER-tag -} => t-pref-def
            | @{mpat RPREM -} => t-rprem
            | @{mpat GEN-ALGO - -} => t-gen-algo
            | @{mpat hn-refine - - - - -} => hnr-tac
            | - => t-fallback
          )
        | - => K no-tac
      )
    end

  end

  (***** Main Translation Tactic *)
  local
    open Sepref-Basic STactical
```

```
  (* ATTENTION: Beware of evaluation order, as some initialization operations
on
     context are expensive, and should not be repeated during proof search! *)
 in


  (* Translate combinator, yields new translation goals and side conditions
    which must be processed in order. *)
  fun trans-comb-tac ctxt = let
    val comb-rl-net = sepref-comb-rules.get ctxt
      |> Bires.build-net

  in
    DETERM o (
      Bires.resolve-from-net-tac ctxt comb-rl-net
      ORELSE' (
        Sepref-Frame.norm-goal-pre-tac ctxt
        THEN' Bires.resolve-from-net-tac ctxt comb-rl-net
      )
    )
  end

  (* Translate operator. Only succeeds if it finds an operator rule such that
    all resulting side conditions can be solved. Takes the first such rule.

    In debug mode, it returns a sequence of the unsolved side conditions of
    each applicable rule.
  *)
  fun gen-trans-op-tac dbg ctxt = let
    val fr-rl-net = sepref-fr-rules.get ctxt |> Bires.build-net
    val fr-rl-tac =
      Bires.resolve-from-net-tac ctxt fr-rl-net (* Try direct match *)
      ORELSE' (
        Sepref-Frame.norm-goal-pre-tac ctxt (* Normalize precondition *)
        THEN' (
          Bires.resolve-from-net-tac ctxt fr-rl-net
          ORELSE' (
              resolve-tac ctxt @{thms cons-pre-rule} (* Finally, generate a frame
condition *)
            THEN-ALL-NEW-LIST [
              SOLVED' (REPEAT-ALL-NEW-FWD (DETERM o resolve-tac ctxt
@{thms CPR-TAG-rules})),
                K all-tac,  (* Frame remains unchanged as first goal, even if fr-rl
creates side-conditions *)
              Bires.resolve-from-net-tac ctxt fr-rl-net
            ]
          )
        )
```

```
      )

      val side-tac = REPEAT-ALL-NEW-FWD (side-cond-dispatch-tac false (K
no-tac) ctxt)

     val fr-tac =
      if dbg then (* Present all possibilities with (partially resolved) side conditions
*)
        fr-rl-tac THEN-ALL-NEW-FWD (TRY o side-tac)
       else (* Choose first rule that solves all side conditions *)
           DETERM o SOLVED' (fr-rl-tac THEN-ALL-NEW-FWD (SOLVED'
side-tac))

   in
     PHASES' [
       (Align goal,Sepref-Frame.align-goal-tac, 0),
       (Frame rule,fn ctxt => resolve-tac ctxt @{thms trans-frame-rule}, 1),
       (* RECOVER-PURE goal *)
       (Recover pure,Sepref-Frame.recover-pure-tac, ~1),
       (* hn-refine goal with stripped precondition *)
       (Apply rule,K fr-tac,~1)
     ] (flag-phases-ctrl dbg) ctxt
   end

   (* Translate combinator, operator, or side condition. *)
   fun gen-trans-step-tac dbg ctxt = side-cond-dispatch-tac dbg
     (trans-comb-tac ctxt ORELSE' gen-trans-op-tac dbg ctxt)
     ctxt

   val trans-step-tac = gen-trans-step-tac false
   val trans-step-keep-tac = gen-trans-step-tac true

   fun gen-trans-tac dbg ctxt =
     PHASES' [
       (Translation steps,REPEAT-DETERM' o trans-step-tac,~1),
     (Constraint solving,fn ctxt => fn - => Sepref-Constraints.process-constraint-slot
ctxt, 0)
     ] (flag-phases-ctrl dbg) ctxt

   val trans-tac = gen-trans-tac false
   val trans-keep-tac = gen-trans-tac true


 end


 val setup = I
   #> sepref-fr-rules.setup
   #> sepref-comb-rules.setup
```

end

›

setup *Sepref-Translate.setup*

Basic Setup

lemma *hn-pass*[*sepref-fr-rules*]:
  shows *hn-refine* (*hn-ctxt P x x'*) (*return x'*) (*hn-invalid P x x'*) *P* (*PASS*$*x*)
  apply *rule* **apply** (*sep-auto simp*: *hn-ctxt-def invalidate-clone'*)
  done

lemma *hn-bind*[*sepref-comb-rules*]:
  assumes *D1*: *hn-refine* 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*))
  using *assms*
  unfolding *APP-def PROTECT2-def bind-ref-tag-def*
  by (*rule hnr-bind*)

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*)
  unfolding *APP-def PROTECT2-def*
  apply (*rule hn-refine-cons-pre*[*OF FR*])
  apply (*rule hnr-RECT*)

  apply (*rule hn-refine-cons-post*[*OF - FR'*])
  apply (*rule S*[*unfolded RCALL-def APP-def*])

```
  apply assumption
  apply fact+
  done

lemma hn-RCALL[sepref-comb-rules]:
  assumes RPREM (hn-refine P' c Q' R (RCALL $ a $ b))
    and P ⟹ₜ F * P'
  shows hn-refine P c (F * Q') R (RCALL $ a $ b)
  using assms hn-refine-frame[where m=RCALL$a$b]
  by simp


definition monadic-WHILEIT I b f s ≡ do {
  RECT (λD 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
  }
  unfolding heap-WHILET-def
  apply (subst heap.mono-body-fixp)
  apply pf-mono
  apply simp
  done
```

lemma *WHILEIT-to-monadic*: *WHILEIT I b f s = monadic-WHILEIT I* (λ*s*. *RETURN* (*b s*)) *f s*
unfolding *WHILEIT-def monadic-WHILEIT-def*
unfolding *WHILEI-body-def bind-ASSERT-eq-if*
by (*simp cong*: *if-cong*)

lemma *WHILEIT-pat*[*def-pat-rules*]:
WHILEIT$*I* ≡ *UNPROTECT* (*WHILEIT I*)
WHILET ≡ *PR-CONST* (*WHILEIT* (λ-. *True*))
by (*simp-all add*: *WHILET-def*)

lemma *id-WHILEIT*[*id-rules*]:
PR-CONST (*WHILEIT I*) ::$_i$ *TYPE*((′*a* ⇒ *bool*) ⇒ (′*a* ⇒ ′*a nres*) ⇒ ′*a* ⇒ ′*a nres*)
by *simp*

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

PR-CONST (*WHILEIT I*) ≡ λ$_2$*b f s*. *SP* (*PR-CONST* (*WHILEIT I*))$(λ$_2$*s*. *b*$*s*)$(λ$_2$*s*. *f*$*s*)$*s*
by (*simp-all add*: *WHILET-def*)

lemma *WHILEIT-comb*[*sepref-monadify-comb*]:
PR-CONST (*WHILEIT I*)$(λ$_2$*x*. *b x*)$*f*$*s* ≡
Refine-Basic.bind$(*EVAL*$*s*)$(λ$_2$*s*.
SP (*PR-CONST* (*monadic-WHILEIT I*))$(λ$_2$*x*. (*EVAL*$(*b x*)))$*f*$*s*
)
by (*simp-all add*: *WHILEIT-to-monadic*)

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*)
(Γ*b s*′ *s*)
(*pure bool-rel*)
(*b*′ *s*′)
assumes *b-fr*: ⋀*s*′ *s*. Γ*b 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*)
(Γ*f s*′ *s*)
Rs
(*f*′ *s*′)
assumes *f-fr*: ⋀*s*′ *s*. Γ*f 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')
  unfolding monadic-WHILEIT-def heap-WHILET-def
  apply1 (rule hn-refine-cons-pre[OF FR])
  apply weaken-hnr-post
  focus (rule hn-refine-cons-pre[OF - hnr-RECT])
    applyS (subst mult-ac(2)[of 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])
            apply (sep-auto simp: hn-ctxt-def pure-def intro!: enttI; fail)
            apply assumption
          solved

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

          apply (sep-auto intro!: enttI; fail)
        solved
        applyF (sep-auto,rule hn-refine-frame)
          applyS (rule hnr-RETURN-pass)

          apply (rule enttI)
          apply (fr-rot-rhs 1)
          apply (fr-rot 1, rule fr-refl)
          apply (rule fr-refl)
          apply solve-entails
        solved

        apply (rule entt-refl)
      solved

      apply (rule enttI)
      applyF (rule ent-disjE)
        apply1 (sep-auto simp: hn-ctxt-def pure-def)
        apply1 (rule ent-true-drop)
        apply1 (rule ent-true-drop)
        applyS (rule ent-refl)
```

applyS (*sep-auto simp*: *hn-ctxt-def pure-def*)
solved
solved
apply *pf-mono*
solved
done

lemma *hn-monadic-WHILE-lin*[*sepref-comb-rules*]:
assumes *INDEP Rs*
assumes *FR*: $P \Longrightarrow_t \Gamma * \textit{hn-ctxt Rs } s' \ s$
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
($\textit{PR-CONST } (\textit{monadic-WHILEIT } I)\$(\lambda_2 s'.\ b' \ s')\$(\lambda_2 s'.\ f' \ s')\$(s')$)
using *assms*(*2*−)
unfolding *APP-def PROTECT2-def CONSTRAINT-def PR-CONST-def*
by (*rule hn-monadic-WHILE-aux*)

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*]: $\bigwedge s' \ s.\ [\![\ (s',s) \in R;\ I \ s;\ I' \ s' \ ]\!] \Longrightarrow b' \ s' \leq \Downarrow \textit{bool-rel } (b \ s)$
assumes [*refine*]: $\bigwedge s' \ s.\ [\![\ (s',s) \in R;\ I \ s;\ I' \ s';\ \textit{nofail } (b \ s);\ \textit{inres } (b \ s) \ \textit{True} \ ]\!] \Longrightarrow f' \ s' \leq \Downarrow R \ (f \ s)$
shows *monadic-WHILEIT I' b' f' s'* $\leq \Downarrow R$ (*monadic-WHILEIT I b f s*)
unfolding *monadic-WHILEIT-def*
by (*refine-rcg bind-refine'*; *assumption?*; *auto*)

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* $I'\ b'\ f'\ s' \leq \Downarrow R\ (WHILEIT\ I\ b\ f\ s)$
unfolding *WHILEIT-to-monadic*
by (*refine-vcg*; *assumption?*; *auto*)

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* $(\lambda\text{-}.\ True)\ b'\ f'\ s' \leq \Downarrow R\ (WHILET\ b\ f\ s)$
unfolding *WHILET-def*
by (*refine-vcg*; *assumption?*)

lemma *monadic-WHILEIT-pat*[*def-pat-rules*]:
monadic-WHILEIT$I \equiv UNPROTECT\ (monadic\text{-}WHILEIT\ I)$
by *auto*

lemma *id-monadic-WHILEIT*[*id-rules*]:
$PR\text{-}CONST\ (monadic\text{-}WHILEIT\ I) ::_i TYPE(('a \Rightarrow bool\ nres) \Rightarrow ('a \Rightarrow 'a\ nres) \Rightarrow 'a \Rightarrow 'a\ nres)$
by *simp*

lemma *monadic-WHILEIT-arities*[*sepref-monadify-arity*]:
$PR\text{-}CONST\ (monadic\text{-}WHILEIT\ I) \equiv \lambda_2 b\ f\ s.\ SP\ (PR\text{-}CONST\ (monadic\text{-}WHILEIT\ I))\$(\lambda_2 s.\ b\$s)\$(\lambda_2 s.\ f\$s)\$s$
by (*simp*)

lemma *monadic-WHILEIT-comb*[*sepref-monadify-comb*]:
$PR\text{-}CONST\ (monadic\text{-}WHILEIT\ I)\$b\$f\$s \equiv$
Refine-Basic.bind$(EVAL\$s)\$(\lambda_2 s.$
$SP\ (PR\text{-}CONST\ (monadic\text{-}WHILEIT\ I))\$b\$f\$s$
$)$
by (*simp*)

definition [*simp*]: *op-ASSERT-bind* $I\ m \equiv$ *Refine-Basic.bind* $(ASSERT\ I)\ (\lambda\text{-}.\ m)$
lemma *pat-ASSERT-bind*[*def-pat-rules*]:
Refine-Basic.bind$(ASSERT\$I)\$(\lambda_2\text{-}.\ m) \equiv UNPROTECT\ (op\text{-}ASSERT\text{-}bind\ I)\$m$
by *simp*

term *PR-CONST* (*op-ASSERT-bind* I)
lemma *id-op-ASSERT-bind*[*id-rules*]:
$PR\text{-}CONST\ (op\text{-}ASSERT\text{-}bind\ I) ::_i TYPE('a\ nres \Rightarrow 'a\ nres)$
by *simp*

lemma *arity-ASSERT-bind*[*sepref-monadify-arity*]:
PR-CONST (*op-ASSERT-bind I*) ≡ $\lambda_2 m$. *SP* (*PR-CONST* (*op-ASSERT-bind I*))$*m*
apply (*rule eq-reflection*)
by *auto*

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*)
using *assms*
apply (*cases I*)
apply *auto*
done

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*
by *simp*

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

lemma *arity-ASSUME-bind*[*sepref-monadify-arity*]:
PR-CONST (*op-ASSUME-bind I*) ≡ $\lambda_2 m$. *SP* (*PR-CONST* (*op-ASSUME-bind I*))$*m*
apply (*rule eq-reflection*)
by *auto*

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*)
apply (*rule hn-refine-preI*)
using *assms*
apply (*cases I*)
apply (*auto simp*: *vassn-tag-def*)
done

1.7.1 Import of Parametricity Theorems

lemma *pure-hn-refineI*:
assumes *Q* ⟶ (*c*,*a*)∈*R*
shows *hn-refine* (↑*Q*) (*return c*) (↑*Q*) (*pure R*) (*RETURN a*)
unfolding *hn-refine-def* **using** *assms*
by (*sep-auto simp*: *pure-def*)

lemma *pure-hn-refineI-no-asm*:
assumes $(c,a)\in R$
shows *hn-refine emp* (*return c*) *emp* (*pure R*) (*RETURN a*)
unfolding *hn-refine-def* **using** *assms*
by (*sep-auto simp*: *pure-def*)

lemma *import-param-0*:
$(P \Longrightarrow Q) \equiv$ *Trueprop* (*PROTECT P* $\longrightarrow$ *Q*)
apply (*rule*, *simp*+)+
done

lemma *import-param-1*:
$(P \Longrightarrow Q) \equiv$ *Trueprop* $(P \longrightarrow Q)$
$(P \longrightarrow Q \longrightarrow R) \longleftrightarrow (P \wedge Q \longrightarrow R)$
PROTECT $(P \wedge Q) \equiv$ *PROTECT P* $\wedge$ *PROTECT Q*
$(P \wedge Q) \wedge R \equiv P \wedge Q \wedge R$
$(a,c)\in Rel \wedge$ *PROTECT P* $\longleftrightarrow$ *PROTECT P* $\wedge$ $(a,c)\in Rel$
apply (*rule*, *simp*+)+
done

lemma *import-param-2*:
Trueprop (*PROTECT P* $\wedge$ *Q* $\longrightarrow$ *R*) $\equiv$ (*P* $\Longrightarrow$ *Q* $\longrightarrow$ *R*)
apply (*rule*, *simp*+)+
done

lemma *import-param-3*:
$\uparrow(P \wedge Q) = \uparrow P * \uparrow Q$
$\uparrow((c,a)\in R) =$ *hn-val R a c*
by (*simp-all add*: *hn-ctxt-def pure-def*)

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

lemma *to-import-frefD*:
assumes $(f,g)\in$*fref P R S*
shows $[\![$*PROTECT* (*P y*); $(x,y)\in R]\!] \Longrightarrow (f\ x,\ g\ y)\in S$
using *assms*
unfolding *fref-def*
by *auto*

lemma *add-PR-CONST*: $(c,a)\in R \Longrightarrow (c,$*PR-CONST a*$)\in R$ **by** *simp*

ML ‹
structure Sepref-Import-Param = struct

(∗ *TODO*: *Almost clone of Sepref-Rules.to-foparam*∗)
fun to-import-fo ctxt thm = let
val unf-thms = @{*thms*

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

  fun add-PR-CONST thm = case Thm.concl-of thm of
     @{mpat Trueprop ((-,-) ∈ fref - - -)} => thm (* TODO: Hack. Need clean
interfaces for fref and param rules. Also add PR-CONST to fref rules! *)
  | @{mpat Trueprop ((-,PR-CONST -) ∈ -)} => thm
  | @{mpat Trueprop ((-,?a) ∈ -)} => if is-Const a orelse is-Free a orelse is-Var a
then
      thm
    else
      thm RS @{thm add-PR-CONST}
  | - => thm


  fun import ctxt thm = let
    open Sepref-Basic
    val thm = thm
      |> Conv.fconv-rule Thm.eta-conversion
      |> add-PR-CONST
      |> Local-Defs.unfold0 ctxt @{thms import-param-0}
      |> Local-Defs.unfold0 ctxt @{thms imp-to-meta}
      |> to-import-fo ctxt
      |> Local-Defs.unfold0 ctxt @{thms import-param-1}
      |> Local-Defs.unfold0 ctxt @{thms import-param-2}

    val thm = case Thm.concl-of thm of
      @{mpat Trueprop (-⟶-)} => thm RS @{thm pure-hn-refineI}
    | - => thm RS @{thm pure-hn-refineI-no-asm}

    val thm = Local-Defs.unfold0 ctxt @{thms import-param-3} thm
     |> Conv.fconv-rule (hn-refine-concl-conv-a (K (Id-Op.protect-conv ctxt)) ctxt)

   val thm = Local-Defs.unfold0 ctxt (Named-Theorems-Rev.get ctxt @{named-theorems-rev
sepref-import-rewrite}) thm
    val thm = Sepref-Rules.add-pure-constraints-rule ctxt thm
  in
    thm
  end
```

```
  val import-attr = Scan.succeed (Thm.mixed-attribute (fn (context,thm) =>
    let
      val thm = import (Context.proof-of context) thm
      val context = Sepref-Translate.sepref-fr-rules.add-thm thm context
    in (context,thm) end
  ))

  val import-attr-rl = Scan.succeed (Thm.rule-attribute [] (fn context =>
   import (Context.proof-of context) #> Sepref-Rules.ensure-hfref (Context.proof-of
context)
  ))

  val setup = I
    #> Attrib.setup @{binding sepref-import-param} import-attr
        Sepref: Import parametricity rule
    #> Attrib.setup @{binding sepref-param} import-attr-rl
        Sepref: Transform parametricity rule to sepref rule
    #> Attrib.setup @{binding sepref-dbg-import-rl-only}
        (Scan.succeed (Thm.rule-attribute [] (import o Context.proof-of)))
        Sepref: Parametricity to hnr−rule, no conversion to hfref

end
›

setup Sepref-Import-Param.setup
```

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)
  unfolding import-rel1-def
  apply simp
  apply (simp add: pure-def)
  done

lemma import-rel2-pure-conv: import-rel2 R (pure A) (pure B) = pure (⟨A,B⟩R)
  unfolding import-rel2-def
  apply simp
  apply (simp add: pure-def)
  done

lemma precise-pure[constraint-rules]: single-valued R ⟹ precise (pure R)
  unfolding precise-def pure-def
  by (auto dest: single-valuedD)
```

lemma *precise-pure-iff-sv*: *precise* (*pure R*) ⟷ *single-valued R*
apply (*auto simp*: *precise-pure*)
using *preciseD*[**where** *R*=*pure R* **and** *F*=*emp* **and** *F'*=*emp*]
by (*sep-auto simp*: *mod-and-dist intro*: *single-valuedI*)

lemma *pure-precise-iff-sv*: ⟦*is-pure R*⟧
⟹ *precise R* ⟷ *single-valued* (*the-pure R*)
by (*auto simp*: *is-pure-conv precise-pure-iff-sv*)

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*
unfolding *DEF*
apply (*rule fun-cong*[*of - - x*])
apply (*rule heap.mono-body-fixp*)
apply *fact*
done

ML ‹
structure Sepref-Extraction = *struct*
val heap-extraction: *Refine-Automation.extraction* = {
pattern = *Logic.varify-global* @{*term heap.fixp-fun x*},
gen-thm = @{*thm heap-fixp-codegen*},
gen-tac = (*fn ctxt* =>
Pf-Mono-Prover.mono-tac ctxt
)
}

val setup = I
(#> Refine-Automation.add-extraction trivial triv-extraction*)*
#> Refine-Automation.add-extraction heap heap-extraction

end
›

setup *Sepref-Extraction.setup*

1.8.2 Synthesis setup for sepref-definition goals

consts *UNSPEC*::*'a*

abbreviation *hfunspec*
:: (*'a* ⇒ *'b* ⇒ *assn*) ⇒ (*'a* ⇒ *'b* ⇒ *assn*)×(*'a* ⇒ *'b* ⇒ *assn*)
(‹(-$^?$)› [*1000*] *999*)
where $R^?$ ≡ *hf-pres R UNSPEC*

definition *SYNTH* :: (*'a* ⇒ *'r nres*) ⇒ ((*'ai* ⇒*'ri Heap*) × (*'a* ⇒ *'r nres*)) *set* ⇒ *bool*
where *SYNTH f R* ≡ *True*

definition [*simp*]: *CP-UNCURRY - -* ≡ *True*
definition [*simp*]: *INTRO-KD - -* ≡ *True*
definition [*simp*]: *SPEC-RES-ASSN - -* ≡ *True*

lemma [*synth-rules*]: *CP-UNCURRY f g* **by** *simp*
lemma [*synth-rules*]: *CP-UNCURRY* (*uncurry0 f*) (*uncurry0 g*) **by** *simp*
lemma [*synth-rules*]: *CP-UNCURRY f g* ⟹ *CP-UNCURRY* (*uncurry f*) (*uncurry g*) **by** *simp*

lemma [*synth-rules*]: ⟦*INTRO-KD R1 R1'*; *INTRO-KD R2 R2'*⟧ ⟹ *INTRO-KD* (*R1*$*_a$*R2*) (*R1'*$*_a$*R2'*) **by** *simp*
lemma [*synth-rules*]: *INTRO-KD* ($R^?$) (*hf-pres R k*) **by** *simp*
lemma [*synth-rules*]: *INTRO-KD* (R^k) (R^k) **by** *simp*
lemma [*synth-rules*]: *INTRO-KD* (R^d) (R^d) **by** *simp*

lemma [*synth-rules*]: *SPEC-RES-ASSN R R* **by** *simp*
lemma [*synth-rules*]: *SPEC-RES-ASSN UNSPEC R* **by** *simp*

lemma *synth-hnrI*:
⟦*CP-UNCURRY fi f*; *INTRO-KD R R'*; *SPEC-RES-ASSN S S'*⟧ ⟹ *SYNTH-TERM* (*SYNTH f* ([*P*]$_a$ *R*→*S*)) ((*fi*,*SDUMMY*)∈*SDUMMY*,(*fi*,*f*)∈([*P*]$_a$ *R'*→*S'*))
by (*simp add*: *SYNTH-def*)

term *starts-with*

ML ‹
structure Sepref-Definition = struct

```
  fun make-hnr-goal t ctxt = let
    val ctxt = Variable.declare-term t ctxt
    val (pat,goal) = case Term-Synth.synth-term @{thms synth-hnrI} ctxt t of
      @{mpat (?pat,?goal)} => (pat,goal) | t => raise TERM(Synthesized term
does not match,[t])
   val pat = Thm.cterm-of ctxt pat |> Refine-Automation.prepare-cd-pattern ctxt
    val goal = HOLogic.mk-Trueprop goal
  in
    ((pat,goal),ctxt)
  end

 val cfg-prep-code = Attrib.setup-config-bool @{binding sepref-definition-prep-code}
(K true)

  local
    open Refine-Util
    val flags = parse-bool-config' prep-code cfg-prep-code
    val parse-flags = parse-paren-list' flags

  in
      val sd-parser = parse-flags -- Parse.binding -- Parse.opt-attribs --|
@{keyword is}
      -- Parse.term --| @{keyword ::} -- Parse.term
  end

  fun mk-synth-term ctxt t-raw r-raw = let
      val t = Syntax.parse-term ctxt t-raw
      val r = Syntax.parse-term ctxt r-raw
      val t = Const (@{const-name SYNTH},dummyT)$t$r
    in
      Syntax.check-term ctxt t
    end


  fun sd-cmd ((((flags,name),attribs),t-raw),r-raw) lthy = let
    local
      val ctxt = Refine-Util.apply-configs flags lthy
    in
      val flag-prep-code = Config.get ctxt cfg-prep-code
    end

    val t = mk-synth-term lthy t-raw r-raw

    val ((pat,goal),ctxt) = make-hnr-goal t lthy

    fun
      after-qed [[thm]] ctxt = let
        val thm = singleton (Variable.export ctxt lthy) thm
```

```
          val (-,lthy)
            = Local-Theory.note
          ((Refine-Automation.mk-qualified (Binding.name-of name) refine-raw,[]),[thm])

              lthy;

           val ((dthm,rthm),lthy) = Refine-Automation.define-concrete-fun NONE
name attribs [] thm [pat] lthy

          val lthy = lthy
            |> flag-prep-code ? Refine-Automation.extract-recursion-eqs
              [Sepref-Extraction.heap-extraction] (Binding.name-of name) dthm

          val - = Thm.pretty-thm lthy dthm |> Pretty.string-of |> writeln
          val - = Thm.pretty-thm lthy rthm |> Pretty.string-of |> writeln
        in
          lthy
        end
      | after-qed thmss - = raise THM (After-qed: Wrong thmss structure,~1,flat
thmss)

    in
      Proof.theorem NONE after-qed [[ (goal,[]) ]] ctxt
    end



  val - = Outer-Syntax.local-theory-to-proof @{command-keyword sepref-definition}
    Synthesis of imperative program
    (sd-parser >> sd-cmd)

  val st-parser = Parse.binding --| @{keyword is} -- Parse.term --| @{keyword
::} -- Parse.term

    fun st-cmd ((name,t-raw),r-raw) lthy = let
      val t = mk-synth-term lthy t-raw r-raw
      val ((-,goal),ctxt) = make-hnr-goal t lthy

      fun
        after-qed [[thm]] ctxt = let
          val thm = singleton (Variable.export ctxt lthy) thm

          val - = Thm.pretty-thm lthy thm |> Pretty.string-of |> tracing

          val (-,lthy)
            = Local-Theory.note
          ((Refine-Automation.mk-qualified (Binding.name-of name) refine-raw,[]),[thm])

              lthy;
```

```
        in
          lthy
        end
      | after-qed thmss - = raise THM (After−qed: Wrong thmss structure,~1,flat
thmss)

    in
      Proof.theorem NONE after-qed [[ (goal,[]) ]] ctxt
    end

    val - = Outer-Syntax.local-theory-to-proof @{command-keyword sepref-thm}
      Synthesis of imperative program: Only generate raw refinement theorem
      (st-parser >> st-cmd)

  end
›

end
```

1.9 Utilities for Interface Specifications and Implementations

```
theory Sepref-Intf-Util
imports Sepref-Rules Sepref-Translate Lib/Term-Synth Sepref-Combinator-Setup
  Lib/Concl-Pres-Clarification
keywords sepref-decl-op :: thy-goal
    and sepref-decl-impl :: thy-goal
begin
```

1.9.1 Relation Interface Binding

```
  definition INTF-OF-REL :: ('a×'b) set ⇒ 'c itself ⇒ bool
    where [simp]: INTF-OF-REL R I ≡ True

  lemma intf-of-relI: INTF-OF-REL (R::(-×'a) set) TYPE('a) by simp
  declare intf-of-relI[synth-rules] — Declare as fallback rule

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

    INTF-OF-REL R TYPE('a) ⟹ INTF-OF-REL (⟨R⟩option-rel) TYPE('a op-
tion)
    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*)
by *simp-all*

lemma *synth-intf-of-relI*: *INTF-OF-REL R I* ⟹ *SYNTH-TERM R I* **by** *simp*

1.9.2 Operations with Precondition

definition *mop* :: ('*a*⇒*bool*) ⇒ ('*a*⇒'*b nres*) ⇒ '*a* ⇒ '*b nres*
— Package operation with precondition
where [*simp*]: *mop P f* ≡ 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*

using *assms*
by (*auto*
simp: *mop-def fref-def pw-nres-rel-iff refine-pw-simps*
dest: *fun-relD*)

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

lemma *mop-spec-rl*: *P x* ⟹ *mop P f x* ≤ *f x* **by** *simp*

lemma *mop-spec-rl-from-def*:
assumes *f* ≡ *mop P g*
assumes *P x*
assumes *g x* ≤ *z*
shows *f x* ≤ *z*
using *assms mop-spec-rl* **by** *simp*

lemma *mop-leof-rl-from-def*:
assumes *f* ≡ *mop P g*
assumes *P x* ⟹ *g x* ≤$_n$ *z*
shows *f x* ≤$_n$ *z*
using *assms*
by (*simp add*: *pw-leof-iff refine-pw-simps*)

lemma *assert-true-bind-conv*: $ASSERT\ True \gg m = m$ **by** *simp*

lemmas *mop-alt-unfolds* = *curry-def curry0-def mop-def uncurry-apply uncurry0-apply o-apply assert-true-bind-conv*

1.9.3 Constraints

lemma *add-is-pure-constraint*: $[\![PROP\ P;\ CONSTRAINT\ is\text{-}pure\ A]\!] \Longrightarrow PROP\ P$.
lemma *sepref-relpropI*: $P\ R = CONSTRAINT\ P\ R$ **by** *simp*

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*:
$P\ R \Longrightarrow IS\text{-}PURE\ P\ (pure\ R)$
by (*auto simp*: *IS-PURE-def*)

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

lemma [*fcomp-norm-simps*]: $CONSTRAINT\ (IS\text{-}PURE\ P)\ A \Longrightarrow P\ (the\text{-}pure\ A)$
by (*auto simp*: *IS-PURE-def*)

lemma *handle-purity1*:
$CONSTRAINT\ (IS\text{-}PURE\ \Phi)\ A \Longrightarrow CONSTRAINT\ \Phi\ (the\text{-}pure\ A)$
by (*auto simp*: *IS-PURE-def*)

lemma *handle-purity2*:
$CONSTRAINT\ (IS\text{-}PURE\ \Phi)\ A \Longrightarrow CONSTRAINT\ is\text{-}pure\ A$
by (*auto simp*: *IS-PURE-def*)

1.9.4 Composition

Preconditions

definition [*simp*]: $tcomp\text{-}pre\ Q\ T\ P \equiv \lambda a.\ Q\ a \wedge (\forall a'.\ (a', a) \in T \longrightarrow P\ a')$
definition *and-pre P1 P2* $\equiv \lambda x.\ P1\ x \wedge P2\ x$
definition *imp-pre P1 P2* $\equiv \lambda x.\ P1\ x \longrightarrow P2\ x$

lemma *and-pre-beta*: $PP \longrightarrow P\ x \wedge Q\ x \Longrightarrow PP \longrightarrow and\text{-}pre\ P\ Q\ x$ **by** (*auto simp*: *and-pre-def*)
lemma *imp-pre-beta*: $PP \longrightarrow P\ x \longrightarrow Q\ x \Longrightarrow PP \longrightarrow imp\text{-}pre\ P\ Q\ x$ **by** (*auto simp*: *imp-pre-def*)

definition *IMP-PRE P1 P2* ≡ ∀ *x*. *P1 x* ⟶ *P2 x*
lemma *IMP-PRED*: *IMP-PRE P1 P2* ⟹ *P1 x* ⟹ *P2 x* **unfolding** *IMP-PRE-def* **by** *auto*
lemma *IMP-PRE-refl*: *IMP-PRE P P* **unfolding** *IMP-PRE-def* **by** *auto*

definition *IMP-PRE-CUSTOM* ≡ *IMP-PRE*
lemma *IMP-PRE-CUSTOMD*: *IMP-PRE-CUSTOM P1 P2* ⟹ *IMP-PRE P1 P2* **by** (*simp add*: *IMP-PRE-CUSTOM-def*)
lemma *IMP-PRE-CUSTOMI*: ⟦⋀*x*. *P1 x* ⟹ *P2 x*⟧ ⟹ *IMP-PRE-CUSTOM P1 P2*
by (*simp add*: *IMP-PRE-CUSTOM-def IMP-PRE-def*)

lemma *imp-and-triv-pre*: *IMP-PRE P* (*and-pre* (λ-. *True*) *P*)
unfolding *IMP-PRE-def and-pre-def* **by** *auto*

Premises

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

lemma *to-IMP-LISTI*:
P ⟹ *IMP-LIST* [] *P*
by (*auto simp*: *IMP-LIST-def*)

lemma *to-IMP-LIST*: (*P* ⟹ *IMP-LIST Ps Q*) ≡ *Trueprop* (*IMP-LIST* (*P*#*Ps*) *Q*)
by (*auto simp*: *IMP-LIST-def ALL-LIST-def intro*!: *equal-intr-rule*)

lemma *from-IMP-LIST*:
Trueprop (*IMP-LIST As B*) ≡ (*ALL-LIST As* ⟹ *B*)
(*ALL-LIST* [] ⟹ *B*) ≡ *Trueprop B*
(*ALL-LIST* (*A*#*As*) ⟹ *B*) ≡ (*A* ⟹ *ALL-LIST As* ⟹ *B*)
by (*auto simp*: *IMP-LIST-def ALL-LIST-def intro*!: *equal-intr-rule*)

lemma *IMP-LIST-trivial*: *IMP-LIST A B* ⟹ *IMP-LIST A B* **.**

Composition Rules

lemma *hfcomp-tcomp-pre*:
assumes *B*: (*g*,*h*) ∈ [*Q*]$_f$ *T* → ⟨*U*⟩*nres-rel*
assumes *A*: (*f*,*g*) ∈ [*P*]$_a$ *RR′* → *S*
shows (*f*,*h*) ∈ [*tcomp-pre Q T P*]$_a$ *hrp-comp RR′ T* → *hr-comp S U*
using *hfcomp*[*OF A B*] **by** *simp*

lemma *transform-pre-param*:
assumes *A*: *IMP-LIST Cns* ((*f*, *h*) ∈ [*tcomp-pre Q T P*]$_a$ *hrp-comp RR′ T* → *hr-comp S U*)
assumes *P*: *IMP-LIST Cns* ((*P*,*P′*) ∈ *T* → *bool-rel*)

assumes *C*: *IMP-PRE PP′* (*and-pre P′ Q*)
shows *IMP-LIST Cns* ((*f*,*h*) ∈ [*PP′*]$_a$ *hrp-comp RR′ T* → *hr-comp S U*)
unfolding *from-IMP-LIST*
apply (*rule hfref-cons*)
apply (*rule A*[*unfolded from-IMP-LIST*])
apply *assumption*
apply (*drule IMP-PRED*[*OF C*])
using *P*[*unfolded from-IMP-LIST*] **unfolding** *and-pre-def*
apply (*auto dest*: *fun-relD*) []
by *simp-all*

lemma *hfref-mop-conv*: ((*g*,*mop P f*) ∈ [*Q*]$_a$ *R* → *S*) ⟷ (*g*,*f*) ∈ [λ*x*. *P x* ∧ *Q x*]$_a$ *R* → *S*
apply (*simp add*: *hfref-to-ASSERT-conv*)
apply (*fo-rule arg-cong fun-cong*)+
by (*auto intro*!: *ext simp*: *pw-eq-iff refine-pw-simps*)

lemma *hfref-op-to-mop*:
assumes *R*: (*impl*,*f*) ∈ [*Q*]$_a$ *R* → *S*
assumes *DEF*: *mf* ≡ *mop P f*
assumes *C*: *IMP-PRE PP′* (*imp-pre P Q*)
shows (*impl*,*mf*) ∈ [*PP′*]$_a$ *R* → *S*
unfolding *DEF hfref-mop-conv*
apply (*rule hfref-cons*[*OF R*])
using *C*
by (*auto simp*: *IMP-PRE-def imp-pre-def*)

lemma *hfref-mop-to-op*:
assumes *R*: (*impl*,*mf*) ∈ [*Q*]$_a$ *R* → *S*
assumes *DEF*: *mf* ≡ *mop P f*
assumes *C*: *IMP-PRE PP′* (*and-pre Q P*)
shows (*impl*,*f*) ∈ [*PP′*]$_a$ *R* → *S*
using *R* **unfolding** *DEF hfref-mop-conv*
apply (*rule hfref-cons*)
using *C*
apply (*auto simp*: *and-pre-def IMP-PRE-def*)
done

Precondition Simplification

lemma *IMP-PRE-eqI*:
assumes ⋀*x*. *P x* ⟶ *Q x*
assumes *CNV P P′*
shows *IMP-PRE P′ Q*
using *assms* **by** (*auto simp*: *IMP-PRE-def*)

lemma *simp-and1*:
assumes *Q* ⟹ *CNV P P′*
assumes *PP* ⟶ *P′* ∧ *Q*

shows $PP \longrightarrow P \wedge Q$
using *assms* **by** *auto*

lemma *simp-and2*:
assumes $P \Longrightarrow CNV\ Q\ Q'$
assumes $PP \longrightarrow P \wedge Q'$
shows $PP \longrightarrow P \wedge Q$
using *assms* **by** *auto*

lemma *triv-and1*: $Q \longrightarrow True \wedge Q$ **by** *blast*

lemma *simp-imp*:
assumes $P \Longrightarrow CNV\ Q\ Q'$
assumes $PP \longrightarrow Q'$
shows $PP \longrightarrow (P \longrightarrow Q)$
using *assms* **by** *auto*

lemma *CNV-split*:
assumes $CNV\ A\ A'$
assumes $CNV\ B\ B'$
shows $CNV\ (A \wedge B)\ (A' \wedge B')$
using *assms* **by** *auto*

lemma *CNV-prove*:
assumes P
shows $CNV\ P\ True$
using *assms* **by** *auto*

lemma *simp-pre-final-simp*:
assumes $CNV\ P\ P'$
shows $P' \longrightarrow P$
using *assms* **by** *auto*

lemma *auto-weaken-pre-uncurry-step'*:
assumes $PROTECT\ f\ a \equiv f'$
shows $PROTECT\ (uncurry\ f)\ (a,b) \equiv f'\ b$
using *assms*
by (*auto simp*: *curry-def dest*!: *meta-eq-to-obj-eq intro*!: *eq-reflection*)

1.9.5 Protected Constants

lemma *add-PR-CONST-to-def*: $x{\equiv}y \Longrightarrow PR\text{-}CONST\ x \equiv y$ **by** *simp*

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 ‹
signature SEPREF-INTF-UTIL = sig
(∗ *Miscellaneous*∗)
val list-filtered-subterms: (*term −> 'a option*) *−> term −> 'a list*

(∗ *Interface types for relations* ∗)
val get-intf-of-rel: *Proof.context −> term −> typ*

(∗ *Constraints* ∗)
(∗ *Convert relations to pure assertions* ∗)
val to-assns-rl: *bool −> Proof.context −> thm −> thm*
(∗ *Recognize, summarize and simplify CONSTRAINT − premises* ∗)
val cleanup-constraints: *Proof.context −> thm −> thm*

(∗ *Preconditions* ∗)
(∗ *Simplify precondition. Goal must be in IMP-PRE or IMP-PRE-CUSTOM form.* ∗)
val simp-precond-tac: *Proof.context −> tactic'*

(∗ *Configuration options* ∗)
val cfg-def: *bool Config.T* (∗ *decl-op*: *Define constant* ∗)
val cfg-ismop: *bool Config.T* (∗ *decl-op*: *Specified term is mop* ∗)
val cfg-mop: *bool Config.T* (∗ *decl-op*, *decl-impl*: *Derive mop* ∗)
val cfg-rawgoals: *bool Config.T* (∗ *decl-op*, *decl-impl*: *Do not pre−process/solve goals* ∗)

(∗ *TODO*: *Make do-cmd usable from ML−level*! ∗)

end

structure Sepref-Intf-Util: *SEPREF-INTF-UTIL = struct*

val cfg-debug =
Attrib.setup-config-bool @{*binding sepref-debug-intf-util*} (*K false*)

val dbg-trace = Sepref-Debugging.dbg-trace-msg cfg-debug

```
val dbg-msg-tac = Sepref-Debugging.dbg-msg-tac cfg-debug


fun list-filtered-subterms f t = let
  fun r t = case f t of
    SOME a => [a]
  | NONE => (
      case t of
        t1$t2 => r t1 @ r t2
      | Abs (-,-,t) => r t
      | - => []
    )
in
  r t
end

fun get-intf-of-rel ctxt R =
  Term-Synth.synth-term @{thms synth-intf-of-relI} ctxt R
    |> fastype-of
    |> Refine-Util.dest-itselfT

local
  fun add-is-pure-constraint ctxt v thm = let
    val v = Thm.cterm-of ctxt v
   val rl = Drule.infer-instantiate' ctxt [NONE, SOME v] @{thm add-is-pure-constraint}
  in
    thm RS rl
  end
in
  fun to-assns-rl add-pure-constr ctxt thm = let
    val orig-ctxt = ctxt

    val (thm,ctxt) = yield-singleton (apfst snd oo Variable.importT) thm ctxt

   val (R,S) = case Thm.concl-of thm of @{mpat Trueprop (-∈fref - ?R ?S)}
=> (R,S)
      | - => raise THM(to-assns-rl: expected fref−thm,~1,[thm])

    fun mk-cn-subst (fname,(iname,C,A)) =
      let
        val T' = A --> C --> @{typ assn}
        val v' = Free (fname,T')
        val ct' = @{mk-term the-pure ?v'} |> Thm.cterm-of ctxt
      in
        (v',(iname,ct'))
      end

       fun relation-flt (name,Type (@{type-name set},[Type (@{type-name
prod},[C,A])])) = SOME (name,C,A)
```

```
    | relation-flt - = NONE


    val vars = []
      |> Term.add-vars R
      |> Term.add-vars S
      |> map-filter (relation-flt)
    val (names,ctxt) = Variable.variant-fixes (map (#1 #> fst) vars) ctxt

    val cn-substs = map mk-cn-subst (names ~~ vars)


    val thm = Drule.infer-instantiate ctxt (map snd cn-substs) thm

        val thm = thm |> add-pure-constr ? fold (fn (v,-) => fn thm =>
add-is-pure-constraint ctxt v thm) cn-substs

    val thm = singleton (Variable.export ctxt orig-ctxt) thm
  in
    thm
  end

  fun cleanup-constraints ctxt thm = let
    val orig-ctxt = ctxt

     val (thm, ctxt) = yield-singleton (apfst snd oo Variable.import true) thm
ctxt

  val xform-thms = Named-Theorems.get ctxt @{named-theorems sepref-relprops-transform}
  val rprops-thms = Named-Theorems.get ctxt @{named-theorems sepref-relprops}
  val simp-thms = Named-Theorems.get ctxt @{named-theorems sepref-relprops-simps}

    fun simp thms = Conv.fconv-rule (
          Simplifier.asm-full-rewrite
            (put-simpset HOL-basic-ss ctxt addsimps thms))

    (* Check for pure (the-pure R) - patterns *)

    local
      val (-,R,S) = case Thm.concl-of thm of
        @{mpat Trueprop (-∈hfref ?P ?R ?S)} => (P,R,S)
      | @{mpat Trueprop (-∈fref ?P ?R ?S)} => (P,R,S)
  | - => raise THM(cleanup-constraints: Expected hfref or fref-theorem,~1,[thm])



    fun flt-pat @{mpat pure (the-pure ?A)} = SOME A | flt-pat - = NONE

    val purify-terms =
```

```
      (list-filtered-subterms flt-pat R @ list-filtered-subterms flt-pat S)
      |> distinct op aconv

    val thm = fold (add-is-pure-constraint ctxt) purify-terms thm
  in
    val thm = thm
  end

  val thm = thm
    |> Local-Defs.unfold0 ctxt xform-thms
    |> Local-Defs.unfold0 ctxt rprops-thms

  val insts = map (fn
       @{mpat Trueprop (CONSTRAINT - (the-pure -))} => @{thm han-
dle-purity1}
    | - => asm-rl
    ) (Thm.prems-of thm)

  val thm = (thm OF insts)
    |> Conv.fconv-rule Thm.eta-conversion
    |> simp @{thms handle-purity2}
    |> simp simp-thms

  val thm = singleton (Variable.export ctxt orig-ctxt) thm

 in
   thm
 end
end

fun simp-precond-tac ctxt = let
    fun simp-only thms = asm-full-simp-tac (put-simpset HOL-basic-ss ctxt
addsimps thms)
  val rtac = resolve-tac ctxt

  val cnv-ss = ctxt delsimps @{thms CNV-def}

 (*val uncurry-tac = SELECT-GOAL (ALLGOALS (DETERM o SOLVED'
(
   REPEAT' (rtac @{thms auto-weaken-pre-uncurry-step'})
   THEN' rtac @{thms auto-weaken-pre-uncurry-finish}
)))*)

    val prove-cnv-tac = SOLVED' (rtac @{thms CNV-prove} THEN' SE-
LECT-GOAL (auto-tac ctxt))

  val do-cnv-tac =
    (cp-clarsimp-tac cnv-ss) THEN-ALL-NEW
    (TRY o REPEAT-ALL-NEW (match-tac ctxt @{thms CNV-split}))
```

```
    THEN-ALL-NEW (prove-cnv-tac ORELSE' rtac @{thms CNV-I})

  val final-simp-tac =
    rtac @{thms simp-pre-final-simp}
    THEN' cp-clarsimp-tac cnv-ss
    THEN' dbg-msg-tac (Sepref-Debugging.msg-subgoal final-simp-tac: Before
CNV-I) ctxt
    THEN' rtac @{thms CNV-I}
    THEN' dbg-msg-tac (Sepref-Debugging.msg-text Final−Simp done) ctxt

  (*val curry-tac = let open Conv in
    CONVERSION (Refine-Util.HOL-concl-conv (fn ctxt => arg1-conv (
    top-conv ( fn - => try-conv (rewr-conv @{thm uncurry-def})) ctxt)) ctxt)
    THEN' REPEAT' (EqSubst.eqsubst-tac ctxt [1] @{thms case-prod-eta})
    THEN' rtac @{thms CNV-I}
    end*)

  val simp-tupled-pre-tac =
      SELECT-GOAL (Local-Defs.unfold0-tac ctxt @{thms prod-casesK un-
curry0-hfref-post})
    THEN' REPEAT' (EqSubst.eqsubst-tac ctxt [1] @{thms case-prod-eta})
    THEN' rtac @{thms CNV-I}

 val unfold-and-tac = rtac @{thms and-pre-beta} THEN-ALL-NEW simp-only
@{thms split}

  val simp-and1-tac =
    rtac @{thms simp-and1} THEN' do-cnv-tac

  val simp-and2-tac =
    rtac @{thms simp-and2} THEN' do-cnv-tac

  val and-plan-tac =
    simp-and1-tac
    THEN' dbg-msg-tac (Sepref-Debugging.msg-subgoal State after and1) ctxt
    THEN' (
      rtac @{thms triv-and1}
      ORELSE'
      dbg-msg-tac (Sepref-Debugging.msg-subgoal Invoking and2 on) ctxt
      THEN' simp-and2-tac
          THEN' dbg-msg-tac (Sepref-Debugging.msg-subgoal State before fi-
nal-simp-tac) ctxt
      THEN' final-simp-tac
    )

 val unfold-imp-tac = rtac @{thms imp-pre-beta} THEN-ALL-NEW simp-only
@{thms split}
  val simp-imp1-tac =
    rtac @{thms simp-imp} THEN' do-cnv-tac
```

```
  val imp-plan-tac = simp-imp1-tac THEN' final-simp-tac

  val imp-pre-tac = APPLY-LIST [
      simp-only @{thms split-tupled-all}
      THEN' Refine-Util.instantiate-tuples-subgoal-tac ctxt
      THEN' CASES' [
        (unfold-and-tac, ALLGOALS and-plan-tac),
        (unfold-imp-tac, ALLGOALS imp-plan-tac)
      ]
    ,
      simp-tupled-pre-tac
    ]

  val imp-pre-custom-tac =
  SELECT-GOAL (Local-Defs.unfold0-tac ctxt @{thms and-pre-def}) THEN'
    TRY o SOLVED' (SELECT-GOAL (auto-tac ctxt))

in
  CASES' [
    (rtac @{thms IMP-PRE-eqI}, imp-pre-tac 1),
    (rtac @{thms IMP-PRE-CUSTOMI}, ALLGOALS imp-pre-custom-tac)
  ]
end




local
  fun inf-bn-aux name =
    case String.tokens (fn c => c = #.) name of
      [] => NONE
    | [a] => SOME (Binding.name a)
    | (-::a::-) => SOME (Binding.name a)
in
  fun infer-basename (Const (-type-constraint-,-)$t) = infer-basename t
    | infer-basename (Const (name,-)) = inf-bn-aux name
    | infer-basename (Free (name,-)) = inf-bn-aux name
    | infer-basename - = NONE
end

val cfg-mop = Attrib.setup-config-bool @{binding sepref-register-mop} (K true)
 val cfg-ismop = Attrib.setup-config-bool @{binding sepref-register-ismop} (K
false)
 val cfg-rawgoals = Attrib.setup-config-bool @{binding sepref-register-rawgoals}
(K false)
 val cfg-transfer = Attrib.setup-config-bool @{binding sepref-decl-impl-transfer}
(K true)
 val cfg-def = Attrib.setup-config-bool @{binding sepref-register-def} (K true)
```

```
    val cfg-register = Attrib.setup-config-bool @{binding sepref-decl-impl-register}
(K true)

    local
      open Refine-Util
      val flags =
            parse-bool-config' mop cfg-mop
        || parse-bool-config' ismop cfg-ismop
        || parse-bool-config' rawgoals cfg-rawgoals
        || parse-bool-config' def cfg-def
      val parse-flags = parse-paren-list' flags

      val parse-name = Scan.option (Parse.binding --| @{keyword :})
      val parse-relconds = Scan.optional (@{keyword where} |-- Parse.and-list1
(Scan.repeat1 Parse.prop) >> flat) []
    in

     val do-parser = parse-flags -- parse-name -- Parse.term --| @{keyword
::} -- Parse.term -- parse-relconds
    end


    fun do-cmd ((((flags,name),opt-raw), relt-raw),relconds-raw) lthy = let
      local
        val ctxt = Refine-Util.apply-configs flags lthy
      in
        val flag-ismop = Config.get ctxt cfg-ismop
        val flag-mop = Config.get ctxt cfg-mop andalso not flag-ismop
        val flag-rawgoals = Config.get ctxt cfg-rawgoals
        val flag-def = Config.get ctxt cfg-def
      end

      open Sepref-Basic Sepref-Rules

      val relt = Syntax.parse-term lthy relt-raw
      val relconds = map (Syntax.parse-prop lthy) relconds-raw

      val - = dbg-trace lthy Parse relation and relation conditions together
      val relt = Const (@{const-name Pure.term}, dummyT) $ relt
      local
        val l = Syntax.check-props lthy (relt::relconds)
      in
        val (relt, relconds) = (hd l, tl l)
      end
      val relt = Logic.dest-term relt

      val opt-pre = Syntax.parse-term lthy opt-raw
```

```
        val - = dbg-trace lthy Infer basename
        val name = case name of
          SOME name => name
        | NONE => (
            case infer-basename opt-pre of
                NONE => (error Could not infer basename: You have to specify a
basename; Binding.empty)
              | SOME name => name
          )


        fun qname s n = Binding.qualify true (Binding.name-of n) (Binding.name
s)
        fun def name t-pre attribs lthy = let
          val t = Syntax.check-term lthy t-pre
            (*|> Thm.cterm-of lthy
            |> Drule.mk-term
            |> Local-Defs.unfold0 lthy @{thms PR-CONST-def}
            |> Drule.dest-term
            |> Thm.term-of*)

          val lthy = (snd o Local-Theory.begin-nested) lthy
          val ((dt,(-,thm)),lthy) = Local-Theory.define
        ((name,Mixfix.NoSyn),((Thm.def-binding name, Code.singleton-default-equation-attrib
:: attribs),t)) lthy;
          val (lthy, lthy-old) = 'Local-Theory.end-nested lthy
          val phi = Proof-Context.export-morphism lthy-old lthy
          val thm = Morphism.thm phi thm
          val dt = Morphism.term phi dt

        in
          ((dt,thm),lthy)
        end

        val - = dbg-trace lthy Analyze Relation
        val (pre,args,res) = analyze-rel relt
        val specified-pre = is-some pre
        val pre = the-default (mk-triv-precond args) pre

        val def-thms = @{thms PR-CONST-def}

        val - = dbg-trace lthy Define op
        val op-name = Binding.prefix-name (if flag-ismop then mop- else op-) name
        val (def-thms,opc,lthy) =
          if flag-def then let
              val ((opc,op-def-thm),lthy) = def op-name opt-pre @{attributes [simp]}
lthy
              val opc = Refine-Util.dummify-tvars opc
              val def-thms = op-def-thm::def-thms
```

```
      in
        (def-thms,opc,lthy)
      end
    else let
        val - = dbg-trace lthy Refine type of opt-pre to get opc
        val opc = Syntax.check-term lthy opt-pre
        val new-ctxt = Variable.declare-term opc lthy
        val opc = singleton (Variable.export-terms new-ctxt lthy) opc
          |> Refine-Util.dummify-tvars
      in
        (def-thms,opc,lthy)
      end


(* PR-CONST Heuristics *)
fun pr-const-heuristics basename c-pre lthy = let
  val - = dbg-trace lthy (PR-CONST heuristics ^@{make-string} c-pre)

  val c = Syntax.check-term lthy c-pre
in
  case c of
    @{mpat PR-CONST -} => ((c-pre,false),lthy)
  | Const - => ((c-pre,false),lthy)
  | - => let
      val (f,args) = strip-comb c

      val lthy = case f of Const - => let
          val ctxt = Variable.declare-term c lthy
          val lhs = Autoref-Tagging.list-APP (f,args)
          val rhs = @{mk-term UNPROTECT ?c}
          val goal = Logic.mk-equals (lhs,rhs) |> Thm.cterm-of ctxt
          val tac =
            Local-Defs.unfold0-tac ctxt @{thms APP-def UNPROTECT-def}
            THEN ALLGOALS (simp-tac (put-simpset HOL-basic-ss ctxt))
          val thm = Goal.prove-internal ctxt [] goal (K tac)
            |> singleton (Variable.export ctxt lthy)

          val (-,lthy) = Local-Theory.note
        ((Binding.suffix-name -def-pat basename,@{attributes [def-pat-rules]}),[thm])
lthy

          val - = Thm.pretty-thm lthy thm |> Pretty.string-of |> writeln
        in
          lthy
        end
      | - => (
        Pretty.block [
            Pretty.str Complex operation pattern. Added PR-CONST but no
pattern rules:,
```

```
          Pretty.brk 1,Syntax.pretty-term lthy c]
        |> Pretty.string-of |> warning
        ; lthy)

      val c-pre = Const(@{const-name PR-CONST},dummyT)$c-pre
    in
      ((c-pre,true),lthy)
    end
end

val ((opc,-),lthy) = pr-const-heuristics op-name opc lthy

(* Register *)
val arg-intfs = map (get-intf-of-rel lthy) args
val res-intf = get-intf-of-rel lthy res


fun register basename c lthy = let
  val - = dbg-trace lthy Register
  open Sepref-Basic
  val c = Syntax.check-term lthy c

  val ri = case (is-nresT (body-type (fastype-of c)), is-nresT res-intf) of
    (true,false) => mk-nresT res-intf
  | (false,true) => dest-nresT res-intf
  | - => res-intf

  val iT = arg-intfs ---> ri

    val ((-,itype-thm),lthy) = Sepref-Combinator-Setup.sepref-register-single
(Binding.name-of basename) c iT lthy
    val - = Document-Output.pretty-thm lthy itype-thm |> Pretty.string-of |>
writeln

in
  lthy
end

val lthy = register op-name opc lthy

val - = dbg-trace lthy Define pre
val pre-name = Binding.prefix-name pre- name
val ((prec,pre-def-thm),lthy) = def pre-name pre @{attributes [simp]} lthy
val prec = Refine-Util.dummify-tvars prec
val def-thms = pre-def-thm::def-thms

(* Re-integrate pre-constant into type-context of relation. TODO: This is
probably not clean/robust *)
val pre = constrain-type-pre (fastype-of pre) prec |> Syntax.check-term lthy
```

```
      val - = dbg-trace lthy Convert both, relation and operation to uncurried form,
and add nres
      val - = dbg-trace lthy Convert relation (arguments have already been separated
by analyze−rel)
        val res = case res of @{mpat ⟨-⟩nres-rel} => res | - => @{mk-term
⟨?res⟩nres-rel}
      val relt = mk-rel (SOME pre,args,res)

      val - = dbg-trace lthy Convert operation
      val opcT = fastype-of (Syntax.check-term lthy opc)
      val op-is-nres = Sepref-Basic.is-nresT (body-type opcT)
      val (opcu, op-ar) = let
        val arity = binder-types #> length
       (* Arity of operation is number of arguments before result (which may be a
fun−type! )*)
        val res-ar = arity (Relators.rel-absT res |> not op-is-nres ? dest-nresT)

        val op-ar = arity opcT − res-ar

        val - = op-ar = length args orelse
          raise TERM(Operation/relation arity mismatch: ^ string-of-int op-ar ^
vs ^ string-of-int (length args),[opc,relt])

        (* Add RETURN o...o if necessary*)
        val opc =
          if op-is-nres then opc
       else mk-compN-pre op-ar (Const(@{const-name Refine-Basic.RETURN},dummyT))
opc

        (* Add uncurry if necessary *)
        val opc = mk-uncurryN-pre op-ar opc
      in
        (opc, op-ar)
      end

      (* Build mop−variant *)
      val declare-mop = (specified-pre orelse not op-is-nres) andalso flag-mop

      val (mop-data,lthy) = if declare-mop then let
          val - = dbg-trace lthy mop definition
          val mop-rhs = Const(@{const-name mop},dummyT) $ prec $ opcu
            |> mk-curryN-pre op-ar
          val mop-name = Binding.prefix-name mop- name
          val ((mopc,mop-def-thm),lthy) = def mop-name mop-rhs [] lthy
          val mopc = Refine-Util.dummify-tvars mopc

           val ((mopc,added-pr-const),lthy) = pr-const-heuristics mop-name mopc
```

```
lthy

            val mop-def-thm' = if added-pr-const then
                mop-def-thm RS @{thm add-PR-CONST-to-def}
              else mop-def-thm

          val (-,lthy) = Local-Theory.note ((Binding.empty, @{attributes [sepref-mop-def-thms]}),[mop-def-thm'])
lthy

            val - = dbg-trace lthy mop alternative definition
            val alt-unfolds = @{thms mop-alt-unfolds}
              |> not specified-pre ? curry op :: pre-def-thm

            val mop-alt-thm = Local-Defs.unfold0 lthy alt-unfolds mop-def-thm
              |> Refine-Util.shift-lambda-leftN op-ar
          val (-,lthy) = Local-Theory.note ((Binding.suffix-name -alt mop-name,@{attributes
[simp]}),[mop-alt-thm]) lthy

            val - = dbg-trace lthy mop: register
            val lthy = register mop-name mopc lthy

            val - = dbg-trace lthy mop: vcg theorem
            local
              val Ts = map Relators.rel-absT args
              val ctxt = Variable.declare-thm mop-def-thm lthy
              val ctxt = fold Variable.declare-typ Ts ctxt
              val (x,ctxt) = Refine-Util.fix-left-tuple-from-Ts x Ts ctxt

              val mop-def-thm = mop-def-thm
                |> Local-Defs.unfold0 ctxt @{thms curry-shl}

              fun prep-thm thm = (thm OF [mop-def-thm])
                |> Drule.infer-instantiate' ctxt [SOME (Thm.cterm-of ctxt x)]
                    |> Local-Defs.unfold0 ctxt @{thms uncurry-apply uncurry0-apply
o-apply}
                |> Local-Defs.unfold0 ctxt (def-thms @
                    @{thms Product-Type.split HOL.True-implies-equals})
                |> singleton (Variable.export ctxt lthy)

            val thms = map prep-thm @{thms mop-spec-rl-from-def mop-leof-rl-from-def}


            in
               val (-,lthy) = Local-Theory.note ((qname vcg mop-name,@{attributes
[refine-vcg]}),thms) lthy
            end

          in
            (SOME (mop-name,mopc,mop-def-thm),lthy)
```

```
    end
  else (NONE,lthy)


  val - = dbg-trace lthy Build Parametricity Theorem
  val param-t = mk-pair-in-pre opcu opcu relt
    |> Syntax.check-term lthy
    |> HOLogic.mk-Trueprop
    |> curry Logic.list-implies relconds

  val - = dbg-trace lthy Build Parametricity Theorem for Precondition
  val param-pre-t =
    let
      val pre-relt = Relators.mk-fun-rel (Relators.list-prodrel-left args) @{term
bool-rel}

      val param-pre-t = mk-pair-in-pre prec prec pre-relt
        |> Syntax.check-term lthy
        |> HOLogic.mk-Trueprop
        |> curry Logic.list-implies relconds
    in
      param-pre-t
    end


  val - = dbg-trace lthy Build goals
  val goals = [[ (param-t, []), (param-pre-t, []) ]]

  fun after-qed [[p-thm, pp-thm]] - (*ctxt*) =
    let
      val - = dbg-trace lthy after-qed
          val p-thm' = p-thm |> not specified-pre ? Local-Defs.unfold0 lthy
[pre-def-thm]

        val (-,lthy) = Local-Theory.note ((qname fref op-name,@{attributes
[sepref-fref-thms]}), [p-thm']) lthy
       val (-,lthy) = Local-Theory.note ((qname param pre-name,@{attributes
[param]}), [pp-thm]) lthy

      val p'-unfolds = pre-def-thm :: @{thms True-implies-equals}
    val (-,lthy) = Local-Theory.note ((qname fref' op-name,[]), [Local-Defs.unfold0
lthy p'-unfolds p-thm]) lthy


      val lthy = case mop-data of NONE => lthy |
        SOME (mop-name,mopc,mop-def-thm) => let
          val - = dbg-trace lthy Build and prove mop-stuff
          (* mop - parametricity theorem: (uncurry^n mopc,uncurry^n mopc) ∈
args →_f res *)
```

```
            val mopcu = mk-uncurryN-pre op-ar mopc
          val param-mop-t = mk-pair-in-pre mopcu mopcu (mk-rel (NONE,args,res))
            |> Syntax.check-term lthy
            |> HOLogic.mk-Trueprop
            |> curry Logic.list-implies relconds

            val ctxt = Proof-Context.augment param-mop-t lthy

            val tac = let
             val p-thm = Local-Defs.unfold0 ctxt @{thms PR-CONST-def} p-thm
            in
             Local-Defs.unfold0-tac ctxt (mop-def-thm :: @{thms PR-CONST-def
uncurry-curry-id uncurry-curry0-id})
              THEN FIRSTGOAL (
                dbg-msg-tac (Sepref-Debugging.msg-subgoal Mop−param thm goal
after unfolding) ctxt THEN'
                resolve-tac ctxt @{thms param-mopI}
                  THEN' SOLVED' (resolve-tac ctxt [p-thm] THEN-ALL-NEW
assume-tac ctxt)
                  THEN' SOLVED' (resolve-tac ctxt [pp-thm] THEN-ALL-NEW
assume-tac ctxt)
              )
            end

                val pm-thm = Goal.prove-internal lthy [] (Thm.cterm-of ctxt
param-mop-t) (K tac)
              |> singleton (Variable.export ctxt lthy)

            val (-,lthy) = Local-Theory.note ((qname fref mop-name,@{attributes
[sepref-fref-thms]}), [pm-thm]) lthy
                val (-,lthy) = Local-Theory.note ((qname fref' mop-name,[]),
[Local-Defs.unfold0 lthy p'-unfolds pm-thm]) lthy


          in
            lthy
          end


        in
          lthy
        end
      | after-qed thmss - = raise THM (After−qed: Wrong thmss structure,~1,flat
thmss)

      fun std-tac ctxt = let
      val ptac = REPEAT-ALL-NEW-FWD (Parametricity.net-tac (Parametricity.get-dflt
ctxt) ctxt)
```

```
        (* Massage simpset a bit *)
        val ctxt = ctxt
          |> Context-Position.set-visible false
          |> Context.proof-map (Thm.attribute-declaration Clasimp.iff-del @{thm
pair-in-Id-conv})

      in
        if flag-rawgoals then
          all-tac
        else
          Local-Defs.unfold0-tac ctxt def-thms THEN ALLGOALS (
            TRY o SOLVED' (
              TRY o resolve-tac ctxt @{thms frefI}
                  THEN' TRY o REPEAT-ALL-NEW (ematch-tac ctxt @{thms
prod-relE})
                THEN' simp-tac (put-simpset HOL-basic-ss ctxt addsimps @{thms
split uncurry-apply uncurry0-apply})
              THEN' (
                SOLVED' (ptac THEN-ALL-NEW asm-full-simp-tac ctxt)
                ORELSE' SOLVED' (cp-clarsimp-tac ctxt THEN-ALL-NEW-FWD
ptac THEN-ALL-NEW SELECT-GOAL (auto-tac ctxt))
              )
            )
          )

      end

       val rf-std = Proof.refine (Method.Basic (fn ctxt => SIMPLE-METHOD
(std-tac ctxt)))
         #> Seq.the-result do-cmd: Standard proof tactic returned empty result
sequence

    in
      Proof.theorem NONE after-qed goals lthy
      |> rf-std
    end

  val - = Outer-Syntax.local-theory-to-proof @{command-keyword sepref-decl-op}
      (do-parser >> do-cmd)




    local

      fun unfold-PR-CONST-tac ctxt = SELECT-GOAL (Local-Defs.unfold0-tac
ctxt @{thms PR-CONST-def})

      fun transfer-precond-rl ctxt t R = let
```

```
        (*val tfrees = Term.add-tfreesT (fastype-of t) []
         val t' = map-types (map-type-tfree (fn x => if member op= tfrees x then
dummyT else TFree x)) t
        *) (* TODO: Brute force approach, that may generalize too much! *)
        val t' = map-types (K dummyT) t

        val goal = Sepref-Basic.mk-pair-in-pre t t' R
          |> Syntax.check-term ctxt
          |> Thm.cterm-of ctxt

            val thm = Drule.infer-instantiate' ctxt [NONE,SOME goal] @{thm
IMP-LIST-trivial}

      in
        thm
      end


      (* Generate a hnr-thm for mop given one for op *)
      fun generate-mop-thm ctxt op-thm = let
        val orig-ctxt = ctxt

          val (op-thm, ctxt) = yield-singleton (apfst snd oo Variable.import true)
op-thm ctxt

        (* Convert mop-def-thms to form uncurry^n f ≡ mop P g *)
        val mop-def-thms = Named-Theorems-Rev.get ctxt @{named-theorems-rev
sepref-mop-def-thms}
          |> map (Local-Defs.unfold0 ctxt @{thms curry-shl})

        fun fail-hnr-tac - - = raise THM(Invalid hnr-theorem,~1,[op-thm])
        fun fail-mop-def-tac i st = let
          val g = nth (Thm.prems-of st) (i-1)
        in
          raise TERM(Found no matching mop-definition,[g])
        end

        (* Tactic to solve preconditions of hfref-op-to-mop *)
        val tac = APPLY-LIST [
          resolve-tac ctxt [op-thm] ORELSE' fail-hnr-tac,
           ((*unfold-PR-CONST-tac ctxt THEN'*) resolve-tac ctxt mop-def-thms)
ORELSE' fail-mop-def-tac,
            simp-precond-tac ctxt ORELSE' Sepref-Debugging.error-tac' precond
simplification failed ctxt
        ] 1

        (* Do synthesis *)
        val st = @{thm hfref-op-to-mop}
        val st = Goal.protect (Thm.nprems-of st) st
```

```
    val mop-thm = tac st |> Seq.hd |> Goal.conclude

    val mop-thm = singleton (Variable.export ctxt orig-ctxt) mop-thm
      |> Sepref-Rules.norm-fcomp-rule orig-ctxt
  in mop-thm end

  (* Generate a hnr-thm for op given one for mop *)
  fun generate-op-thm ctxt mop-thm = let (* TODO: Almost-clone of generate-mop-thm *)
    val orig-ctxt = ctxt

    val (mop-thm, ctxt) = yield-singleton (apfst snd oo Variable.import true) mop-thm ctxt

    (* Convert mop-def-thms to form uncurry^n f ≡ mop P g *)
    val mop-def-thms = Named-Theorems-Rev.get ctxt @{named-theorems-rev sepref-mop-def-thms}
      |> map (Local-Defs.unfold0 ctxt @{thms curry-shl})

    fun fail-hnr-tac - - = raise THM(Invalid hnr-theorem,~1,[mop-thm])
    fun fail-mop-def-tac i st = let
      val g = nth (Thm.prems-of st) (i-1)
    in
      raise TERM(Found no matching mop-definition,[g])
    end

    (* Tactic to solve preconditions of hfref-mop-to-op *)
    val tac = APPLY-LIST [
      resolve-tac ctxt [mop-thm] ORELSE' fail-hnr-tac,
       ((*unfold-PR-CONST-tac ctxt THEN'*) resolve-tac ctxt mop-def-thms) ORELSE' fail-mop-def-tac,
          simp-precond-tac ctxt ORELSE' Sepref-Debugging.error-tac' precond simplification failed ctxt
    ] 1

    (* Do synthesis *)
    val st = @{thm hfref-mop-to-op}
    val st = Goal.protect (Thm.nprems-of st) st
    val op-thm = tac st |> Seq.hd |> Goal.conclude

    val op-thm = singleton (Variable.export ctxt orig-ctxt) op-thm
      |> Sepref-Rules.norm-fcomp-rule orig-ctxt
  in op-thm end



  fun chk-result ctxt thm = let
    val (-,R,S) = case Thm.concl-of thm of
      @{mpat Trueprop (-∈hfref ?P ?R ?S)} => (P,R,S)
```

```
      | - => raise THM(chk-result: Expected hfref-theorem,~1,[thm])

      fun err t = let
        val ts = Syntax.pretty-term ctxt t |> Pretty.string-of
      in
        raise THM (chk-result: Invalid pattern left in assertions: ^ts,~1,[thm])
      end
      fun check-invalid (t as @{mpat hr-comp - -}) = err t
        | check-invalid (t as @{mpat hrp-comp - -}) = err t
        | check-invalid (t as @{mpat pure (the-pure -)}) = err t
        | check-invalid (t as @{mpat - O -}) = err t
        | check-invalid - = false


      val - = exists-subterm check-invalid R
      val - = exists-subterm check-invalid S
    in
      ()
    end

    fun to-IMP-LIST ctxt thm =
        (thm RS @{thm to-IMP-LISTI}) |> Local-Defs.unfold0 ctxt @{thms
to-IMP-LIST}

      fun from-IMP-LIST ctxt thm = thm |> Local-Defs.unfold0 ctxt @{thms
from-IMP-LIST}

  in

    local
      open Refine-Util
      val flags =
          parse-bool-config' mop cfg-mop
        || parse-bool-config' ismop cfg-ismop
        || parse-bool-config' transfer cfg-transfer
        || parse-bool-config' rawgoals cfg-rawgoals
        || parse-bool-config' register cfg-register
      val parse-flags = parse-paren-list' flags

        val parse-precond = Scan.option (@{keyword [} |-- Parse.term --|
@{keyword ]})

      val parse-fref-thm = Scan.option (@{keyword uses} |-- Parse.thm)

    in
    val di-parser = parse-flags -- Scan.optional (Parse.binding --| @{keyword
:}) Binding.empty -- parse-precond -- Parse.thm -- parse-fref-thm
    end
```

```
fun di-cmd ((((flags,name), precond-raw), i-thm-raw), p-thm-raw) lthy = let
  val i-thm = singleton (Attrib.eval-thms lthy) i-thm-raw
  val p-thm = map-option (singleton (Attrib.eval-thms lthy)) p-thm-raw

  local
    val ctxt = Refine-Util.apply-configs flags lthy
  in
    val flag-mop = Config.get ctxt cfg-mop
    val flag-ismop = Config.get ctxt cfg-ismop
    val flag-rawgoals = Config.get ctxt cfg-rawgoals
    val flag-transfer = Config.get ctxt cfg-transfer
    val flag-register = Config.get ctxt cfg-register
  end

  val fr-attribs = if flag-register then @{attributes [sepref-fr-rules]} else []


  val ctxt = lthy

  (* Compose with fref-theorem *)
  val - = dbg-trace lthy Compose with fref

  local
    val hf-tcomp-pre = @{thm hfcomp-tcomp-pre} OF [asm-rl,i-thm]
    fun compose p-thm = let
      val p-thm = p-thm |> to-assns-rl false lthy
    in
      hf-tcomp-pre OF [p-thm]
    end

  in
    val thm = case p-thm of
      SOME p-thm => compose p-thm
    | NONE => let
        val p-thms = Named-Theorems-Rev.get ctxt @{named-theorems-rev
sepref-fref-thms}

        fun err () = let
        val prem-s = nth (Thm.prems-of hf-tcomp-pre) 0 |> Syntax.pretty-term
ctxt |> Pretty.string-of
        in
          error (Found no fref-theorem matching ^ prem-s)
        end

      in
        case get-first (try compose) p-thms of
          NONE => err ()
        | SOME thm => thm
```

```
      end
    end

     val (thm,ctxt) = yield-singleton (apfst snd oo Variable.import true) thm
ctxt

    val - = dbg-trace lthy Transfer Precond
    val thm = to-IMP-LIST ctxt thm
    val thm = thm RS @{thm transform-pre-param}

    local
      val (pre,R,pp-name,pp-type) = case Thm.prems-of thm of
          [@{mpat Trueprop (IMP-LIST - ((?pre,-)∈?R))}, @{mpat Trueprop
(IMP-PRE (mpaq-STRUCT (mpaq-Var ?pp-name ?pp-type)) -)}] => (pre,R,pp-name,pp-type)
        | - => raise THM(di-cmd: Cannot recognize first prems of trans-
form-pre-param: , ~1,[thm])

    in
       val thm = if flag-transfer then thm OF [transfer-precond-rl ctxt pre R]
else thm

      val thm = case precond-raw of
        NONE => thm
      | SOME precond-raw => let
          val precond = Syntax.parse-term ctxt precond-raw
            |> Sepref-Basic.constrain-type-pre pp-type
            |> Syntax.check-term ctxt
            |> Thm.cterm-of ctxt

          val thm = Drule.infer-instantiate ctxt [(pp-name,precond)] thm
          val thm = thm OF [asm-rl,@{thm IMP-PRE-CUSTOMD}]
        in
          thm
        end

    end

    val - = dbg-trace lthy Build goals
    val goals = [map (fn x => (x,[])) (Thm.prems-of thm)]

    fun after-qed thmss - = let
      val - = dbg-trace lthy After QED
      val prems-thms = hd thmss

      val thm = thm OF prems-thms

      val thm = from-IMP-LIST ctxt thm

      (* Two rounds of cleanup−constraints, norm-fcomp *)
```

```
val - = dbg-trace lthy Cleanup
val thm = thm
  |> cleanup-constraints ctxt
  |> Sepref-Rules.norm-fcomp-rule ctxt
  |> cleanup-constraints ctxt
  |> Sepref-Rules.norm-fcomp-rule ctxt

val thm = thm
  |> singleton (Variable.export ctxt lthy)
  |> zero-var-indexes

val - = dbg-trace lthy Check Result
val - = chk-result lthy thm


  fun qname suffix = if Binding.is-empty name then name else Bind-
ing.suffix-name suffix name

val thm-name = if flag-ismop then qname -hnr-mop else qname -hnr
val (-,lthy) = Local-Theory.note ((thm-name,fr-attribs),[thm]) lthy

val - = Thm.pretty-thm lthy thm |> Pretty.string-of |> writeln

(* Create mop theorem from op−theorem *)
val cr-mop-thm = flag-mop andalso not flag-ismop
val lthy =
  if cr-mop-thm then
    let
      val - = dbg-trace lthy Create mop−thm
      val mop-thm = thm
        |> generate-mop-thm lthy
        |> zero-var-indexes

 val (-,lthy) = Local-Theory.note ((qname -hnr-mop,fr-attribs),[mop-thm])
lthy
      val - = Thm.pretty-thm lthy mop-thm |> Pretty.string-of |> writeln
    in lthy end
  else lthy

(* Create op theorem from mop−theorem *)
val cr-op-thm = flag-ismop
val lthy =
  if cr-op-thm then
    let
      val - = dbg-trace lthy Create op−thm
      val op-thm = thm
        |> generate-op-thm lthy
        |> zero-var-indexes
```

```
                val (-,lthy) = Local-Theory.note ((qname -hnr,fr-attribs),[op-thm])
lthy
                val - = Thm.pretty-thm lthy op-thm |> Pretty.string-of |> writeln
              in lthy end
            else lthy


        in
          lthy
        end

        fun std-tac ctxt = let
          val ptac = REPEAT-ALL-NEW-FWD (
                Parametricity.net-tac (Parametricity.get-dflt ctxt) ctxt ORELSE'
assume-tac ctxt
            )
        in
          if flag-rawgoals orelse not flag-transfer then
            all-tac
          else
            APPLY-LIST [
            SELECT-GOAL (Local-Defs.unfold0-tac ctxt @{thms from-IMP-LIST})
THEN' TRY o SOLVED' ptac,
              simp-precond-tac ctxt
            ] 1

        end

        val rf-std = Proof.refine (Method.Basic (fn ctxt => SIMPLE-METHOD
(std-tac ctxt)))
            #> Seq.the-result di-cmd: Standard proof tactic returned empty result
sequence

      in
        Proof.theorem NONE after-qed goals ctxt
        |> rf-std

      end

    val - = Outer-Syntax.local-theory-to-proof @{command-keyword sepref-decl-impl}
        (di-parser >> di-cmd)
    end

  end
›
```

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 $SPEC\ (\lambda x.\ x{=}r) \leq_n m \Longrightarrow f \leq_n m$
using *assms*
by (*auto simp*: *pw-le-iff pw-leof-iff*)

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 $[\![\Phi \Longrightarrow SPEC\ (\lambda x.\ x{=}r) \leq_n m]\!] \Longrightarrow f \leq_n m$
using *assms*
by (*auto simp*: *pw-le-iff pw-leof-iff refine-pw-simps*)

lemma *vcg-of-SPEC*:
assumes $f \equiv do\ \{\ ASSERT\ pre;\ SPEC\ post\ \}$
shows $[\![pre;\ SPEC\ post \leq m]\!] \Longrightarrow f \leq m$
and $[\![pre \Longrightarrow SPEC\ post \leq_n m]\!] \Longrightarrow f \leq_n m$
using *assms*
by (*auto simp*: *pw-le-iff pw-leof-iff refine-pw-simps*)

lemma *vcg-of-SPEC-np*:
assumes $f \equiv SPEC\ post$
shows $SPEC\ post \leq m \Longrightarrow f \leq m$
and $SPEC\ post \leq_n m \Longrightarrow f \leq_n m$
using *assms*
by *auto*

lemma *mk-mop-rl1*:
assumes $\bigwedge x.\ mf\ x \equiv ASSERT\ (P\ x) \gg RETURN\ (f\ x)$
shows $(RETURN\ o\ f,\ mf) \in Id \to \langle Id\rangle nres\text{-}rel$
unfolding *assms*[*abs-def*]
by (*auto intro*!: *nres-relI simp*: *pw-le-iff refine-pw-simps*)

lemma *mk-mop-rl2*:
assumes $\bigwedge x\ y.\ mf\ x\ y \equiv ASSERT\ (P\ x\ y) \gg RETURN\ (f\ x\ y)$
shows $(RETURN\ oo\ f,\ mf) \in Id \to Id \to \langle Id\rangle nres\text{-}rel$
unfolding *assms*[*abs-def*]
by (*auto intro*!: *nres-relI simp*: *pw-le-iff refine-pw-simps*)

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 $(RETURN\ ooo\ f,\ mf) \in Id \to Id \to Id \to \langle Id\rangle nres\text{-}rel$
unfolding *assms*[*abs-def*]
by (*auto intro*!: *nres-relI simp*: *pw-le-iff refine-pw-simps*)

lemma *mk-mop-rl0-np*:
assumes $mf \equiv RETURN\ f$
shows $(RETURN\ f,\ mf) \in \langle Id\rangle nres\text{-}rel$
unfolding *assms*[*abs-def*]
by (*auto intro*!: *nres-relI simp*: *pw-le-iff refine-pw-simps*)

lemma *mk-mop-rl1-np*:
assumes $\bigwedge x.\ mf\ x \equiv RETURN\ (f\ x)$
shows $(RETURN\ o\ f,\ mf) \in Id \to \langle Id\rangle nres\text{-}rel$
unfolding *assms*[*abs-def*]
by (*auto intro*!: *nres-relI simp*: *pw-le-iff refine-pw-simps*)

lemma *mk-mop-rl2-np*:
assumes $\bigwedge x\ y.\ mf\ x\ y \equiv RETURN\ (f\ x\ y)$
shows $(RETURN\ oo\ f,\ mf) \in Id \to Id \to \langle Id\rangle nres\text{-}rel$
unfolding *assms*[*abs-def*]
by (*auto intro*!: *nres-relI simp*: *pw-le-iff refine-pw-simps*)

lemma *mk-mop-rl3-np*:
assumes $\bigwedge x\ y\ z.\ mf\ x\ y\ z \equiv RETURN\ (f\ x\ y\ z)$
shows $(RETURN\ ooo\ f,\ mf) \in Id \to Id \to Id \to \langle Id\rangle nres\text{-}rel$
unfolding *assms*[*abs-def*]
by (*auto intro*!: *nres-relI simp*: *pw-le-iff refine-pw-simps*)

lemma *mk-op-rl0-np*:
assumes $mf \equiv RETURN\ f$
shows $(uncurry0\ mf,\ uncurry0\ (RETURN\ f)) \in unit\text{-}rel \to_f \langle Id\rangle nres\text{-}rel$
apply (*intro frefI nres-relI*)
apply (*auto simp*: *assms*)
done

lemma *mk-op-rl1*:
assumes $\bigwedge x.\ mf\ x \equiv ASSERT\ (P\ x) \gg RETURN\ (f\ x)$
shows $(mf,\ RETURN\ o\ f) \in [P]_f\ Id \to \langle Id\rangle nres\text{-}rel$
apply (*intro frefI nres-relI*)
apply (*auto simp*: *assms*)
done

lemma *mk-op-rl1-np*:
assumes $\bigwedge x.\ mf\ x \equiv RETURN\ (f\ x)$
shows $(mf,\ (RETURN\ o\ f)) \in Id \to_f \langle Id\rangle nres\text{-}rel$
apply (*intro frefI nres-relI*)
apply (*auto simp*: *assms*)
done

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*)) ∈ [*uncurry P*]$_f$ *Id*×$_r$*Id* → ⟨*Id*⟩*nres-rel*
apply (*intro frefI nres-relI*)
apply (*auto simp*: *assms*)
done

lemma *mk-op-rl2-np*:
assumes ⋀*x y*. *mf x y* ≡ *RETURN* (*f x y*)
shows (*uncurry mf*, *uncurry* (*RETURN oo f*)) ∈ *Id*×$_r$*Id* →$_f$ ⟨*Id*⟩*nres-rel*
apply (*intro frefI nres-relI*)
apply (*auto simp*: *assms*)
done

lemma *mk-op-rl3*:
assumes ⋀*x y z*. *mf x y z* ≡ *ASSERT* (*P x y z*) ≫ *RETURN* (*f x y z*)
shows (*uncurry2 mf*, *uncurry2* (*RETURN ooo f*)) ∈ [*uncurry2 P*]$_f$ (*Id*×$_r$*Id*)×$_r$*Id* → ⟨*Id*⟩*nres-rel*
apply (*intro frefI nres-relI*)
apply (*auto simp*: *assms*)
done

lemma *mk-op-rl3-np*:
assumes ⋀*x y z*. *mf x y z* ≡ *RETURN* (*f x y z*)
shows (*uncurry2 mf*, *uncurry2* (*RETURN ooo f*)) ∈ (*Id*×$_r$*Id*)×$_r$*Id* →$_f$ ⟨*Id*⟩*nres-rel*
apply (*intro frefI nres-relI*)
apply (*auto simp*: *assms*)
done

end

1.10 Sepref Tool

theory *Sepref-Tool*
imports *Sepref-Translate Sepref-Definition Sepref-Combinator-Setup Sepref-Intf-Util*
begin

In this theory, we set up the sepref tool.

1.10.1 Sepref Method

lemma *CONS-init*:
assumes *hn-refine* Γ *c* Γ′ *R a*

assumes $\Gamma' \Longrightarrow_t \Gamma c'$
assumes $\bigwedge a\ c.\ \textit{hn-ctxt}\ R\ a\ c \Longrightarrow_t \textit{hn-ctxt}\ Rc\ a\ c$
shows *hn-refine* $\Gamma\ c\ \Gamma c'\ Rc\ a$
apply (*rule hn-refine-cons*)
apply (*rule entt-refl*)
apply (*rule assms*[*unfolded hn-ctxt-def*])+
done

lemma *ID-init*: $[\![\textit{ID}\ a\ a'\ \textit{TYPE}('T);\ \textit{hn-refine}\ \Gamma\ c\ \Gamma'\ R\ a']\!]$
$\Longrightarrow$ *hn-refine* $\Gamma\ c\ \Gamma'\ R\ a$ **by** *simp*

lemma *TRANS-init*: $[\![\ \textit{hn-refine}\ \Gamma\ c\ \Gamma'\ R\ a;\ \textit{CNV}\ c\ c'\ ]\!]$
$\Longrightarrow$ *hn-refine* $\Gamma\ c'\ \Gamma'\ R\ a$
by *simp*

lemma *infer-post-triv*: $P \Longrightarrow_t P$ **by** (*rule entt-refl*)

ML ‹
structure Sepref = struct
structure sepref-preproc-simps = Named-Thms (
val name = @{binding sepref-preproc}
val description = Sepref: Preprocessor simplifications
)

structure sepref-opt-simps = Named-Thms (
val name = @{binding sepref-opt-simps}
val description = Sepref: Post−Translation optimizations, phase 1
)

structure sepref-opt-simps2 = Named-Thms (
val name = @{binding sepref-opt-simps2}
val description = Sepref: Post−Translation optimizations, phase 2
)

fun cons-init-tac ctxt = Sepref-Frame.weaken-post-tac ctxt THEN' resolve-tac ctxt @{thms CONS-init}
fun cons-solve-tac dbg ctxt = let
val dbgSOLVED' = if dbg then I else SOLVED'
in
dbgSOLVED' (
resolve-tac ctxt @{thms infer-post-triv}
ORELSE' Sepref-Translate.side-frame-tac ctxt
)
end

fun preproc-tac ctxt = let
val ctxt = put-simpset HOL-basic-ss ctxt
val ctxt = ctxt addsimps (sepref-preproc-simps.get ctxt)
in

```
  Sepref-Rules.prepare-hfref-synth-tac ctxt THEN'
  Simplifier.simp-tac ctxt
end

fun id-tac ctxt =
  resolve-tac ctxt @{thms ID-init}
  THEN' CONVERSION Thm.eta-conversion
  THEN' DETERM o Id-Op.id-tac Id-Op.Normal ctxt

fun id-init-tac ctxt =
  resolve-tac ctxt @{thms ID-init}
  THEN' CONVERSION Thm.eta-conversion
  THEN' Id-Op.id-tac Id-Op.Init ctxt

fun id-step-tac ctxt =
  Id-Op.id-tac Id-Op.Step ctxt

fun id-solve-tac ctxt =
  Id-Op.id-tac Id-Op.Solve ctxt

(*fun id-param-tac ctxt = CONVERSION (Refine-Util.HOL-concl-conv
  (K (Sepref-Param.id-param-conv ctxt)) ctxt)*)

fun monadify-tac ctxt = Sepref-Monadify.monadify-tac ctxt

(*fun lin-ana-tac ctxt = Sepref-Lin-Ana.lin-ana-tac ctxt*)

fun trans-tac ctxt = Sepref-Translate.trans-tac ctxt

fun opt-tac ctxt = let
  val opt1-ss = put-simpset HOL-basic-ss ctxt
    |> Simplifier.add-simps (sepref-opt-simps.get ctxt)
    |> Simplifier.add-proc @{simproc HOL.let-simp}
    |> Simplifier.add-cong @{thm SP-cong}
    |> Simplifier.add-cong @{thm PR-CONST-cong}

  val unsp-ss = put-simpset HOL-basic-ss ctxt addsimps @{thms SP-def}

  val opt2-ss = put-simpset HOL-basic-ss ctxt
    |> Simplifier.add-simps (sepref-opt-simps2.get ctxt)
    |> Simplifier.add-proc @{simproc HOL.let-simp}

in
  simp-tac opt1-ss THEN' simp-tac unsp-ss THEN'
  simp-tac opt2-ss THEN' simp-tac unsp-ss THEN'
  CONVERSION Thm.eta-conversion THEN'
  resolve-tac ctxt @{thms CNV-I}
end
```

```
    fun sepref-tac dbg ctxt =
      (K Sepref-Constraints.ensure-slot-tac)
      THEN'
      Sepref-Basic.PHASES'
        [
          (preproc,preproc-tac,0),
          (cons-init,cons-init-tac,2),
          (id,id-tac,0),
          (monadify,monadify-tac false,0),
          (opt-init,fn ctxt => resolve-tac ctxt @{thms TRANS-init},1),
          (trans,trans-tac,~1),
          (opt,opt-tac,~1),
          (cons-solve1,cons-solve-tac false,~1),
          (cons-solve2,cons-solve-tac false,~1),
            (constraints,fn ctxt => K (Sepref-Constraints.solve-constraint-slot ctxt
THEN Sepref-Constraints.remove-slot-tac),~1)
        ] (Sepref-Basic.flag-phases-ctrl dbg) ctxt

    val setup = I
      #> sepref-preproc-simps.setup
      #> sepref-opt-simps.setup
      #> sepref-opt-simps2.setup
  end
›

setup Sepref.setup

method-setup sepref = ‹Scan.succeed (fn ctxt =>
  SIMPLE-METHOD (DETERM (SOLVED' (IF-EXGOAL (
      Sepref.sepref-tac false ctxt
    )) 1)))›
  ‹Automatic refinement to Imperative/HOL›

method-setup sepref-dbg-keep = ‹Scan.succeed (fn ctxt => let
    (*val ctxt = Config.put Id-Op.cfg-id-debug true ctxt*)
  in
    SIMPLE-METHOD (IF-EXGOAL (Sepref.sepref-tac true ctxt) 1)
  end)›
  ‹Automatic refinement to Imperative/HOL, debug mode›
```

Default Optimizer Setup

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

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

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

Revert case-pulling done by monadify

lemma *case-prod-return-opt*[*sepref-opt-simps*]:
case-prod (λ*a b*. *return* (*f a b*)) *p* = *return* (*case-prod f p*)
by (*simp split*: *prod.split*)

lemma *case-option-return-opt*[*sepref-opt-simps*]:
case-option (*return fn*) (λ*s*. *return* (*fs s*)) *v* = *return* (*case-option fn fs v*)
by (*simp split*: *option.split*)

lemma *case-list-return*[*sepref-opt-simps*]:
case-list (*return fn*) (λ*x xs*. *return* (*fc x xs*)) *l* = *return* (*case-list fn fc l*)
by (*simp split*: *list.split*)

lemma *if-return*[*sepref-opt-simps*]:
If b (*return t*) (*return e*) = *return* (*If b t e*) **by** *simp*

In some cases, pushing in the returns is more convenient

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*))
by *auto*

1.10.2 Debugging Methods

ML ‹
fun SIMPLE-METHOD-NOPARAM' tac = *Scan.succeed* (*fn ctxt* => *SIMPLE-METHOD'* (*IF-EXGOAL* (*tac ctxt*)))
fun SIMPLE-METHOD-NOPARAM tac = *Scan.succeed* (*fn ctxt* => *SIMPLE-METHOD* (*tac ctxt*))
›
method-setup *sepref-dbg-preproc* = ‹*SIMPLE-METHOD-NOPARAM'* (*fn ctxt* => *K* (*Sepref-Constraints.ensure-slot-tac*) *THEN' Sepref.preproc-tac ctxt*)›
‹*Sepref debug*: *Preprocessing phase*›

method-setup *sepref-dbg-cons-init* = ‹*SIMPLE-METHOD-NOPARAM' Sepref.cons-init-tac*›
‹*Sepref debug*: *Initialize consequence reasoning*›
method-setup *sepref-dbg-id* = ‹*SIMPLE-METHOD-NOPARAM'* (*Sepref.id-tac*)›
‹*Sepref debug*: *Identify operations phase*›
method-setup *sepref-dbg-id-keep* = ‹*SIMPLE-METHOD-NOPARAM'* (*Config.put Id-Op.cfg-id-debug true #> Sepref.id-tac*)›
‹*Sepref debug*: *Identify operations phase. Debug mode, keep intermediate subgoals on failure.*›
method-setup *sepref-dbg-monadify* = ‹*SIMPLE-METHOD-NOPARAM'* (*Sepref.monadify-tac false*)›
‹*Sepref debug*: *Monadify phase*›
method-setup *sepref-dbg-monadify-keep* = ‹*SIMPLE-METHOD-NOPARAM'* (*Sepref.monadify-tac true*)›
‹*Sepref debug*: *Monadify phase*›

method-setup *sepref-dbg-monadify-arity* = ‹*SIMPLE-METHOD-NOPARAM'* (*Sepref-Monadify.arity-tac*)›
‹*Sepref debug*: *Monadify phase*: *Arity phase*›
method-setup *sepref-dbg-monadify-comb* = ‹*SIMPLE-METHOD-NOPARAM'* (*Sepref-Monadify.comb-tac*)›
‹*Sepref debug*: *Monadify phase*: *Comb phase*›
method-setup *sepref-dbg-monadify-check-EVAL* = ‹*SIMPLE-METHOD-NOPARAM'*
(*K* (*CONCL-COND'* (*not o Sepref-Monadify.contains-eval*)))›
‹*Sepref debug*: *Monadify phase*: *check-EVAL phase*›
method-setup *sepref-dbg-monadify-mark-params* = ‹*SIMPLE-METHOD-NOPARAM'*
(*Sepref-Monadify.mark-params-tac*)›
‹*Sepref debug*: *Monadify phase*: *mark-params phase*›
method-setup *sepref-dbg-monadify-dup* = ‹*SIMPLE-METHOD-NOPARAM'* (*Sepref-Monadify.dup-tac*)›
‹*Sepref debug*: *Monadify phase*: *dup phase*›
method-setup *sepref-dbg-monadify-remove-pass* = ‹*SIMPLE-METHOD-NOPARAM'*
(*Sepref-Monadify.remove-pass-tac*)›
‹*Sepref debug*: *Monadify phase*: *remove-pass phase*›

method-setup *sepref-dbg-opt-init* = ‹*SIMPLE-METHOD-NOPARAM'* (*fn ctxt*
=> resolve-tac ctxt @{*thms TRANS-init*})›
‹*Sepref debug*: *Translation phase initialization*›
method-setup *sepref-dbg-trans* = ‹*SIMPLE-METHOD-NOPARAM' Sepref.trans-tac*›
‹*Sepref debug*: *Translation phase*›
method-setup *sepref-dbg-opt* = ‹*SIMPLE-METHOD-NOPARAM'* (*fn ctxt =>*
Sepref.opt-tac ctxt
THEN' CONVERSION Thm.eta-conversion
THEN' TRY o resolve-tac ctxt @{*thms CNV-I*}
)›
‹*Sepref debug*: *Optimization phase*›
method-setup *sepref-dbg-cons-solve* = ‹*SIMPLE-METHOD-NOPARAM'* (*Sepref.cons-solve-tac*
false)›
‹*Sepref debug*: *Solve post−consequences*›
method-setup *sepref-dbg-cons-solve-keep* = ‹*SIMPLE-METHOD-NOPARAM'* (*Sepref.cons-solve-tac*
true)›
‹*Sepref debug*: *Solve post−consequences, keep intermediate results*›

method-setup *sepref-dbg-constraints* = ‹*SIMPLE-METHOD-NOPARAM'* (*fn ctxt*
=> IF-EXGOAL (*K* (
Sepref-Constraints.solve-constraint-slot ctxt
THEN Sepref-Constraints.remove-slot-tac
)))›
‹*Sepref debug*: *Solve accumulated constraints*›

method-setup *sepref-dbg-id-init* = ‹*SIMPLE-METHOD-NOPARAM' Sepref.id-init-tac*›
‹*Sepref debug*: *Initialize operation identification phase*›
method-setup *sepref-dbg-id-step* = ‹*SIMPLE-METHOD-NOPARAM' Sepref.id-step-tac*›
‹*Sepref debug*: *Single step operation identification phase*›
method-setup *sepref-dbg-id-solve* = ‹*SIMPLE-METHOD-NOPARAM' Sepref.id-solve-tac*›

‹*Sepref debug*: *Complete current operation identification goal*›

method-setup *sepref-dbg-trans-keep* = ‹*SIMPLE-METHOD-NOPARAM' Sepref-Translate.trans-keep-tac*›
‹*Sepref debug*: *Translation phase, stop at failed subgoal*›

method-setup *sepref-dbg-trans-step* = ‹*SIMPLE-METHOD-NOPARAM' Sepref-Translate.trans-step-tac*›
‹*Sepref debug*: *Translation step*›

method-setup *sepref-dbg-trans-step-keep* = ‹*SIMPLE-METHOD-NOPARAM' Sepref-Translate.trans-step-ke*
‹*Sepref debug*: *Translation step, keep unsolved subgoals*›

method-setup *sepref-dbg-side* = ‹*SIMPLE-METHOD-NOPARAM' (fn ctxt =>*
REPEAT-ALL-NEW-FWD (Sepref-Translate.side-cond-dispatch-tac false (K no-tac)
ctxt))›
method-setup *sepref-dbg-side-unfold* = ‹*SIMPLE-METHOD-NOPARAM'* (*Sepref-Translate.side-unfold-tac*)
method-setup *sepref-dbg-side-keep* = ‹*SIMPLE-METHOD-NOPARAM' (fn ctxt*
=> REPEAT-ALL-NEW-FWD (Sepref-Translate.side-cond-dispatch-tac true (K
no-tac) ctxt))›

method-setup *sepref-dbg-prepare-frame* = ‹*SIMPLE-METHOD-NOPARAM' Sepref-Frame.prepare-frame-tac*
‹*Sepref debug*: *Prepare frame inference*›

method-setup *sepref-dbg-frame* = ‹*SIMPLE-METHOD-NOPARAM'* (*Sepref-Frame.frame-tac*
(*Sepref-Translate.side-fallback-tac*))›
‹*Sepref debug*: *Frame inference*›

method-setup *sepref-dbg-merge* = ‹*SIMPLE-METHOD-NOPARAM'* (*Sepref-Frame.merge-tac*
(*Sepref-Translate.side-fallback-tac*))›
‹*Sepref debug*: *Frame inference, merge*›

method-setup *sepref-dbg-frame-step* = ‹*SIMPLE-METHOD-NOPARAM'* (*Sepref-Frame.frame-step-tac*
(*Sepref-Translate.side-fallback-tac*) *false*)›
‹*Sepref debug*: *Frame inference, single−step*›

method-setup *sepref-dbg-frame-step-keep* = ‹*SIMPLE-METHOD-NOPARAM'* (*Sepref-Frame.frame-step-tac*
(*Sepref-Translate.side-fallback-tac*) *true*)›
‹*Sepref debug*: *Frame inference, single−step, keep partially solved side conditions*›

1.10.3 Utilities

Manual hfref-proofs

method-setup *sepref-to-hnr* = ‹*SIMPLE-METHOD-NOPARAM' (fn ctxt =>*
Sepref.preproc-tac ctxt THEN' Sepref-Frame.weaken-post-tac ctxt)›
‹*Sepref*: *Convert to hnr−goal and weaken postcondition*›

method-setup *sepref-to-hoare* = ‹
let
fun sepref-to-hoare-tac ctxt = let
val ss = put-simpset HOL-basic-ss ctxt

```
      addsimps @{thms hn-ctxt-def pure-def}

    in
      Sepref.preproc-tac ctxt
      THEN' Sepref-Frame.weaken-post-tac ctxt
      THEN' resolve-tac ctxt @{thms hn-refineI}
      THEN' asm-full-simp-tac ss
    end
  in
    SIMPLE-METHOD-NOPARAM' sepref-to-hoare-tac
  end
› ‹Sepref: Convert to hoare−triple›
```

Copying of Parameters

lemma *fold-COPY*: $x = COPY\ x$ **by** *simp*

sepref-register *COPY*

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

lemma *hnr-pure-COPY*[*sepref-fr-rules*]:
$CONSTRAINT\ is\text{-}pure\ R \Longrightarrow (return, RETURN\ o\ COPY) \in R^k \rightarrow_a R$
by (*sep-auto simp*: *is-pure-conv pure-def intro*!: *hfrefI hn-refineI*)

Short-Circuit Boolean Evaluation

Convert boolean operators to short-circuiting. When applied before monadify, this will generate a short-circuit execution.

lemma *short-circuit-conv*:
$(a \wedge b) \longleftrightarrow (if\ a\ then\ b\ else\ False)$
$(a \vee b) \longleftrightarrow (if\ a\ then\ True\ else\ b)$
$(a \longrightarrow b) \longleftrightarrow (if\ a\ then\ b\ else\ True)$
by *auto*

Eliminating higher-order

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

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

Precision Proofs

We provide a method that tries to extract equalities from an assumption of the form - $\models P1 * \ldots * Pn \wedge_A P1' * \ldots * Pn'$, 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′ ≡ h ⊨ Γ ∗ true ∧A Γ′ ∗ true*
lemma *prec-specI*: *h ⊨ Γ ∧A Γ′ ⟹ prec-spec h Γ Γ′*
unfolding *prec-spec-def*
by (*auto simp*: *mod-and-dist mod-star-trueI*)

lemma *prec-split1-aux*: *A∗B∗true ⟹A A∗true*
apply (*fr-rot 2*, *fr-rot-rhs 1*)
apply (*rule ent-star-mono*)
by *simp-all*

lemma *prec-split2-aux*: *A∗B∗true ⟹A B∗true*
apply (*fr-rot 1*, *fr-rot-rhs 1*)
apply (*rule ent-star-mono*)
by *simp-all*

lemma *prec-spec-splitE*:
assumes *prec-spec h* (*A∗B*) (*C∗D*)
obtains *prec-spec h A C prec-spec h B D*
apply (*thin-tac* ⟦-;-⟧ ⟹ -)
apply (*rule that*)
using *assms*
apply −
unfolding *prec-spec-def*
apply (*erule entailsD*[*rotated*])
apply (*rule ent-conjI*)
apply (*rule ent-conjE1*)
apply (*rule prec-split1-aux*)
apply (*rule ent-conjE2*)
apply (*rule prec-split1-aux*)

apply (*erule entailsD*[*rotated*])
apply (*rule ent-conjI*)
apply (*rule ent-conjE1*)
apply (*rule prec-split2-aux*)
apply (*rule ent-conjE2*)
apply (*rule prec-split2-aux*)
done

lemma *prec-specD*:
assumes *precise R*
assumes *prec-spec h* (*R a p*) (*R a′ p*)
shows *a*=*a′*
using *assms* **unfolding** *precise-def prec-spec-def CONSTRAINT-def* **by** *blast*

ML ‹
fun prec-extract-eqs-tac ctxt = let

```
    fun is-precise thm = case Thm.concl-of thm of
      @{mpat Trueprop (precise -)} => true
    | - => false

    val thms = Sepref-Constraints.get-constraint-rules ctxt
      @ Sepref-Constraints.get-safe-constraint-rules ctxt
    val thms = thms
      |> filter is-precise
    val thms = @{thms snga-prec sngr-prec} @ thms
    val thms = map (fn thm => thm RS @{thm prec-specD}) thms

    val thin-prec-spec-rls = @{thms thin-rl[Pure.of prec-spec a b c for a b c]}

    val tac =
      forward-tac ctxt @{thms prec-specI}
      THEN' REPEAT-ALL-NEW (ematch-tac ctxt @{thms prec-spec-splitE})
      THEN' REPEAT o (dresolve-tac ctxt thms)
      THEN' REPEAT o (eresolve-tac ctxt thin-prec-spec-rls )
  in tac end
›

method-setup prec-extract-eqs = ‹SIMPLE-METHOD-NOPARAM' prec-extract-eqs-tac›
  ‹Extract equalities from - |= - & - assumption, using precision rules›
```

Combinator Rules

```
lemma split-merge: ⟦A ∨_A B ⟹_t X; X ∨_A C ⟹_t D⟧ ⟹ (A ∨_A B ∨_A C ⟹_t
D)
proof −
  assume a1: X ∨_A C ⟹_t D
  assume A ∨_A B ⟹_t X
  then have A ∨_A B ⟹_A D ∗ true
    using a1 by (meson ent-disjI1-direct ent-frame-fwd enttD entt-def-true)
  then show ?thesis
    using a1 by (metis (no-types) Assertions.ent-disjI2 ent-disjE enttD enttI
semigroup.assoc sup.semigroup-axioms)
qed


ML ‹
  fun prep-comb-rule thm = let
    fun mrg t = case Logic.strip-assums-concl t of
      @{mpat Trueprop (- ∨_A - ∨_A - ⟹_t -)} => (@{thm split-merge},true)
    | @{mpat Trueprop (hn-refine - - ?G - -)} => (
        if not (is-Var (head-of G)) then (@{thm hn-refine-cons-post}, true)
        else (asm-rl,false)
      )
    | - => (asm-rl,false)
```

```
    val inst = Thm.prems-of thm |> map mrg
  in
    if exists snd inst then
      prep-comb-rule (thm OF (map fst inst))
    else
      thm |> zero-var-indexes
  end
›

  attribute-setup sepref-prep-comb-rule = ‹Scan.succeed (Thm.rule-attribute []
(K prep-comb-rule))›
    ‹Preprocess combinator rule: Split merge−rules and add missing frame rules›

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*)
by *simp*
  lemma [*sepref-fr-rules*]: (*return o* (λ*x*. *x*), *RETURN o PR-CONST* (*ASSN-ANNOT A*)) ∈ $A^d \rightarrow_a A$
    by *sepref-to-hoare sep-auto*
end

lemma *annotate-assn*: *x* ≡ *ASSN-ANNOT A x* **by** *simp*

2.1.2 Shortcuts

abbreviation (*input*) *nat-assn* ≡ (*id-assn*::*nat* ⇒ -)
abbreviation (*input*) *int-assn* ≡ (*id-assn*::*int* ⇒ -)
abbreviation (*input*) *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*)
by (*auto simp*: *IS-ID-def*)

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*)
by (*auto simp*: *IS-ID-def IS-BELOW-ID-def option-rel-def sum-rel-def list-rel-def*)

lemma *IS-BELOW-ID-fun-rel-aux*: *R1⊇Id* ⟹ *IS-BELOW-ID R2* ⟹ *IS-BELOW-ID* (*R1* → *R2*)
by (*auto simp*: *IS-BELOW-ID-def dest*: *fun-relD*)

corollary *IS-BELOW-ID-fun-rel*[*safe-constraint-rules*]:
IS-ID R1 ⟹ *IS-BELOW-ID R2* ⟹ *IS-BELOW-ID* (*R1* → *R2*)
using *IS-BELOW-ID-fun-rel-aux*[*of Id R2*]
by (*auto simp*: *IS-ID-def*)

lemma *IS-BELOW-ID-list-rel*[*safe-constraint-rules*]:
IS-BELOW-ID R ⟹ *IS-BELOW-ID* (⟨*R*⟩*list-rel*)
unfolding *IS-BELOW-ID-def*
proof *safe*
fix *l l′*
assume *A*: *R⊆Id*
assume (*l*,*l′*)∈⟨*R*⟩*list-rel*
thus *l*=*l′*
apply *induction*
using *A* **by** *auto*
qed

lemma *IS-ID-imp-BELOW-ID*[*constraint-rules*]:
IS-ID R ⟹ *IS-BELOW-ID R*
by (*auto simp*: *IS-ID-def IS-BELOW-ID-def*)

2.1.4 Inverse Relation

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

by (*auto dest*: *fun-relD*)

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

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

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

lemma *inv-list-rel-eq*[*simp*]: $(\langle R \rangle list\text{-}rel)^{-1} = \langle R^{-1} \rangle list\text{-}rel$
unfolding *list-rel-def*
apply *safe*
apply (*subst list.rel-flip*[*symmetric*])
apply (*simp add*: *conversep-iff*[*abs-def*])
apply (*subst list.rel-flip*[*symmetric*])
apply (*simp add*: *conversep-iff*[*abs-def*])
done

lemmas [*constraint-simps*] =
Relation.converse-Id
inv-fun-rel-eq
inv-option-rel-eq
inv-prod-rel-eq
inv-sum-rel-eq
inv-list-rel-eq

2.1.5 Single Valued and Total Relations

definition *IS-LEFT-UNIQUE R* ≡ *single-valued* (R^{-1})
definition *IS-LEFT-TOTAL R* ≡ *Domain R* = *UNIV*
definition *IS-RIGHT-TOTAL R* ≡ *Range R* = *UNIV*
abbreviation (*input*) *IS-RIGHT-UNIQUE* ≡ *single-valued*

lemmas *IS-RIGHT-UNIQUED* = *single-valuedD*
lemma *IS-LEFT-UNIQUED*: ⟦*IS-LEFT-UNIQUE r*; (*y*, *x*) ∈ *r*; (*z*, *x*) ∈ *r*⟧ ⟹ *y* = *z*
by (*auto simp*: *IS-LEFT-UNIQUE-def dest*: *single-valuedD*)

lemma *prop2p*:
IS-LEFT-UNIQUE R = *left-unique* (*rel2p R*)
IS-RIGHT-UNIQUE R = *right-unique* (*rel2p R*)
right-unique (*rel2p* (R^{-1})) = *left-unique* (*rel2p R*)
IS-LEFT-TOTAL R = *left-total* (*rel2p R*)
IS-RIGHT-TOTAL R = *right-total* (*rel2p R*)
by (*auto*
simp: *IS-LEFT-UNIQUE-def left-unique-def single-valued-def*
simp: *right-unique-def*

simp: *IS-LEFT-TOTAL-def left-total-def*
simp: *IS-RIGHT-TOTAL-def right-total-def*
simp: *rel2p-def*
)

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*
by (*auto*
simp: *IS-LEFT-UNIQUE-def left-unique-def single-valued-def*
simp: *right-unique-def bi-unique-alt-def*
simp: *IS-LEFT-TOTAL-def left-total-def*
simp: *IS-RIGHT-TOTAL-def right-total-def*
simp: *p2rel-def*
)

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*)
by (*auto simp*: *IS-LEFT-UNIQUE-def prod-rel-sv sum-rel-sv option-rel-sv list-rel-sv*)

lemma *IS-LEFT-TOTAL-alt*: *IS-LEFT-TOTAL R* $\longleftrightarrow$ ($\forall$ *x*. $\exists$ *y*. (*x*,*y*)$\in$*R*)
by (*auto simp*: *IS-LEFT-TOTAL-def*)

lemma *IS-RIGHT-TOTAL-alt*: *IS-RIGHT-TOTAL R* $\longleftrightarrow$ ($\forall$ *x*. $\exists$ *y*. (*y*,*x*)$\in$*R*)
by (*auto simp*: *IS-RIGHT-TOTAL-def*)

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*)
apply (*auto simp*: *IS-LEFT-TOTAL-alt sum-rel-def option-rel-def list-rel-def*)
apply (*rename-tac x*; *case-tac x*; *auto*)
apply (*rename-tac x*; *case-tac x*; *auto*)
done

lemma [*safe-constraint-rules*]: *IS-LEFT-TOTAL R* $\Longrightarrow$ *IS-LEFT-TOTAL* (⟨*R*⟩*list-rel*)
unfolding *IS-LEFT-TOTAL-alt*
proof *safe*
assume *A*: ∀ *x*.∃ *y*. (*x*,*y*)∈*R*
fix *l*
show ∃ *l'*. (*l*,*l'*)∈⟨*R*⟩*list-rel*
apply (*induction l*)
using *A*
by (*auto simp*: *list-rel-split-right-iff*)
qed

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* (⟨*R1*,*R2*⟩*sum-rel*)
IS-RIGHT-TOTAL R $\Longrightarrow$ *IS-RIGHT-TOTAL* (⟨*R*⟩*option-rel*)
apply (*auto simp*: *IS-RIGHT-TOTAL-alt sum-rel-def option-rel-def*) []
apply (*auto simp*: *IS-RIGHT-TOTAL-alt sum-rel-def option-rel-def*) []
apply (*auto simp*: *IS-RIGHT-TOTAL-alt sum-rel-def option-rel-def*) []
apply (*rename-tac x*; *case-tac x*; *auto*)
apply (*clarsimp simp*: *IS-RIGHT-TOTAL-alt option-rel-def*)
apply (*rename-tac x*; *case-tac x*; *auto*)
done

lemma [*safe-constraint-rules*]: *IS-RIGHT-TOTAL R* $\Longrightarrow$ *IS-RIGHT-TOTAL* (⟨*R*⟩*list-rel*)
unfolding *IS-RIGHT-TOTAL-alt*
proof *safe*
assume *A*: ∀ *x*.∃ *y*. (*y*,*x*)∈*R*
fix *l*
show ∃ *l'*. (*l'*,*l*)∈⟨*R*⟩*list-rel*
apply (*induction l*)
using *A*
by (*auto simp*: *list-rel-split-left-iff*)
qed

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*
by (*auto simp*: *IS-RIGHT-TOTAL-alt IS-LEFT-TOTAL-alt IS-LEFT-UNIQUE-def*)

lemma [*safe-constraint-rules*]:
IS-RIGHT-UNIQUE A $\Longrightarrow$ *IS-RIGHT-TOTAL B* $\Longrightarrow$ *IS-RIGHT-TOTAL* (*A*→*B*)
IS-RIGHT-TOTAL A $\Longrightarrow$ *IS-RIGHT-UNIQUE B* $\Longrightarrow$ *IS-RIGHT-UNIQUE* (*A*→*B*)
IS-LEFT-UNIQUE A $\Longrightarrow$ *IS-LEFT-TOTAL B* $\Longrightarrow$ *IS-LEFT-TOTAL* (*A*→*B*)
IS-LEFT-TOTAL A $\Longrightarrow$ *IS-LEFT-UNIQUE B* $\Longrightarrow$ *IS-LEFT-UNIQUE* (*A*→*B*)
apply (*simp-all add*: *prop2p rel2p*)

apply (*blast intro*!: *transfer-raw*)+
done

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*
by (*auto simp*: *IS-BELOW-ID-def IS-ID-def IS-LEFT-UNIQUE-def IS-RIGHT-TOTAL-def IS-LEFT-TOTAL-def*
intro: *single-valuedI*)

thm *constraint-rules*

Additional Parametricity Lemmas

lemma *param-distinct*[*param*]: $[\![$*IS-LEFT-UNIQUE A*; *IS-RIGHT-UNIQUE A*$]\!] \Longrightarrow$ (*distinct*, *distinct*) $\in \langle A\rangle$*list-rel* $\rightarrow$ *bool-rel*
apply (*fold rel2p-def*)
apply (*simp add*: *rel2p*)
apply (*rule distinct-transfer*)
apply (*simp add*: *p2prop*)
done

lemma *param-Image*[*param*]:
assumes *IS-LEFT-UNIQUE A IS-RIGHT-UNIQUE A*
shows ((‘‘), (‘‘)) $\in \langle A\times_r B\rangle$*set-rel* $\rightarrow \langle A\rangle$*set-rel* $\rightarrow \langle B\rangle$*set-rel*
apply (*clarsimp simp*: *set-rel-def*; *intro conjI*)
apply (*fastforce dest*: *IS-RIGHT-UNIQUED*[*OF assms*(*2*)])
apply (*fastforce dest*: *IS-LEFT-UNIQUED*[*OF assms*(*1*)])
done

lemma *pres-eq-iff-svb*: ((=),(=))$\in K\rightarrow K\rightarrow$*bool-rel* $\longleftrightarrow$ (*single-valued K* $\wedge$ *single-valued* (K^{-1}))
apply (*safe intro*!: *single-valuedI*)
apply (*metis* (*full-types*) *IdD fun-relD1*)
apply (*metis* (*full-types*) *IdD fun-relD1*)
by (*auto dest*: *single-valuedD*)

definition *IS-PRES-EQ R* $\equiv$ ((=), (=))$\in R\rightarrow R\rightarrow$*bool-rel*
lemma [*constraint-rules*]: $[\![$*single-valued R*; *single-valued* (R^{-1})$]\!] \Longrightarrow$ *IS-PRES-EQ R*
by (*simp add*: *pres-eq-iff-svb IS-PRES-EQ-def*)

2.1.6 Bounded Assertions

definition *b-rel R P* $\equiv R \cap$ *UNIV*$\times$*Collect P*
definition *b-assn A P* $\equiv \lambda x\ y.\ A\ x\ y * \uparrow(P\ x)$

lemma *b-assn-pure-conv*[*constraint-simps*]: *b-assn* (*pure R*) *P* = *pure* (*b-rel R P*)
by (*auto intro*!: *ext simp*: *b-rel-def b-assn-def pure-def*)
lemmas [*sepref-import-rewrite*, *sepref-frame-normrel-eqs*, *fcomp-norm-unfold*]
= *b-assn-pure-conv*[*symmetric*]

lemma *b-rel-nesting*[*simp*]:
b-rel (*b-rel R P1*) *P2* = *b-rel R* (λ*x*. *P1 x* ∧ *P2 x*)
by (*auto simp*: *b-rel-def*)
lemma *b-rel-triv*[*simp*]:
b-rel R (λ-. *True*) = *R*
by (*auto simp*: *b-rel-def*)
lemma *b-assn-nesting*[*simp*]:
b-assn (*b-assn A P1*) *P2* = *b-assn A* (λ*x*. *P1 x* ∧ *P2 x*)
by (*auto simp*: *b-assn-def pure-def intro*!: *ext*)
lemma *b-assn-triv*[*simp*]:
b-assn A (λ-. *True*) = *A*
by (*auto simp*: *b-assn-def pure-def intro*!: *ext*)

lemmas [*simp*,*constraint-simps*,*sepref-import-rewrite*, *sepref-frame-normrel-eqs*, *fcomp-norm-unfold*]
= *b-rel-nesting b-assn-nesting*

lemma *b-rel-simp*[*simp*]: (*x*,*y*)∈*b-rel R P* ⟷ (*x*,*y*)∈*R* ∧ *P y*
by (*auto simp*: *b-rel-def*)

lemma *b-assn-simp*[*simp*]: *b-assn A P x y* = *A x y* ∗ ↑(*P x*)
by (*auto simp*: *b-assn-def*)

lemma *b-rel-Range*[*simp*]: *Range* (*b-rel R P*) = *Range R* ∩ *Collect P* **by** *auto*
lemma *b-assn-rdom*[*simp*]: *rdomp* (*b-assn R P*) *x* ⟷ *rdomp R x* ∧ *P x*
by (*auto simp*: *rdomp-def*)

lemma *b-rel-below-id*[*constraint-rules*]:
IS-BELOW-ID R ⟹ *IS-BELOW-ID* (*b-rel R P*)
by (*auto simp*: *IS-BELOW-ID-def*)

lemma *b-rel-left-unique*[*constraint-rules*]:
IS-LEFT-UNIQUE R ⟹ *IS-LEFT-UNIQUE* (*b-rel R P*)
by (*auto simp*: *IS-LEFT-UNIQUE-def single-valued-def*)

lemma *b-rel-right-unique*[*constraint-rules*]:
IS-RIGHT-UNIQUE R ⟹ *IS-RIGHT-UNIQUE* (*b-rel R P*)
by (*auto simp*: *single-valued-def*)

— Registered as safe rule, although may loose information in the odd case that purity depends condition.
lemma *b-assn-is-pure*[*safe-constraint-rules*]:
is-pure A ⟹ *is-pure* (*b-assn A P*)
by (*auto simp*: *is-pure-conv b-assn-pure-conv*)

— Most general form
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*⟧ ⟹ *P' x*
shows *hn-ctxt* (*b-assn A P*) *x y* $\Longrightarrow_t$ *hn-ctxt* (*b-assn A' P'*) *x y*
using *assms*
unfolding *hn-ctxt-def b-assn-def entailst-def entails-def*
by (*fastforce simp*: *vassn-tag-def mod-star-conv*)

— Simplified forms:
lemma *b-assn-subtyping-match-eqA*[*sepref-frame-match-rules*]:
assumes ⟦*vassn-tag* (*hn-ctxt A x y*); *P x*⟧ ⟹ *P' x*
shows *hn-ctxt* (*b-assn A P*) *x y* $\Longrightarrow_t$ *hn-ctxt* (*b-assn A P'*) *x y*
apply (*rule b-assn-subtyping-match*)
subgoal
unfolding *hn-ctxt-def b-assn-def entailst-def entails-def*
by (*fastforce simp*: *vassn-tag-def mod-star-conv*)
subgoal
using *assms* **.**
done

lemma *b-assn-subtyping-match-tR*[*sepref-frame-match-rules*]:
assumes ⟦*P x*⟧ ⟹ *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*
using *assms*
unfolding *hn-ctxt-def b-assn-def entailst-def entails-def*
by (*fastforce simp*: *vassn-tag-def mod-star-conv*)

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*)⟧ ⟹ *P' x*
shows *hn-ctxt A x y* $\Longrightarrow_t$ *hn-ctxt* (*b-assn A' P'*) *x y*
using *assms*
unfolding *hn-ctxt-def b-assn-def entailst-def entails-def*
by (*fastforce simp*: *vassn-tag-def mod-star-conv*)

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*
unfolding *hn-ctxt-def b-assn-def*
by (*sep-auto intro*!: *enttI*)

lemma *b-assn-subtyping-match-eqA-tL*[*sepref-frame-match-rules*]:
assumes ⟦*vassn-tag* (*hn-ctxt A x y*)⟧ ⟹ *P' x*
shows *hn-ctxt A x y* $\Longrightarrow_t$ *hn-ctxt* (*b-assn A P'*) *x y*
using *assms*
unfolding *hn-ctxt-def b-assn-def entailst-def entails-def*
by (*fastforce simp*: *vassn-tag-def mod-star-conv*)

— General form
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* (λx. *P x* $\vee$ *P' x*)) *x y*
using *assms*
unfolding *hn-ctxt-def b-assn-def entailst-def entails-def*
by (*fastforce simp*: *vassn-tag-def*)

— Simplified forms
lemma *b-rel-subtyping-merge-eqA*[*sepref-frame-merge-rules*]:
shows *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* (λx. *P x* $\vee$ *P' x*)) *x y*
apply (*rule b-rel-subtyping-merge*)
by *simp*

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*
using *b-rel-subtyping-merge*[*of A x y A' Am* λ-. *True P'*, *simplified*] *assms* .

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*
using *b-rel-subtyping-merge*[*of A x y A' Am P* λ-. *True*, *simplified*] *assms* .

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*
using *b-rel-subtyping-merge-eqA*[*of A* λ-. *True x y P'*, *simplified*] .

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*
using *b-rel-subtyping-merge-eqA*[*of A P x y* λ-. *True*, *simplified*] .

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* (λx. *P x* $\vee$ *P' x*)) *x y*
by (*sep-auto simp*: *hn-ctxt-def invalid-assn-def entailst-def*)

lemma *b-assn-invalid-merge2*: *hn-invalid* (*b-assn A P*) *x y* $\vee_A$ *hn-invalid A x y*
$\Longrightarrow_t$ *hn-invalid A x y*
by (*sep-auto simp*: *hn-ctxt-def invalid-assn-def entailst-def*)
lemma *b-assn-invalid-merge3*: *hn-invalid A x y* $\vee_A$ *hn-invalid* (*b-assn A P*) *x y*
$\Longrightarrow_t$ *hn-invalid A x y*
by (*sep-auto simp*: *hn-ctxt-def invalid-assn-def entailst-def*)

lemma *b-assn-invalid-merge4*: *hn-invalid* (*b-assn A P*) *x y* $\vee_A$ *hn-ctxt* (*b-assn A*

P') x y
  $\Longrightarrow_t$ *hn-invalid* (*b-assn A* ($\lambda x.$ P x $\vee$ P' x)) x y
  by (*sep-auto simp*: *hn-ctxt-def invalid-assn-def entailst-def*)
lemma *b-assn-invalid-merge5*: *hn-ctxt* (*b-assn A P'*) x y $\vee_A$ *hn-invalid* (*b-assn A P*) x y
  $\Longrightarrow_t$ *hn-invalid* (*b-assn A* ($\lambda x.$ P x $\vee$ P' x)) x y
  by (*sep-auto simp*: *hn-ctxt-def invalid-assn-def entailst-def*)

lemma *b-assn-invalid-merge6*: *hn-invalid* (*b-assn A P*) x y $\vee_A$ *hn-ctxt A x y*
  $\Longrightarrow_t$ *hn-invalid A x y*
  by (*sep-auto simp*: *hn-ctxt-def invalid-assn-def entailst-def*)
lemma *b-assn-invalid-merge7*: *hn-ctxt A x y* $\vee_A$ *hn-invalid* (*b-assn A P*) x y
  $\Longrightarrow_t$ *hn-invalid A x y*
  by (*sep-auto simp*: *hn-ctxt-def invalid-assn-def entailst-def*)

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*
  by (*sep-auto simp*: *hn-ctxt-def invalid-assn-def entailst-def*)
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*
  by (*sep-auto simp*: *hn-ctxt-def invalid-assn-def entailst-def*)

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

abbreviation *nbn-rel* :: *nat* $\Rightarrow$ (*nat* $\times$ *nat*) *set*
  — Natural numbers with upper bound.
  where *nbn-rel n* $\equiv$ *b-rel nat-rel* (λx::*nat*. x<n)

abbreviation *nbn-assn* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *assn*
  — Natural numbers with upper bound.
  where *nbn-assn n* $\equiv$ *b-assn nat-assn* (λx::*nat*. x<n)

2.1.7 Tool Setup

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

lemma [*sepref-relprops-simps*]:
CONSTRAINT (*IS-PURE IS-ID*) *A* ⟹ *CONSTRAINT* (*IS-PURE IS-BELOW-ID*) *A*
CONSTRAINT (*IS-PURE IS-ID*) *A* ⟹ *CONSTRAINT* (*IS-PURE IS-LEFT-TOTAL*) *A*
CONSTRAINT (*IS-PURE IS-ID*) *A* ⟹ *CONSTRAINT* (*IS-PURE IS-RIGHT-TOTAL*) *A*
CONSTRAINT (*IS-PURE IS-BELOW-ID*) *A* ⟹ *CONSTRAINT* (*IS-PURE IS-LEFT-UNIQUE*) *A*
CONSTRAINT (*IS-PURE IS-BELOW-ID*) *A* ⟹ *CONSTRAINT* (*IS-PURE IS-RIGHT-UNIQUE*) *A*
by (*auto*
simp: *IS-ID-def IS-BELOW-ID-def IS-PURE-def IS-LEFT-UNIQUE-def*
simp: *IS-LEFT-TOTAL-def IS-RIGHT-TOTAL-def*
simp: *single-valued-below-Id*)

declare *True-implies-equals*[*sepref-relprops-simps*]

lemma [*sepref-relprops-transform*]: *single-valued* (R^{-1}) = *IS-LEFT-UNIQUE R*
by (*auto simp*: *IS-LEFT-UNIQUE-def*)

2.1.8 HOL Combinators

lemma *hn-if*[*sepref-comb-rules*]:
assumes *P*: Γ ⟹$_t$ 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*)
using *P RT RE IMP*[*OF TERMI*]
unfolding *APP-def PROTECT2-def*
by (*rule hnr-If*)

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

lemma *hn-let*[*sepref-comb-rules*]:

assumes *P*: $\Gamma \Longrightarrow_t \Gamma 1 * \textit{hn-ctxt}\ R\ v\ v'$
assumes *R*: $\bigwedge x\ x'.\ x{=}v \Longrightarrow \textit{hn-refine}\ (\Gamma 1 * \textit{hn-ctxt}\ R\ x\ x')\ (f'\ x')\ (\Gamma'\ x\ x')\ R2\ (f\ x)$
assumes *F*: $\bigwedge x\ x'.\ \Gamma'\ x\ x' \Longrightarrow_t \Gamma 2 * \textit{hn-ctxt}\ R'\ x\ x'$
shows
$\textit{hn-refine}\ \Gamma\ (\textit{Let}\ v'\ f')\ (\Gamma 2 * \textit{hn-ctxt}\ R'\ v\ v')\ R2\ (\textit{Let\$v\$}(\lambda_2 x.\ f\ x))$
apply (*rule hn-refine-cons*[*OF P - F entt-refl*])
apply (*simp*)
apply (*rule R*)
by *simp*

2.1.9 Basic HOL types

lemma *hnr-default*[*sepref-import-param*]: (*default*,*default*)∈*Id* **by** *simp*

lemma *unit-hnr*[*sepref-import-param*]: ((),())∈*unit-rel* **by** *auto*

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* **by** *auto*
lemma *param-max-nat*[*param*,*sepref-import-param*]: (*max*,*max*)∈*nat-rel* → *nat-rel* → *nat-rel* **by** *auto*

lemma *param-min-int*[*param*,*sepref-import-param*]: (*min*,*min*)∈*int-rel* → *int-rel* → *int-rel* **by** *auto*
lemma *param-max-int*[*param*,*sepref-import-param*]: (*max*,*max*)∈*int-rel* → *int-rel* → *int-rel* **by** *auto*

lemma *uminus-hnr*[*sepref-import-param*]: (*uminus*,*uminus*)∈*int-rel* → *int-rel* **by** *auto*

lemma *nat-param*[*param*,*sepref-import-param*]: (*nat*,*nat*) ∈ *int-rel* → *nat-rel* **by** *auto*
lemma *int-param*[*param*,*sepref-import-param*]: (*int*,*int*) ∈ *nat-rel* → *int-rel* **by**

auto

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 $\Longrightarrow$ *precise P2* $\Longrightarrow$ *precise* (*prod-assn P1 P2*)
  apply *rule*
  apply (*clarsimp simp*: *prod-assn-def star-assoc*)
  apply *safe*
  apply (*erule* (*1*) *prec-frame*) **apply** *frame-inference+*
  apply (*erule* (*1*) *prec-frame*) **apply** *frame-inference+*
  done

lemma
  precise P1 $\Longrightarrow$ *precise P2* $\Longrightarrow$ *precise* (*prod-assn P1 P2*) — Original proof
  apply *rule*
  apply (*clarsimp simp*: *prod-assn-def*)
proof (*rule conjI*)
  fix *F F' h as a b a' b' ap bp*
  assume *P1*: *precise P1* **and** *P2*: *precise P2*
  assume *F*: (*h*, *as*) $\models$ *P1 a ap* $*$ *P2 b bp* $*$ *F* $\wedge_A$ *P1 a' ap* $*$ *P2 b' bp* $*$ *F'*

  from *F* **have** (*h*, *as*) $\models$ *P1 a ap* $*$ (*P2 b bp* $*$ *F*) $\wedge_A$ *P1 a' ap* $*$ (*P2 b' bp* $*$ *F'*)
    by (*simp only*: *mult.assoc*)
  with *preciseD*[*OF P1*] **show** *a*=*a'* **.**
  from *F* **have** (*h*, *as*) $\models$ *P2 b bp* $*$ (*P1 a ap* $*$ *F*) $\wedge_A$ *P2 b' bp* $*$ (*P1 a' ap* $*$ *F'*)
    by (*simp only*: *mult.assoc*[**where** *'a*=*assn*] *mult.commute*[**where** *'a*=*assn*] *mult.left-commute*[**where** *'a*=*assn*])
  with *preciseD*[*OF P2*] **show** *b*=*b'* **.**
qed

lemma *intf-of-prod-assn*[*intf-of-assn*]:
  assumes *intf-of-assn A TYPE*('*a*) *intf-of-assn B TYPE*('*b*)
  shows *intf-of-assn* (*prod-assn A B*) *TYPE*('*a* $*$ '*b*)
by *simp*

lemma *pure-prod*[*constraint-rules*]:
  assumes *P1*: *is-pure P1* **and** *P2*: *is-pure P2*
  shows *is-pure* (*prod-assn P1 P2*)
proof −
  from *P1* **obtain** *P1'* **where** *P1'*: $\bigwedge x\ x'$. *P1 x x'* = $\uparrow$(*P1' x x'*)
    using *is-pureE* **by** *blast*
  from *P2* **obtain** *P2'* **where** *P2'*: $\bigwedge x\ x'$. *P2 x x'* = $\uparrow$(*P2' x x'*)
    using *is-pureE* **by** *blast*

```
  show ?thesis proof
    fix x x'
    show prod-assn P1 P2 x x' =
        ↑ (case (x, x') of ((a1, a2), c1, c2) ⇒ P1' a1 c1 ∧ P2' a2 c2)
      unfolding prod-assn-def
      apply (simp add: P1' P2' split: prod.split)
      done
  qed
qed

lemma prod-frame-match[sepref-frame-match-rules]:
  assumes hn-ctxt A (fst x) (fst y) ⟹ₜ hn-ctxt A' (fst x) (fst y)
  assumes hn-ctxt B (snd x) (snd y) ⟹ₜ hn-ctxt B' (snd x) (snd y)
  shows hn-ctxt (prod-assn A B) x y ⟹ₜ hn-ctxt (prod-assn A' B') x y
  apply (cases x; cases y; simp)
  apply (simp add: hn-ctxt-def)
  apply (rule entt-star-mono)
  using assms apply (auto simp: hn-ctxt-def)
  done

lemma prod-frame-merge[sepref-frame-merge-rules]:
  assumes hn-ctxt A (fst x) (fst y) ∨_A hn-ctxt A' (fst x) (fst y) ⟹ₜ hn-ctxt Am
(fst x) (fst y)
  assumes hn-ctxt B (snd x) (snd y) ∨_A hn-ctxt B' (snd x) (snd y) ⟹ₜ hn-ctxt
Bm (snd x) (snd y)
  shows hn-ctxt (prod-assn A B) x y ∨_A hn-ctxt (prod-assn A' B') x y ⟹ₜ hn-ctxt
(prod-assn Am Bm) x y
  by (blast intro: entt-disjE prod-frame-match
    entt-disjD1[OF assms(1)] entt-disjD2[OF assms(1)]
    entt-disjD1[OF assms(2)] entt-disjD2[OF assms(2)])

lemma entt-invalid-prod: hn-invalid (prod-assn A B) p p' ⟹ₜ hn-ctxt (prod-assn
(invalid-assn A) (invalid-assn B)) p p'
    apply (simp add: hn-ctxt-def invalid-assn-def[abs-def])
    apply (rule enttI)
    apply clarsimp
    apply (cases p; cases p'; auto simp: mod-star-conv pure-def)
    done

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 ⟹ hn-ctxt (prod-assn A1 A2) x
y = z
  by (simp add: hn-ctxt-def)

lemma hn-case-prod'[sepref-prep-comb-rule,sepref-comb-rules]:
  assumes FR: Γ⟹ₜhn-ctxt (prod-assn P1 P2) p' p ∗ Γ1
  assumes Pair: ⋀a1 a2 a1' a2'. ⟦p'=(a1',a2')⟧
    ⟹ hn-refine (hn-ctxt P1 a1' a1 ∗ hn-ctxt P2 a2' a2 ∗ Γ1 ∗ 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*])
apply (*rule hn-refine-cons*[*OF - Pair - entt-refl*])
applyS (*simp add*: *hn-ctxt-def*)
applyS *simp*
applyS (*simp add*: *hn-ctxt-def entt-fr-refl entt-fr-drop*)
done

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*])
apply (*rule hn-refine-preI*)
apply (*simp add*: *hn-ctxt-def assn-aci*)
apply (*rule hn-refine-cons*[*OF - R*])
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*
done

lemma *hn-Pair*[*sepref-fr-rules*]: *hn-refine*
(*hn-ctxt P1 x1 x1'* ∗ *hn-ctxt P2 x2 x2'*)
(*return* (*x1'*,*x2'*))
(*hn-invalid P1 x1 x1'* ∗ *hn-invalid P2 x2 x2'*)
(*prod-assn P1 P2*)
(*RETURN*$(*Pair*$*x1*$*x2*))
unfolding *hn-refine-def*
apply (*sep-auto simp*: *hn-ctxt-def prod-assn-def*)
apply (*rule ent-frame-fwd*[*OF invalidate-clone'*[*of P1*]], *frame-inference*)
apply (*rule ent-frame-fwd*[*OF invalidate-clone'*[*of P2*]], *frame-inference*)

apply *sep-auto*
done

lemma *fst-hnr*[*sepref-fr-rules*]: (*return o fst*,*RETURN o fst*) ∈ (*prod-assn A B*)^d →$_a$ *A*
by *sepref-to-hoare sep-auto*
lemma *snd-hnr*[*sepref-fr-rules*]: (*return o snd*,*RETURN o snd*) ∈ (*prod-assn A B*)^d →$_a$ *B*
by *sepref-to-hoare sep-auto*

lemmas [*constraint-simps*] = *prod-assn-pure-conv*
lemmas [*sepref-import-param*] = *param-prod-swap*

lemma *rdomp-prodD*[*dest*!]: *rdomp* (*prod-assn A B*) (*a*,*b*) ⟹ *rdomp A a* ∧ *rdomp B b*
unfolding *rdomp-def prod-assn-def*
by (*sep-auto simp*: *mod-star-conv*)

2.1.11 Option

fun *option-assn* :: ('*a* ⇒ '*c* ⇒ *assn*) ⇒ '*a option* ⇒ '*c option* ⇒ *assn* **where**
option-assn P None None = *emp*
| *option-assn P* (*Some a*) (*Some c*) = *P a c*
| *option-assn* - - - = *false*

lemma *option-assn-simps*[*simp*]:
option-assn P None v' = ↑(*v'*=*None*)
option-assn P v None = ↑(*v*=*None*)
apply (*cases v'*, *simp-all*)
apply (*cases v*, *simp-all*)
done

lemma *option-assn-alt-def*: *option-assn R a b* =
(*case* (*a*,*b*) *of* (*Some x*, *Some y*) ⇒ *R x y*
| (*None*,*None*) ⇒ *emp*
| - ⇒ *false*)
by (*auto split*: *option.split*)

lemma *option-assn-pure-conv*[*constraint-simps*]: *option-assn* (*pure R*) = *pure* (⟨*R*⟩*option-rel*)
apply (*intro ext*)
apply (*rename-tac a c*)
apply (*case-tac* (*pure R*,*a*,*c*) *rule*: *option-assn.cases*)
by (*auto simp*: *pure-def*)

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')
  unfolding hr-comp-def[abs-def]
  apply (intro ext ent-iffI)
  apply solve-entails
  apply (case-tac (R,b,c) rule: option-assn.cases)
  apply clarsimp-all

  apply (sep-auto simp: option-assn-alt-def split: option.splits)
  apply (clarsimp simp: option-assn-alt-def split: option.splits; safe)
  apply (sep-auto split: option.splits)
  apply (intro ent-ex-preI)
  apply (rule ent-ex-postI)
  apply (sep-auto split: option.splits)
  done


lemma option-assn-precise[safe-constraint-rules]:
  assumes precise P
  shows precise (option-assn P)
proof
  fix a a' p h F F'
  assume A: h ⊨ option-assn P a p * F ∧_A option-assn P a' p * F'
  thus a=a' proof (cases (P,a,p) rule: option-assn.cases)
    case (2 - av pv) hence [simp]: a=Some av p=Some pv by simp-all

    from A obtain av' where [simp]: a'=Some av' by (cases a', simp-all)

    from A have h ⊨ P av pv * F ∧_A P av' pv * F' by simp
    with ‹precise P› have av=av' by (rule preciseD)
    thus ?thesis by simp
  qed simp-all
qed

lemma pure-option[safe-constraint-rules]:
  assumes P: is-pure P
  shows is-pure (option-assn P)
proof −
  from P obtain P' where P': ⋀x x'. P x x' = ↑(P' x x')
    using is-pureE by blast

  show ?thesis proof
    fix x x'
    show option-assn P x x' =
        ↑ (case (x, x') of
            (None,None) ⇒ True | (Some v, Some v') ⇒ P' v v' | - ⇒ False
          )
      apply (simp add: P' split: prod.split option.split)
```

```
    done
  qed
qed

lemma hn-ctxt-option: option-assn A x y = z ⟹ hn-ctxt (option-assn A) x y = z
  by (simp add: hn-ctxt-def)

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' * Γ1') 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' * Γ2') R (f2 x)
  assumes MERGE1: Γ1' ∨_A Γ2' ⟹ₜ Γ'
  shows hn-refine Γ (case-option f1' f2' p') (hn-ctxt (option-assn P') p p' * Γ') R
(case-option$f1$(λ2x. f2 x)$p)
    apply (rule hn-refine-cons-pre[OF FR])
    apply1 extract-hnr-invalids
    apply (cases p; cases p'; simp add: option-assn.simps[THEN hn-ctxt-option])
    subgoal
      apply (rule hn-refine-cons[OF - Rn - entt-refl]; assumption?)
      applyS (simp add: hn-ctxt-def)

      apply (subst mult.commute, rule entt-fr-drop)
      apply (rule entt-trans[OF - MERGE1])
      apply (simp add: ent-disjI1' ent-disjI2')
    done

    subgoal
      apply (rule hn-refine-cons[OF - Rs - entt-refl]; assumption?)
      applyS (simp add: hn-ctxt-def)
      apply (rule entt-star-mono)
      apply1 (rule entt-fr-drop)
      applyS (simp add: hn-ctxt-def)
      apply1 (rule entt-trans[OF - MERGE1])
      applyS (simp add: hn-ctxt-def)
    done
    done

lemma hn-None[sepref-fr-rules]:
  hn-refine emp (return None) emp (option-assn P) (RETURN$None)
  by rule sep-auto

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)*))
by *rule* (*sep-auto simp*: *hn-ctxt-def invalidate-clone'*)

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*: ⋀*va va' vb vb'*. *hn-refine*
(*hn-ctxt P va va'* ∗ *hn-ctxt P vb vb'* ∗ Γ*1*)
(*eq' va' vb'*)
(Γ' *va va' vb vb'*)
bool-assn
(*RETURN$*((=) $*va*$*vb*))
assumes *F2*:
⋀*va va' vb vb'*.
Γ' *va va' vb vb'* ⟹$_t$ *hn-ctxt P va va'* ∗ *hn-ctxt P vb vb'* ∗ Γ*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'* ∗ Γ*1*)
bool-assn
(*RETURN$*((=) $*a*$*b*))
apply (*rule hn-refine-cons-pre*[*OF F1*])
unfolding *imp-option-eq-def*
apply *rule*
apply (*simp split*: *option.split add*: *hn-ctxt-def*, *intro impI conjI*)

apply (*sep-auto split*: *option.split simp*: *hn-ctxt-def pure-def*)
apply (*cases a*, (*sep-auto split*: *option.split simp*: *hn-ctxt-def pure-def*)+)[]
apply (*cases a*, (*sep-auto split*: *option.split simp*: *hn-ctxt-def pure-def*)+)[]
apply (*cases b*, (*sep-auto split*: *option.split simp*: *hn-ctxt-def pure-def*)+)[]
apply (*rule cons-post-rule*)
apply (*rule hn-refineD*[*OF EQ*[*unfolded hn-ctxt-def*]])
apply *simp*
apply (*rule ent-frame-fwd*[*OF F2*[*THEN enttD*,*unfolded hn-ctxt-def*]])
apply (*fr-rot 2*)
apply (*fr-rot-rhs 1*)
apply (*rule fr-refl*)
apply (*rule ent-refl*)
apply (*sep-auto simp*: *pure-def*)
done

lemma [*pat-rules*]:
(=) aNone ≡ *is-None*$a
(=) $None$a ≡ *is-None*$a
apply (*rule eq-reflection*, *simp split*: *option.split*)+
done

lemma *hn-is-None*[*sepref-fr-rules*]: *hn-refine*
(*hn-ctxt* (*option-assn P*) *a a'*)
(*return* (*is-None a'*))
(*hn-ctxt* (*option-assn P*) *a a'*)
bool-assn
(*RETURN*$(*is-None*$*a*))
apply *rule*
apply (*sep-auto split*: *option.split simp*: *hn-ctxt-def pure-def*)
done

lemma (**in** −) *sepref-the-complete*[*sepref-fr-rules*]:
assumes *x*≠*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*))
using *assms*
apply (*cases x*)
apply *simp*
apply (*cases xi*)
apply (*simp add*: *hn-ctxt-def*)
apply *rule*
apply (*sep-auto simp*: *hn-ctxt-def invalidate-clone' vassn-tagI invalid-assn-const*)
done

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*))
using *assms*
apply (*clarsimp simp*: *IS-PURE-def IS-ID-def hn-ctxt-def is-pure-conv*)
apply (*cases x*)
apply *simp*
apply (*cases xi*)
apply (*simp add*: *hn-ctxt-def invalid-assn-def*)

apply *rule* **apply** (*sep-auto simp*: *pure-def*)
apply *rule* **apply** (*sep-auto*)
apply (*simp add*: *option-assn-pure-conv*)
apply *rule* **apply** (*sep-auto simp*: *pure-def invalid-assn-def*)
done

2.1.12 Lists

fun *list-assn* :: ('*a* ⇒ '*c* ⇒ *assn*) ⇒ '*a list* ⇒ '*c list* ⇒ *assn* **where**
list-assn P [] [] = *emp*
| *list-assn P* (*a*#*as*) (*c*#*cs*) = *P a c* ∗ *list-assn P as cs*
| *list-assn* - - - = *false*

lemma *list-assn-aux-simps*[*simp*]:
list-assn P [] *l'* = (↑(*l'*=[]))
list-assn P l [] = (↑(*l*=[]))
unfolding *hn-ctxt-def*
apply (*cases l'*)
apply *simp*
apply *simp*
apply (*cases l*)
apply *simp*
apply *simp*
done

lemma *list-assn-aux-append*[*simp*]:
length l1=*length l1'* ⟹
list-assn P (*l1*@*l2*) (*l1'*@*l2'*)
= *list-assn P l1 l1'* ∗ *list-assn P l2 l2'*
apply (*induct rule*: *list-induct2*)
apply *simp*
apply (*simp add*: *star-assoc*)
done

lemma *list-assn-aux-ineq-len*: *length l* ≠ *length li* ⟹ *list-assn A l li* = *false*
proof (*induction l arbitrary*: *li*)
case (*Cons x l li*) **thus** *?case* **by** (*cases li*; *auto*)
qed *simp*

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'*
apply (*cases length l1* = *length l1'*)
apply (*erule list-assn-aux-append*)
apply (*simp add*: *list-assn-aux-ineq-len assms*)
done

lemma *list-assn-pure-conv*[*constraint-simps*]: *list-assn* (*pure R*) = *pure* (⟨*R*⟩*list-rel*)

```
proof (intro ext)
  fix l li
  show list-assn (pure R) l li = pure (⟨R⟩list-rel) l li
    apply (induction pure R l li rule: list-assn.induct)
    by (auto simp: pure-def)
qed

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


lemma list-assn-simps[simp]:
  hn-ctxt (list-assn P) [] l' = (↑(l'=[]))
  hn-ctxt (list-assn P) l [] = (↑(l=[]))
  hn-ctxt (list-assn P) [] [] = emp
  hn-ctxt (list-assn P) (a#as) (c#cs) = hn-ctxt P a c * hn-ctxt (list-assn P) as cs
  hn-ctxt (list-assn P) (a#as) [] = false
  hn-ctxt (list-assn P) [] (c#cs) = false
  unfolding hn-ctxt-def
  apply (cases l')
  apply simp
  apply simp
  apply (cases l)
  apply simp
  apply simp
  apply simp-all
  done

lemma list-assn-precise[constraint-rules]: precise P ⟹ precise (list-assn P)
proof
  fix l1 l2 l h F1 F2
  assume P: precise P
  assume h⊨list-assn P l1 l * F1 ∧_A list-assn P l2 l * F2
  thus l1=l2
  proof (induct l arbitrary: l1 l2 F1 F2)
    case Nil thus ?case by simp
  next
    case (Cons a ls)
    from Cons obtain a1 ls1 where [simp]: l1=a1#ls1
      by (cases l1, simp)
    from Cons obtain a2 ls2 where [simp]: l2=a2#ls2
      by (cases l2, simp)

    from Cons.prems have M:
      h ⊨ P a1 a * list-assn P ls1 ls * F1
        ∧_A P a2 a * list-assn P ls2 ls * F2 by simp
    have a1=a2
      apply (rule preciseD[OF P, where a=a1 and a'=a2 and p=a
        and F= list-assn P ls1 ls * F1
```

```
      and F'=list-assn P ls2 ls * F2
      ])
    using M
    by (simp add: star-assoc)

  moreover have ls1=ls2
   apply (rule Cons.hyps[where ?F1.0=P a1 a * F1 and ?F2.0=P a2 a * F2])
    using M
    by (simp only: star-aci)
  ultimately show ?case by simp
 qed
qed
lemma list-assn-pure[constraint-rules]:
 assumes P: is-pure P
 shows is-pure (list-assn P)
proof -
 from P obtain P' where P-eq: ⋀x x'. P x x' = ↑(P' x x')
  by (rule is-pureE) blast

 {
  fix l l'
  have list-assn P l l' = ↑(list-all2 P' l l')
   by (induct P≡P l l' rule: list-assn.induct)
     (simp-all add: P-eq)
 } thus ?thesis by rule
qed

lemma list-assn-mono:
 ⟦⋀x x'. P x x'⟹_A P' x x'⟧ ⟹ list-assn P l l' ⟹_A list-assn P' l l'
 unfolding hn-ctxt-def
 apply (induct P l l' rule: list-assn.induct)
 by (auto intro: ent-star-mono)

lemma list-assn-monot:
 ⟦⋀x x'. P x x'⟹_t P' x x'⟧ ⟹ list-assn P l l' ⟹_t list-assn P' l l'
 unfolding hn-ctxt-def
 apply (induct P l l' rule: list-assn.induct)
 by (auto intro: entt-star-mono)

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'
 unfolding hn-ctxt-def
 by (induct A l l' rule: list-assn.induct) (simp-all add: entt-star-mono)

lemma list-merge-cong[sepref-frame-merge-rules]:
 assumes ⋀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'
 apply (blast intro: entt-disjE list-match-cong entt-disjD1[OF assms] entt-disjD2[OF
assms])
 done

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
 by (fastforce simp: invalid-assn-def intro!: enttI simp: mod-star-conv)

lemma entt-invalid-list: hn-invalid (list-assn A) l l' ⟹t hn-ctxt (list-assn (invalid-assn
A)) l l'
 apply (induct A l l' rule: list-assn.induct)
 applyS simp

 subgoal
  apply1 (simp add: hn-ctxt-def cong del: invalid-assn-cong)
  apply1 (rule entt-trans[OF invalid-list-split])
  apply (rule entt-star-mono)
   applyS simp

   apply (rule entt-trans)
    applyS assumption
    applyS simp
  done

 applyS (simp add: hn-ctxt-def invalid-assn-def)
 applyS (simp add: hn-ctxt-def invalid-assn-def)
 done

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


lemma list-assn-comp[fcomp-norm-unfold]: hr-comp (list-assn A) (⟨B⟩list-rel) =
list-assn (hr-comp A B)
proof (intro ext)
 { fix x l y m
  have hr-comp (list-assn A) (⟨B⟩list-rel) (x # l) (y # m) =
    hr-comp A B x y ∗ hr-comp (list-assn A) (⟨B⟩list-rel) l m
    by (sep-auto
      simp: hr-comp-def list-rel-split-left-iff
      intro!: ent-ex-preI ent-iffI)
 } note aux = this

 fix l li
 show hr-comp (list-assn A) (⟨B⟩list-rel) l li = list-assn (hr-comp A B) l li
  apply (induction l arbitrary: li; case-tac li; intro ent-iffI)
  apply (sep-auto simp: hr-comp-def; fail)+
  by (simp-all add: aux)
```

qed

lemma *hn-ctxt-eq*: $A\ x\ y = z \Longrightarrow$ *hn-ctxt* $A\ x\ y = z$ **by** (*simp add*: *hn-ctxt-def*)

lemmas *hn-ctxt-list* = *hn-ctxt-eq*[*of list-assn* A **for** A]

lemma *hn-case-list*[*sepref-prep-comb-rule*, *sepref-comb-rules*]:
fixes $p\ p'\ P$
defines [*simp*]: $INVE \equiv$ *hn-invalid* (*list-assn* P) $p\ p'$
assumes *FR*: $\Gamma \Longrightarrow_t$ *hn-ctxt* (*list-assn* P) $p\ p' * F$
assumes *Rn*: $p{=}[] \Longrightarrow$ *hn-refine* (*hn-ctxt* (*list-assn* P) $p\ p' * F$) $f1'$ (*hn-ctxt XX1* $p\ p' * \Gamma 1'$) R *f1*
assumes *Rs*: $\bigwedge x\ l\ x'\ l'.\ [\![\ p{=}x\#l;\ p'{=}x'\#l'\ ]\!] \Longrightarrow$
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* $p\ p' * \Gamma 2'$) R ($f2\ x\ l$)
assumes *MERGE1*[*unfolded hn-ctxt-def*]: $\bigwedge x\ x'$. *hn-ctxt* $P1'\ x\ x' \vee_A$ *hn-ctxt* $P2'\ x\ x' \Longrightarrow_t$ *hn-ctxt* $P'\ x\ x'$
assumes *MERGE2*: $\Gamma 1' \vee_A \Gamma 2' \Longrightarrow_t \Gamma'$
shows *hn-refine* Γ (*case-list* $f1'\ f2'\ p'$) (*hn-ctxt* (*list-assn* P') $p\ p' * \Gamma'$) R (*case-list*$f1$$(\lambda_2 x\ l.\ f2\ x\ l)$$p)
apply (*rule hn-refine-cons-pre*[*OF FR*])
apply1 *extract-hnr-invalids*
apply (*cases p*; *cases p'*; *simp add*: *list-assn.simps*[*THEN hn-ctxt-list*])
subgoal
apply (*rule hn-refine-cons*[*OF - Rn - entt-refl*]; *assumption?*)
applyS (*simp add*: *hn-ctxt-def*)

apply (*subst mult.commute*, *rule entt-fr-drop*)
apply (*rule entt-trans*[*OF - MERGE2*])
apply (*simp add*: *ent-disjI1' ent-disjI2'*)
done

subgoal
apply (*rule hn-refine-cons*[*OF - Rs - entt-refl*]; *assumption?*)
applyS (*simp add*: *hn-ctxt-def*)
apply (*rule entt-star-mono*)
apply1 (*rule entt-fr-drop*)
apply (*rule entt-star-mono*)

apply1 (*simp add*: *hn-ctxt-def*)
apply1 (*rule entt-trans*[*OF - MERGE1*])
applyS (*simp*)

apply1 (*simp add*: *hn-ctxt-def*)
apply (*rule list-assn-monot*)
apply1 (*rule entt-trans*[*OF - MERGE1*])
applyS (*simp*)

apply1 (*rule entt-trans*[*OF - MERGE2*])

applyS (*simp*)
done
done

lemma *hn-Nil*[*sepref-fr-rules*]:
hn-refine emp (*return* []) *emp* (*list-assn P*) (*RETURN*$[])
unfolding *hn-refine-def*
by *sep-auto*

lemma *hn-Cons*[*sepref-fr-rules*]: *hn-refine* (*hn-ctxt P x x'* ∗ *hn-ctxt* (*list-assn P*) *xs xs'*)
(*return* (*x'*#*xs'*)) (*hn-invalid P x x'* ∗ *hn-invalid* (*list-assn P*) *xs xs'*) (*list-assn P*)
(*RETURN*$((#) $*x*$*xs*))
unfolding *hn-refine-def*
apply (*sep-auto simp*: *hn-ctxt-def*)
apply (*rule ent-frame-fwd*[*OF invalidate-clone'*[*of P*]], *frame-inference*)
apply (*rule ent-frame-fwd*[*OF invalidate-clone'*[*of list-assn P*]], *frame-inference*)
apply *solve-entails*
done

lemma *list-assn-aux-len*:
list-assn P l l' = *list-assn P l l'* ∗ ↑(*length l* = *length l'*)
apply (*induct P*≡*P l l' rule*: *list-assn.induct*)
apply *simp-all*
subgoal for *a as c cs*
by (*erule-tac t*=*list-assn P as cs* **in** *subst*[*OF sym*]) *simp*
done

lemma *list-assn-aux-eqlen-simp*:
vassn-tag (*list-assn P l l'*) ⟹ *length l'* = *length l*
h ⊨ (*list-assn P l l'*) ⟹ *length l'* = *length l*
apply (*subst* (*asm*) *list-assn-aux-len*; *auto simp*: *vassn-tag-def*)+
done

lemma *hn-append*[*sepref-fr-rules*]: *hn-refine* (*hn-ctxt* (*list-assn P*) *l1 l1'* ∗ *hn-ctxt* (*list-assn P*) *l2 l2'*)
(*return* (*l1'*@*l2'*)) (*hn-invalid* (*list-assn P*) *l1 l1'* ∗ *hn-invalid* (*list-assn P*) *l2 l2'*) (*list-assn P*)
(*RETURN*$((@) $*l1*$*l2*))
apply *rule*
apply (*sep-auto simp*: *hn-ctxt-def*)
apply (*subst list-assn-aux-len*)
apply (*sep-auto*)
apply (*rule ent-frame-fwd*[*OF invalidate-clone'*[*of list-assn P*]], *frame-inference*)
apply (*rule ent-frame-fwd*[*OF invalidate-clone'*[*of list-assn P*]], *frame-inference*)
apply *solve-entails*
done

lemma *list-assn-aux-cons-conv1*:
list-assn R (a#l) m = (∃ A b m'. R a b ∗ list-assn R l m' ∗ ↑(m=b#m'))
apply (*cases m*)
apply *sep-auto*
apply (*sep-auto intro*!: *ent-iffI*)
done
lemma *list-assn-aux-cons-conv2*:
list-assn R l (b#m) = (∃ A a l'. R a b ∗ list-assn R l' m ∗ ↑(l=a#l'))
apply (*cases l*)
apply *sep-auto*
apply (*sep-auto intro*!: *ent-iffI*)
done
lemmas *list-assn-aux-cons-conv = list-assn-aux-cons-conv1 list-assn-aux-cons-conv2*

lemma *list-assn-aux-append-conv1*:
list-assn R (l1@l2) m = (∃ Am1 m2. list-assn R l1 m1 ∗ list-assn R l2 m2 ∗ ↑(m=m1@m2))
apply (*induction l1 arbitrary*: *m*)
apply (*sep-auto intro*!: *ent-iffI*)
apply (*sep-auto intro*!: *ent-iffI simp*: *list-assn-aux-cons-conv*)
done
lemma *list-assn-aux-append-conv2*:
list-assn R l (m1@m2) = (∃ Al1 l2. list-assn R l1 m1 ∗ list-assn R l2 m2 ∗ ↑(l=l1@l2))
apply (*induction m1 arbitrary*: *l*)
apply (*sep-auto intro*!: *ent-iffI*)
apply (*sep-auto intro*!: *ent-iffI simp*: *list-assn-aux-cons-conv*)
done
lemmas *list-assn-aux-append-conv = list-assn-aux-append-conv1 list-assn-aux-append-conv2*

declare *param-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)*
apply (*auto simp*: *is-pure-iff-pure-assn*)
apply (*rename-tac x x'*)
apply (*case-tac x*; *case-tac x'*; *simp add*: *pure-def*)
done

lemma *sum-assn-id*[*simp*]: *sum-assn id-assn id-assn* = *id-assn*
apply (*intro ext*)
subgoal for *x y* **by** (*cases x*; *cases y*; *simp add*: *pure-def*)
done

lemma *sum-assn-pure-conv*[*simp*]: *sum-assn* (*pure A*) (*pure B*) = *pure* (⟨*A*,*B*⟩*sum-rel*)
apply (*intro ext*)
subgoal for *a b* **by** (*cases a*; *cases b*; *auto simp*: *pure-def*)
done

lemma *sum-match-cong*[*sepref-frame-match-rules*]:
⟦
⋀*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*′
by (*cases e*; *cases e*′; *simp add*: *hn-ctxt-def entt-star-mono*)

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*′
apply (*rule entt-disjE*)
apply (*rule sum-match-cong*)
apply (*rule entt-disjD1*[*OF assms*(*1*)]; *simp*)
apply (*rule entt-disjD1*[*OF assms*(*2*)]; *simp*)

apply (*rule sum-match-cong*)
apply (*rule entt-disjD2*[*OF assms*(*1*)]; *simp*)
apply (*rule entt-disjD2*[*OF assms*(*2*)]; *simp*)
done

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*′
apply (*simp add*: *hn-ctxt-def invalid-assn-def*[*abs-def*])
apply (*rule enttI*)
apply *clarsimp*
apply (*cases e*; *cases e*′; *auto simp*: *mod-star-conv pure-def*)
done

lemmas *invalid-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*

by *sepref-to-hoare sep-auto*
lemma [*sepref-fr-rules*]: (*return o Inr*,*RETURN o Inr*) $\in B^d \to_a$ *sum-assn A B*
by *sepref-to-hoare sep-auto*

sepref-register *case-sum*

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

lemma [*sepref-monadify-arity*]: *case-sum* $\equiv \lambda_2 f1\ f2\ x.$ *SP case-sum*\$($\lambda_2 x.$ *f1*\$*x*)\$($\lambda_2 x.$ *f2*\$*x*)\$*x*
by *simp*

This determines an evaluation order for the first-order operands

lemma [*sepref-monadify-comb*]: *case-sum*\$*f1*\$*f2*\$*x* $\equiv$ ($\gg\!=$) \$(*EVAL*\$*x*)\$($\lambda_2 x.$ *SP case-sum*\$*f1*\$*f2*\$*x*) **by** *simp*

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*)
$\equiv$ ($\gg\!=$) \$(*EVAL*\$*x*)\$($\lambda_2 x.$ *SP case-sum*\$($\lambda_2 x.$ *EVAL* \$ *f1 x*)\$($\lambda_2 x.$ *EVAL* \$ *f2 x*)\$*x*)
apply (*rule eq-reflection*)
by (*simp split*: *sum.splits*)

Auxiliary lemma, to lift simp-rule over *hn-ctxt*

lemma *sum-assn-ctxt*: *sum-assn A B x y* $= z \Longrightarrow$ *hn-ctxt* (*sum-assn A B*) *x y* $=$ *z*
by (*simp add*: *hn-ctxt-def*)

The cases lemma first extracts the refinement for the datatype from the precondition. Next, it generate proof obligations to refine the functions for every case. Finally the postconditions of the refinement are merged.

Note that we handle the destructed values separately, to allow reconstruction of the original datatype after the case-expression.

Moreover, we provide (invalidated) versions of the original compound value to the cases, which allows access to pure compound values from inside the case.

lemma *sum-cases-hnr*:
fixes *A B e e′*
defines [*simp*]: *INVe* $\equiv$ *hn-invalid* (*sum-assn A B*) *e e′*
assumes *FR*: $\Gamma \Longrightarrow_t$ *hn-ctxt* (*sum-assn A B*) *e e′* $*$ *F*
assumes *E1*: $\bigwedge$*x1 x1a*. $[\![e =$ *Inl x1*; *e′* $=$ *Inl x1a*$]\!] \Longrightarrow$ *hn-refine* (*hn-ctxt A x1 x1a* $*$ *INVe* $*$ *F*) (*f1′ x1a*) (*hn-ctxt A′ x1 x1a* $*$ *hn-ctxt XX1 e e′* $*$ $\Gamma 1'$) *R* (*f1 x1*)
assumes *E2*: $\bigwedge$*x2 x2a*. $[\![e =$ *Inr x2*; *e′* $=$ *Inr x2a*$]\!] \Longrightarrow$ *hn-refine* (*hn-ctxt B x2 x2a* $*$ *INVe* $*$ *F*) (*f2′ x2a*) (*hn-ctxt B′ x2 x2a* $*$ *hn-ctxt XX2 e e′* $*$ $\Gamma 2'$) *R* (*f2 x2*)
assumes *MERGE*[*unfolded hn-ctxt-def*]: $\Gamma 1' \vee_A \Gamma 2' \Longrightarrow_t \Gamma'$
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*)

```
  apply (rule hn-refine-cons-pre[OF FR])
  apply1 extract-hnr-invalids
  apply (cases e; cases e'; simp add: sum-assn.simps[THEN sum-assn-ctxt])
  subgoal
    apply (rule hn-refine-cons[OF - E1 - entt-refl]; assumption?)
    applyS (simp add: hn-ctxt-def) — Match precondition for case, get enum-assn
      from assumption generated by extract-hnr-invalids
     apply (rule entt-star-mono) — Split postcondition into pairs for compounds
      and frame, drop hn-ctxt XX
    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')
  done
  subgoal
    apply (rule hn-refine-cons[OF - E2 - entt-refl]; assumption?)
    applyS (simp add: hn-ctxt-def)
    apply (rule entt-star-mono)
    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')
  done
done
```

After some more preprocessing (adding extra frame-rules for non-atomic postconditions, and splitting the merge-terms into binary merges), this rule can be registered

```
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
  apply sepref-to-hoare
  subgoal for a b by (cases a; cases b; sep-auto)
  done

lemma projl-hnr[sepref-fr-rules]: (return o projl,RETURN o projl) ∈ [isl]_a (sum-assn
A B)^d → A
  apply sepref-to-hoare
  subgoal for a b by (cases a; cases b; sep-auto)
  done

lemma projr-hnr[sepref-fr-rules]: (return o projr,RETURN o projr) ∈ [Not o isl]_a
(sum-assn A B)^d → B
  apply sepref-to-hoare
  subgoal for a b by (cases a; cases b; sep-auto)
  done
```

2.1.14 String Literals

sepref-register *PR-CONST String.empty-literal*

lemma *empty-literal-hnr* [*sepref-import-param*]:
(*String.empty-literal*, *PR-CONST String.empty-literal*) ∈ *Id*
by *simp*

lemma *empty-literal-pat* [*def-pat-rules*]:
String.empty-literal ≡ *UNPROTECT String.empty-literal*
by *simp*

context
fixes *b0 b1 b2 b3 b4 b5 b6* :: *bool*
and *s* :: *String.literal*
begin

sepref-register *PR-CONST* (*String.Literal b0 b1 b2 b3 b4 b5 b6 s*)

lemma *Literal-hnr* [*sepref-import-param*]:
(*String.Literal b0 b1 b2 b3 b4 b5 b6 s*,
PR-CONST (*String.Literal b0 b1 b2 b3 b4 b5 b6 s*)) ∈ *Id*
by *simp*

end

lemma *Literal-pat* [*def-pat-rules*]:
String.Literal $ *b0* $ *b1* $ *b2* $ *b3* $ *b4* $ *b5* $ *b6* $ *s* ≡
UNPROTECT (*String.Literal* $ *b0* $ *b1* $ *b2* $ *b3* $ *b4* $ *b5* $ *b6* $ *s*)
by *simp*

end

2.2 Setup for Foreach Combinator

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 σ0 ≡ do {
  ASSERT (finite S);
  xs0 ← it-to-sorted-list R S;
  (-,σ) ← RECT (λW (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,σ0);
  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
 unfolding FOREACHoci-def monadic-FOREACH-def WHILEIT-def WHILEI-body-def
  unfolding it-to-sorted-list-def FOREACH-cond-def FOREACH-inv-def
  apply simp
  apply (fo-rule arg-cong[THEN cong] | rule refl ext)+
  apply (simp split: prod.split)
  apply (rule refl)+
  done
```

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

```
definition monadic-nfoldli l c f s ≡ 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
      }
  )
  apply (subst monadic-nfoldli-def)
  apply (subst RECT-unfold)
  apply (tagged-solver)
  apply (subst monadic-nfoldli-def[symmetric])
```

```
  apply simp
  done

lemma monadic-nfoldli-simp[simp]:
  monadic-nfoldli [] c f s = RETURN s
  monadic-nfoldli (x#ls) c f s = do {
    b←c s;
    if b then f x s ≫= monadic-nfoldli ls c f else RETURN s
  }
  apply (subst monadic-nfoldli-eq, simp)
  apply (subst monadic-nfoldli-eq, simp)
  done

lemma nfoldli-to-monadic:
  nfoldli l c f = monadic-nfoldli l (λx. RETURN (c x)) f
  apply (induct l)
  apply auto
  done

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

lemma nfoldli-alt-eq:
  nfoldli-alt 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}
  )
  apply (subst nfoldli-alt-def)
  apply (subst RECT-unfold)
  apply (tagged-solver)
  apply (subst nfoldli-alt-def[symmetric])
  apply simp
  done

lemma nfoldli-alt-simp[simp]:
  nfoldli-alt [] c f s = RETURN s
  nfoldli-alt (x#ls) c f s = do {
    let b = c s;
    if b then f x s ≫= nfoldli-alt ls c f else RETURN s
  }
  apply (subst nfoldli-alt-eq, simp)
  apply (subst nfoldli-alt-eq, simp)
  done
```

lemma *nfoldli-alt*:
(*nfoldli*::'*a list* ⇒ ('*b* ⇒ *bool*) ⇒ ('*a* ⇒ '*b* ⇒ '*b nres*) ⇒ '*b* ⇒ '*b nres*)
= *nfoldli-alt*
proof (*intro ext*)
fix *l*::'*a list* **and** *c*::'*b* ⇒ *bool* **and** *f*::'*a* ⇒ '*b* ⇒ '*b nres* **and** *s* :: '*b*
have *nfoldli l c f* = *nfoldli-alt l c f*
by (*induct l*) *auto*
thus *nfoldli l c f s* = *nfoldli-alt l c f s* **by** *simp*
qed

lemma *monadic-nfoldli-rec*:
monadic-nfoldli x' c f σ
≤⇓*Id* (*REC*$_T$
(λ*W* (*xs*, *σ*).
ASSERT (*FOREACH-inv xs0 I* (*xs*, *σ*)) ⤜
(λ-. *if xs* = [] *then RETURN* (*xs*, *σ*)
else c σ ⤜
(λ*b*. *if b then FOREACH-body f* (*xs*, *σ*) ⤜ *W*
else RETURN (*xs*, *σ*))))
(*x'*, *σ*) ⤜
(λ(-, *y*). *RETURN y*))
apply (*induct x' arbitrary*: *σ*)

apply (*subst RECT-unfold*, *refine-mono*)
apply (*simp*)
apply (*rule le-ASSERTI*)
apply *simp*

apply (*subst RECT-unfold*, *refine-mono*)
apply (*subst monadic-nfoldli-simp*)
apply (*simp del*: *conc-Id cong*: *if-cong*)
apply *refine-rcg*
apply *simp*
apply (*clarsimp simp add*: *FOREACH-body-def*)
apply (*rule-tac R*=*br* (*Pair x'*) (λ-. *True*) **in** *intro-prgR*)
apply (*simp add*: *pw-le-iff refine-pw-simps br-def*)

apply (*rule order-trans*)
apply *rprems*
apply (*simp add*: *br-def*)
done

lemma *monadic-nfoldli-arities*[*sepref-monadify-arity*]:
monadic-nfoldli ≡ λ$_2$*s c f σ*. *SP* (*monadic-nfoldli*)$*s*$(λ$_2$*x*. *c*$*x*)$(λ$_2$*x σ*. *f*$*x*$*σ*)$*σ*
by (*simp-all*)

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

$\bigwedge s\ c\ f\ \sigma.\ (monadic\text{-}nfoldli)\$s\$c\$f\$\sigma \equiv$
Refine-Basic.bind$(EVAL\$s)\$(\lambda_2 s.$ *Refine-Basic.bind*$(EVAL\$\sigma)\$(\lambda_2 \sigma.$
SP $(monadic\text{-}nfoldli)\$s\$c\$f\$\sigma$
))
by (*simp-all*)

lemma *list-rel-congD*:
assumes *A*: $(li,l)\in\langle S\rangle$*list-rel*
shows $(li,l)\in\langle S\cap(set\ li\times set\ l)\rangle$*list-rel*
proof −
{
fix *Si0 S0*
assume *set li* ⊆ *Si0 set l* ⊆ *S0*
with *A* **have** $(li,l)\in\langle S\cap(Si0\times S0)\rangle$*list-rel*
by (*induction rule*: *list-rel-induct*) *auto*
} **from** *this*[*OF order-refl order-refl*] **show** *?thesis* **.**
qed

lemma *monadic-nfoldli-refine*[*refine*]:
assumes *L*: $(li,\ l)\ \in\ \langle S\rangle$*list-rel*
and [*simp*]: $(si,\ s)\ \in\ R$
and *CR*[*refine*]: $\bigwedge si\ s.\ (si,s)\in R \Longrightarrow ci\ si \leq \Downarrow$*bool-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* ≤ ⇓ *R* (*monadic-nfoldli l c f s*)

supply *RELATESI*[*of S*∩(*set li*×*set l*), *refine-dref-RELATES*]
supply *RELATESI*[*of R*, *refine-dref-RELATES*]
unfolding *monadic-nfoldli-def*
apply (*refine-rcg bind-refine'*)
apply *refine-dref-type*
apply (*vc-solve simp*: *list-rel-congD*[*OF L*])
done

lemma *monadic-FOREACH-itsl*:
fixes *R I tsl*
shows
do { *l* ← *it-to-sorted-list R s*; *monadic-nfoldli l c f σ* }
≤ *monadic-FOREACH R I s c f σ*
apply (*rule refine-IdD*)
unfolding *monadic-FOREACH-def it-to-sorted-list-def*
apply (*refine-rcg*)
apply *simp*
apply (*rule monadic-nfoldli-rec*[*simplified*])
done

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
apply (*rule refine-IdD*)
unfolding *FOREACHoci-def it-to-sorted-list-def*
apply *refine-rcg*
apply *simp*
apply (*rule nfoldli-while*)
done

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*)
by (*simp-all add*:
FOREACHci-def FOREACHi-def[*abs-def*] *FOREACHc-def FOREACH-def*[*abs-def*])

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*)
by *simp*

We set up the monadify-phase such that all FOREACH-loops get rewritten to the monadic version of FOREACH

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

PR-CONST (*FOREACHoci R I*) $\equiv \lambda_2 s\ c\ f\ \sigma$. *SP* (*PR-CONST* (*FOREACHoci R I*))\$*s*\$($\lambda_2 x$. *c*\$*x*)\$($\lambda_2 x\ \sigma$. *f*\$*x*\$$\sigma$)\$$\sigma$
by (*simp-all*)

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$
))
by (*simp-all add*: *FOREACH-oci-to-monadic*)

Imperative Version of nfoldli

We define an imperative version of *nfoldli*. It is the equivalent to the monadic version in the nres-monad

definition *imp-nfoldli l c f s* $\equiv$ *heap.fixp-fun* (λD (*l*,*s*). *case l of*
[] $\Rightarrow$ *return s*
| *x*#*ls* $\Rightarrow$ *do* {
b$\leftarrow$*c s*;
if b then do { *s*'$\leftarrow$*f x s*; *D* (*ls*,*s*')} *else return s*
}

```
  ) (l,s)

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
  })
  apply -
  unfolding imp-nfoldli-def
  apply (subst heap.mono-body-fixp)
  apply pf-mono
  apply simp
  apply (subst heap.mono-body-fixp)
  apply pf-mono
  apply simp
  done



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
    apply (rule hn-refine-cons-post)
    apply (rule hn-refine-frame[OF hnr-RETURN-pass])
    apply (tactic ‹Sepref-Frame.frame-tac (K (K no-tac)) @{context} 1›)
```

apply (*simp add*: *hn-ctxt-def ent-true-drop invalid-assn-const*)
solved

apply1 *weaken-hnr-post*
apply1 (*simp only*: *imp-nfoldli-simps monadic-nfoldli-simp*)
applyF (*rule hnr-bind*)
  apply1 (*rule hn-refine-frame*[*OF c-ref*])
  applyS (*tactic* ‹*Sepref-Frame.frame-tac* (*K* (*K no-tac*)) @{*context*} *1*›)

  applyF (*rule hnr-If*)
    applyS (*tactic* ‹*Sepref-Frame.frame-tac* (*K* (*K no-tac*)) @{*context*} *1*›)
    applyF (*rule hnr-bind*)
      apply1 (*rule hn-refine-frame*[*OF f-ref*])
      apply1 (*simp add*: *assn-assoc*)

      apply1 (*rule ent-imp-entt*)
      apply1 (*fr-rot 1*, *rule fr-refl*)
      apply1 (*fr-rot 2*, *rule fr-refl*)
      apply1 (*fr-rot 1*, *rule fr-refl*)
      applyS (*rule ent-refl*)

      applyF (*rule hn-refine-frame*)
        applyS *rprems*

        apply1 (*simp add*: *assn-assoc*)
        apply1 (*rule ent-imp-entt*)
        apply (*rule fr-refl*)
        apply1 (*fr-rot 3*, *rule fr-refl*)
        apply1 (*fr-rot 3*, *rule fr-refl*)
        applyS (*rule ent-refl*)
      solved

      apply *simp*

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

    apply1 (*rule hn-refine-frame*[*OF hnr-RETURN-pass*])
    applyS (*tactic* ‹*Sepref-Frame.frame-tac* (*K* (*K no-tac*)) @{*context*} *1*›)

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

  apply (*rule enttI*)
  apply (*fr-rot-rhs 1*)
  apply (*fr-rot 3*, *rule fr-refl*)
  applyS (*fr-rot 3*, *rule ent-star-mono*[*rotated*]; *sep-auto simp*: *hn-ctxt-def*)
solved

applyS (*simp add*: *hn-ctxt-def invalid-assn-def*)

applyS (*rule*, *sep-auto*)
solved
done

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

apply (*rule hn-refine-cons-pre*[*OF FR*])
unfolding *APP-def*
apply (*rule monadic-nfoldli-refine-aux*)
apply (*rule c-ref*[*unfolded APP-def*])
apply (*rule f-ref*[*unfolded APP-def*])
done

definition
imp-foreach :: $('b \Rightarrow 'c\ list\ Heap) \Rightarrow 'b \Rightarrow ('a \Rightarrow bool\ Heap) \Rightarrow ('c \Rightarrow 'a \Rightarrow 'a\ Heap) \Rightarrow 'a \Rightarrow 'a\ Heap$
where
imp-foreach tsl s c f σ ≡ *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*)
apply *rule*
apply (*rule ccpo.fixp-mono*[*OF heap.ccpo*, *THEN fun-ordD*])
apply (*rule mono-fun-fun-cnv*, *erule thin-rl*, *pf-mono*)+

apply (*rule fun-ordI*)
apply (*erule monotoneD*[*of fun-ord Heap-ord Heap-ord*, *rotated*])
apply *pf-mono*
done

lemma *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)
unfolding *imp-nfoldli-def*
by *pf-mono*

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* ($\lambda x.$ *imp-foreach tsl l c* (*f x*) σ)
unfolding *imp-foreach-def*
by *pf-mono*

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

definition
IS-TO-SORTED-LIST Ω *Rs Rk tsl* $\equiv$ (*tsl*,*it-to-sorted-list* Ω) $\in$ $(Rs)^k \to_a$ *list-assn Rk*

lemma *IS-TO-SORTED-LISTI*:
assumes (*tsl*,*PR-CONST* (*it-to-sorted-list* Ω)) $\in$ $(Rs)^k \to_a$ *list-assn Rk*
shows *IS-TO-SORTED-LIST* Ω *Rs Rk tsl*
using *assms* **unfolding** *IS-TO-SORTED-LIST-def PR-CONST-def* .

lemma *hn-monadic-FOREACH*[*sepref-comb-rules*]:
assumes *INDEP Rk INDEP Rs INDEP* $R\sigma$
assumes *FR*: $P \Longrightarrow_t \Gamma$ * *hn-ctxt Rs s' s* * *hn-ctxt* $R\sigma$ σ' σ
assumes *STL*: *GEN-ALGO tsl* (*IS-TO-SORTED-LIST ordR Rs Rk*)
assumes *c-ref*: $\bigwedge\sigma\ \sigma'.$ *hn-refine*
(Γ * *hn-ctxt Rs s' s* * *hn-ctxt* $R\sigma$ σ' σ)
(*c* σ)
(Γc σ' σ)
bool-assn
(*c'* σ')
assumes *C-FR*:
$\bigwedge\sigma'\ \sigma.$ *TERM monadic-FOREACH* $\Longrightarrow$
Γc σ' σ $\Longrightarrow_t$ Γ * *hn-ctxt Rs s' s* * *hn-ctxt* $R\sigma$ σ' σ

assumes *f-ref*: $\bigwedge x'\ x\ \sigma'\ \sigma.$ *hn-refine*
(Γ * *hn-ctxt Rs s' s* * *hn-ctxt Rk x' x* * *hn-ctxt* $R\sigma$ σ' σ)
(*f x* σ)
(Γf x' x σ' σ) $R\sigma$
(*f'* x' σ')
assumes *F-FR*: $\bigwedge x'\ x\ \sigma'\ \sigma.$ *TERM monadic-FOREACH* $\Longrightarrow$ Γf x' x σ' σ $\Longrightarrow_t$

```
    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))
      $s′$(λ2σ′. c′ σ′)$(λ2x′ σ′. f′ x′ σ′)$σ′
    )
proof −
  from STL have STL: (tsl,it-to-sorted-list ordR) ∈ (Rs)^k →a list-assn Rk
    unfolding GEN-ALGO-def IS-TO-SORTED-LIST-def by simp

  show ?thesis
    apply (rule hn-refine-cons-pre[OF FR])
    apply weaken-hnr-post
    unfolding APP-def PROTECT2-def PR-CONST-def imp-foreach-def
    apply (rule hn-refine-ref[OF monadic-FOREACH-itsl])
    apply (rule hn-refine-guessI)
    apply (rule hnr-bind)
    apply (rule hn-refine-frame)
    apply (rule STL[to-hnr, unfolded APP-def])
    apply (tactic ‹Sepref-Frame.frame-tac (K (K no-tac)) @{context} 1›)
    apply (rule hn-monadic-nfoldli[unfolded APP-def])
    apply (tactic ‹Sepref-Frame.frame-tac (K (K no-tac)) @{context} 1›)
    apply (rule hn-refine-cons-post)
    apply (rule c-ref[unfolded APP-def])
    apply (rule C-FR)
    apply (rule TERMI)
    apply weaken-hnr-post
    apply (rule hn-refine-cons-post)
    apply (rule f-ref[unfolded APP-def])
    apply (rule entt-trans[OF F-FR])
    apply (rule TERMI)
    applyS (tactic ‹Sepref-Frame.frame-tac (K (K no-tac)) @{context} 1›)
    applyS (tactic ‹Sepref-Frame.frame-tac (K (K no-tac)) @{context} 1›)

    apply simp
    done

qed

lemma monadic-nfoldli-assert-aux:
  assumes set l ⊆ S
  shows monadic-nfoldli l c (λx s. ASSERT (x∈S)≫f x s) s = monadic-nfoldli l
c f s
  using assms
  apply (induction l arbitrary: s)
```

apply (*auto simp*: *pw-eq-iff refine-pw-simps*)
done

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

lemma *nfoldli-arities*[*sepref-monadify-arity*]:
nfoldli ≡ $\lambda_2 s\ c\ f\ \sigma$. *SP* (*nfoldli*)$s$($\lambda_2 x$. $c$$x$)$($\lambda_2 x\ \sigma$. $f$$x$$\sigma$)$\sigma$
by (*simp-all*)

lemma *nfoldli-comb*[*sepref-monadify-comb*]:
$\bigwedge s\ c\ f\ \sigma$. (*nfoldli*)$s$($\lambda_2 x$. $c\ x$)$f$$\sigma$ ≡
Refine-Basic.bind$(*EVAL*$s$)$($\lambda_2 s$. *Refine-Basic.bind*$(*EVAL*$\sigma$)$($\lambda_2 \sigma$.
SP (*monadic-nfoldli*)s($\lambda_2 x$. (*EVAL*$(c\ x)$)))$f$$\sigma$
))
by (*simp-all add*: *nfoldli-to-monadic*)

lemma *monadic-nfoldli-refine-aux'*:
assumes *SS*: *set* $l' \subseteq S$
assumes *c-ref*: $\bigwedge s\ s'$. *hn-refine*
(Γ ∗ *hn-ctxt Rs* s' *s*)
(*c s*)
(Γ ∗ *hn-ctxt Rs* s' *s*)
bool-assn
(c' s')
assumes *f-ref*: $\bigwedge x\ x'\ s\ s'$. $[\![x' \in S]\!] \Longrightarrow$ *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*]: $\bigwedge x\ x'$. *hn-ctxt Rl'* x' *x* $\vee_A$ *hn-ctxt Rl* x' *x* $\Longrightarrow_t$ *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*
apply (*rule hn-refine-cons-post*)
apply (*rule hn-refine-frame*[*OF hnr-RETURN-pass*])
apply (*tactic* ‹*Sepref-Frame.frame-tac* (*K* (*K no-tac*)) @{*context*} *1*›)
apply (*simp add*: *hn-ctxt-def ent-true-drop*)
solved

apply (*simp only*: *imp-nfoldli-simps monadic-nfoldli-simp*)
apply (*rule hnr-bind*)
apply (*rule hn-refine-frame*[*OF c-ref*])
apply (*tactic* ‹*Sepref-Frame.frame-tac* (*K* (*K no-tac*)) @{*context*} *1*›)

apply (*rule hnr-If*)
apply (*tactic* ‹*Sepref-Frame.frame-tac* (*K* (*K no-tac*)) @{*context*} *1*›)
apply (*simp only*: *nres-monad-laws*)
apply (*rule hnr-ASSERT*)
apply (*rule hnr-bind*)
apply (*rule hn-refine-frame*[*OF f-ref*])
apply *assumption*
apply (*simp add*: *assn-aci*)
apply (*rule ent-imp-entt*)
apply (*fr-rot-rhs 1*)
apply (*fr-rot 2*)
apply (*rule fr-refl*)
apply (*rule fr-refl*)
apply (*rule fr-refl*)
apply (*rule ent-refl*)

applyF (*rule hn-refine-frame*)
  applyS *rprems*

  focus
    apply (*simp add*: *assn-aci*)
    apply (*rule ent-imp-entt*)

    apply (*fr-rot-rhs 1*, *rule fr-refl*)
    apply (*fr-rot 2*, *rule fr-refl*)
    apply (*fr-rot 1*, *rule fr-refl*)
    apply (*rule ent-refl*)
  solved
solved

focus (*simp add*: *assn-assoc*)
  apply (*rule ent-imp-entt*)
  apply (*rule fr-refl*)
  apply (*rule ent-refl*)
solved

apply1 (*rule hn-refine-frame*[*OF hnr-RETURN-pass*])
applyS (*tactic* ‹*Sepref-Frame.frame-tac* (*K* (*K no-tac*)) @{*context*} *1*›)

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

apply *simp*
apply (*rule ent-imp-entt*)
apply *solve-entails*
apply (*rule*, *sep-auto*)
apply (*rule*, *sep-auto*)
solved
done

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*: $\bigwedge\sigma\ \sigma'.\ \textit{hn-refine}$
($\Gamma * \textit{hn-ctxt}\ R\sigma\ \sigma'\ \sigma$)
($c\ \sigma$)
($\Gamma c\ \sigma'\ \sigma$)
bool-assn
($c'\ \sigma'$)
assumes *C-FR*:
$\bigwedge\sigma'\ \sigma.\ \textit{TERM monadic-nfoldli} \Longrightarrow$
$\Gamma c\ \sigma'\ \sigma \Longrightarrow_t \Gamma * \textit{hn-ctxt}\ R\sigma\ \sigma'\ \sigma$

assumes *f-ref*: $\bigwedge x'\ x\ \sigma'\ \sigma.\ [\![x' \in \textit{set}\ s']\!] \Longrightarrow \textit{hn-refine}$
($\Gamma * \textit{hn-ctxt}\ Rk\ x'\ x * \textit{hn-ctxt}\ R\sigma\ \sigma'\ \sigma$)
($f\ x\ \sigma$)
($\Gamma f\ x'\ x\ \sigma'\ \sigma$) $R\sigma$
($f'\ x'\ \sigma'$)
assumes *F-FR*: $\bigwedge x'\ x\ \sigma'\ \sigma.\ \textit{TERM monadic-nfoldli} \Longrightarrow \Gamma f\ x'\ x\ \sigma'\ \sigma \Longrightarrow_t$
$\Gamma * \textit{hn-ctxt}\ Rk'\ x'\ x * \textit{hn-ctxt}\ Pf\sigma\ \sigma'\ \sigma$

assumes *MERGE*: $\bigwedge x\ x'.\ \textit{hn-ctxt}\ Rk'\ x'\ x \vee_A \textit{hn-ctxt}\ Rk\ x'\ x \Longrightarrow_t \textit{hn-ctxt}\ Rk''\ x'\ x$

shows *hn-refine*
P
(*imp-nfoldli s c f σ*)
($\Gamma * \textit{hn-ctxt}\ (\textit{list-assn}\ Rk'')\ s'\ s * \textit{hn-invalid}\ R\sigma\ \sigma'\ \sigma$)
$R\sigma$
((*monadic-nfoldli*)
$s'$$(λ_2σ'. c' σ')$$(λ_2x' σ'. f' x' σ')$$σ'
)
unfolding *APP-def PROTECT2-def PR-CONST-def*
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
apply (*rule hn-refine-cons-post*)
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

apply (*rule MERGE*)
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
done

lemma *nfoldli-assert*:
assumes *set l* ⊆ *S*
shows *nfoldli l c* (λ *x s*. *ASSERT* (*x* ∈ *S*) ≫ *f x s*) *s* = *nfoldli l c f s*
using *assms* **by** (*induction l arbitrary*: *s*) (*auto simp*: *pw-eq-iff refine-pw-simps*)

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*
apply (*induction l arbitrary*: *s*) **apply** (*auto*) **done**

lemma *fold-eq-nfoldli-assert*:
RETURN (*fold f l s*) = *nfoldli l* (λ-. *True*) (λ*x s*. *ASSERT* (*x*∈*set l*) ≫ *RETURN* (*f x s*)) *s*
by (*simp add*: *nfoldli-assert' fold-eq-nfoldli*)

lemma *fold-arity*[*sepref-monadify-arity*]: *fold* ≡ λ$_2$*f l s*. *SP fold*$(λ$_2$*x s*. *f*$*x*$*s*)$*l*$*s*
by *auto*

lemma *monadify-plain-fold*[*sepref-monadify-comb*]:
EVAL$(*fold*$(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*))

by (*simp add*: *fold-eq-nfoldli*)

lemma *monadify-plain-fold-old-rl*:
EVAL$(*fold*$($\lambda_2 x$ *s*. *f x s*)$*l*$*s*) ≡ (≫=)$(*EVAL*$*l*)$($\lambda_2 l$. (≫=)$(*EVAL*$*s*)$($\lambda_2 s$. *nfoldli*$*l*$(λ_2-. *True*)$($\lambda_2 x$ *s*. *PR-CONST* (*op-ASSERT-bind* (*x*∈*set l*))$(*EVAL*$(*f x s*)))$*s*))
by (*simp add*: *fold-eq-nfoldli-assert*)

foldli

lemma *foldli-eq-nfoldli*:
RETURN (*foldli l c f s*) = *nfoldli l c* (λx *s*. *RETURN* (*f x s*)) *s*
by (*induction l arbitrary*: *s*) *auto*

lemma *foldli-arities*[*sepref-monadify-arity*]:
foldli ≡ $\lambda_2 s$ *c f* σ. *SP* (*foldli*)$*s*$($\lambda_2 x$. *c*$*x*)$($\lambda_2 x$ σ. *f*$*x*$$\sigma$)$$\sigma$
by (*simp-all*)

lemma *monadify-plain-foldli*[*sepref-monadify-comb*]:
EVAL$(*foldli*$*l*$*c*$($\lambda_2 x$ *s*. *f x s*)$*s*) ≡
(≫=)$(*EVAL*$*l*)$
($\lambda_2 l$. (≫=)$(*EVAL*$*s*)$
($\lambda_2 s$. *nfoldli*$*l*$*c*$($\lambda_2 x$ *s*. (*EVAL*$(*f x s*)))$*s*))
by (*simp add*: *foldli-eq-nfoldli*)

Deforestation

lemma *nfoldli-filter-deforestation*:
nfoldli (*filter P xs*) *c f s* = *nfoldli xs c* (λx *s*. *if P x then f x s else RETURN s*) *s*
apply (*induction xs arbitrary*: *s*)
by (*auto simp*: *pw-eq-iff refine-pw-simps*)

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 −
obtain *xs2* **where** {*x*∈*S*. ¬*P x*} = *set xs2 distinct xs2*
using *finite-distinct-list*[**where** *A*={*x*∈*S*. ¬*P x*}] **by** *auto*
with *A* **have** *xs′* = *filter P* (*xs′*@*xs2*) *distinct* (*xs′*@*xs2*) *set* (*xs′*@*xs2*) = *S*
by (*auto simp*: *filter-empty-conv*)
from *that*[*OF this*] **show** *?thesis* **.**
qed

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

```
 unfolding FOREACHc-def FOREACHci-def FOREACHoci-by-LIST-FOREACH
LIST-FOREACH'-eq
     LIST-FOREACH'-def it-to-sorted-list-def
 subgoal
 proof (induction rule: antisym[consumes 0, case-names 1 2])
   case 1
   then show ?case
     apply (rule le-ASSERTI)
     apply (rule ASSERT-leI, simp)
     apply (rule intro-spec-refine[where R=Id, simplified]; clarsimp)
    apply (rule extend-list-of-filtered-set[OF FIN - sym], assumption, assumption)
     subgoal for xs' xs
       apply (rule rhs-step-bind-SPEC[where R=Id and x'=xs, simplified])
       applyS simp
       applyS (simp add: nfoldli-filter-deforestation)
       done
     done
 next
   case 2
   then show ?case
   apply (rule le-ASSERTI)
   apply (rule ASSERT-leI, (simp; fail))
   apply (rule intro-spec-refine[where R=Id, simplified]; clarsimp)
   subgoal for xs
    apply (rule rhs-step-bind-SPEC[where R=Id and x'=filter P xs, simplified])
     apply simp
     apply (simp add: nfoldli-filter-deforestation)
     done
   done
 qed
 done

lemma FOREACHc-filter-deforestation2:
 assumes [simp]: distinct xs
 shows (FOREACHc (set (filter P xs)) c f s)
   = FOREACHc (set xs) c (λx s. if P x then f x s else RETURN s) s
 using FOREACHc-filter-deforestation[of set xs, simplified, folded set-filter]
 .
```

2.2.2 For Loops

```
partial-function (heap) imp-for :: nat ⇒ 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}
by (auto simp: imp-for.simps)

lemma imp-nfoldli-deforest[sepref-opt-simps]:
  imp-nfoldli [l..<u] c = imp-for l u c
 apply (intro ext)
 subgoal for f s
  apply (induction u - l arbitrary: l u s)
  apply (simp add: upt-conv-Cons; fail)
  apply (simp add: upt-conv-Cons)
  apply (fo-rule arg-cong)
 by (auto cong: if-cong)
done

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 ≥ u ⟹ imp-for' i u f s = return s
  i < u ⟹ imp-for' i u f s = f i s ⤜ imp-for' (i + 1) u f
by (auto simp: imp-for'.simps)

lemma imp-for-imp-for'[sepref-opt-simps]:
  imp-for i u (λ -. return True) = imp-for' i u
apply (intro ext)
subgoal for f s
  apply (induction u - i arbitrary: i u s)
  apply (simp; fail)
  apply simp
  apply (fo-rule arg-cong)
  by auto
done

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 (intro ext)
  fix f s
  show imp-nfoldli (rev [l..<u]) c f s =
         (if l ≥ u then return s else imp-for-down l u c f s)
  proof cases
    assume l≥u thus ?thesis by auto
  next
    assume ¬(l≥u) hence l<u by auto
    thus ?thesis
      apply simp
    proof (induction u − l arbitrary: u s)
      case 0 thus ?case by auto
    next
      case (Suc u')
      from Suc.prems Suc.hyps(2) have [simp]: rev [l..<u] = (u−1)#rev [l..<u−1]
          apply simp
          apply (subst upt-Suc-append[symmetric])
          apply auto
          done
        show ?case using Suc.hyps(1)[of u−1] Suc.hyps(2) Suc.prems
          apply (subst imp-for-down.simps)
          apply (cases l < u − Suc 0)
          apply (auto simp: Let-def cong: if-cong)
          done
      qed
    qed
  qed

context begin

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

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
  apply (induction l u rule: imp-for-down-induction-scheme.induct)
  apply (intro ext)
  apply (subst imp-for-down.simps)
  apply (subst imp-for-down'.simps)
  apply (simp cong: if-cong)
  done

end


lemma imp-for'-rule:
  assumes LESS: l≤u
  assumes PRE: P ⟹_A I l s
  assumes STEP: ⋀i s. ⟦ l≤i; i<u ⟧ ⟹ <I i s> f i s <I (i+1)>
  shows <P> imp-for' l u f s <I u>
  apply (rule Hoare-Triple.cons-pre-rule[OF PRE])
  using LESS
proof (induction arbitrary: s rule: inc-induct)
  case base thus ?case by sep-auto
next
  case (step k)
  show ?case using step.hyps
    by (sep-auto heap: STEP step.IH)
qed
```

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

```
lemma fold-idx-conv: fold f l s = fold (λi. f (l!i)) [0..<length l] s
proof (induction l arbitrary: s rule: rev-induct)
  case Nil thus ?case by simp
next
  case (snoc x l)
  { fix x s
   have fold (λa. f ((l @ [x]) ! a)) [0..<length l] s = fold (λa. f (l ! a)) [0..<length
l] s
     by (rule fold-cong) (simp-all add: nth-append)
  }
  with snoc show ?case by simp
qed


end
```

2.3 Ad-Hoc Solutions

theory *Sepref-Improper*
imports
Sepref-Tool
Sepref-HOL-Bindings

Sepref-Foreach
Sepref-Intf-Util
begin

This theory provides some ad-hoc solutions to practical problems, that, however, still need a more robust/clean solution

2.3.1 Pure Higher-Order Functions

Ad-Hoc way to support pure higher-order arguments

definition *pho-apply* :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ **where** [*code-unfold*,*simp*]: *pho-apply f x* = *f x*
sepref-register *pho-apply*

lemmas *fold-pho-apply* = *pho-apply-def*[*symmetric*]

lemma *pure-fun-refine*[*sepref-fr-rules*]: *hn-refine*
(*hn-val* (*A*→*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*))
by (*sep-auto intro*!: *hn-refineI simp*: *pure-def hn-ctxt-def dest*: *fun-relD*)

end
theory *Sepref*
imports
Sepref-Tool
Sepref-HOL-Bindings

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

end

Chapter 3

The Imperative Isabelle Collection Framework

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

3.1 Set Interface

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

3.1.1 Operations

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

definition *op-set-copy* :: *'a set* ⇒ *'a set* **where** [*simp*]: *op-set-copy s* ≡ *s*

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

sepref-decl-op (*no-def*) *set-copy*: *op-set-copy* :: ⟨*A*⟩*set-rel* → ⟨*A*⟩*set-rel* **where** *A* = *Id* .
sepref-decl-op *set-empty*: {} :: ⟨*A*⟩*set-rel* .
sepref-decl-op (*no-def*) *set-is-empty*: *op-set-is-empty* :: ⟨*A*⟩*set-rel* → *bool-rel* .
sepref-decl-op *set-member*: (∈) :: *A* → ⟨*A*⟩*set-rel* → *bool-rel* **where** *IS-LEFT-UNIQUE A IS-RIGHT-UNIQUE A* .
sepref-decl-op *set-insert*: *Set.insert* :: *A* → ⟨*A*⟩*set-rel* → ⟨*A*⟩*set-rel* **where** *IS-RIGHT-UNIQUE A* .
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* .
sepref-decl-op *set-union*: $(\cup) :: \langle A\rangle$*set-rel* $\to \langle A\rangle$*set-rel* $\to \langle A\rangle$*set-rel* .
sepref-decl-op *set-inter*: $(\cap) :: \langle A\rangle$*set-rel* $\to \langle A\rangle$*set-rel* $\to \langle A\rangle$*set-rel* **where** *IS-LEFT-UNIQUE A IS-RIGHT-UNIQUE A* .
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* .
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* .
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* .

sepref-decl-op *set-pick*: *RES* $:: [\lambda s.\ s{\neq}\{\}]_f\ \langle K\rangle$*set-rel* $\to K$ **by** *auto*

sepref-decl-op *set-size*: (*card*) $:: \langle A\rangle$*set-rel* $\to$ *nat-rel* **where** *IS-LEFT-UNIQUE A IS-RIGHT-UNIQUE A* .

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*
by (*auto intro*!: *eq-reflection*)

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*
by (*auto intro*!: *eq-reflection*)

locale *set-custom-empty* =
fixes *empty* **and** *op-custom-empty* :: *'a set*
assumes *op-custom-empty-def*: *op-custom-empty* = *op-set-empty*
begin
sepref-register *op-custom-empty* :: *'ax set*

lemma *fold-custom-empty*:
{} = *op-custom-empty*
op-set-empty = *op-custom-empty*
mop-set-empty = *RETURN op-custom-empty*
unfolding *op-custom-empty-def* **by** *simp-all*
end

end

3.2 Sets by Lists that Own their Elements

theory *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*)
unfolding *lso-assn-def* **by** *safe-constraint*

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

∈ *unit-rel* →$_f$ ⟨*br set* (λ-. *True*)⟩*nres-rel*
by (*auto simp*: *in-br-conv intro*!: *frefI nres-relI*)

lemma *lso-ins-aref*: (*uncurry* (*RETURN oo* ((#))), *uncurry* (*RETURN oo op-set-insert*))
∈ *Id* ×$_r$ *br set* (λ-. *True*) →$_f$ ⟨*br set* (λ-. *True*)⟩*nres-rel*
by (*auto simp*: *in-br-conv intro*!: *frefI nres-relI*)

sepref-decl-impl (*no-register*) *lso-empty*: *hn-Nil*[*to-hfref*] **uses** *lso-empty-aref* .

definition [*simp*]: *op-lso-empty* ≡ *op-set-empty*
lemma *lso-fold-custom-empty*:
{} = *op-lso-empty*
op-set-empty = *op-lso-empty*
by *auto*
lemmas [*sepref-fr-rules*] = *lso-empty-hnr*[*folded op-lso-empty-def*]

sepref-decl-impl *lso-insert*: *hn-Cons*[*to-hfref*] **uses** *lso-ins-aref* .

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

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

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

lemma *op-mop-lso-bex*: *RETURN* (*op-lso-bex P S*) = *mop-lso-bex* (*RETURN o P*) *S* **by** *simp*

sepref-register *op-lso-bex*
lemma *lso-bex-arity*[*sepref-monadify-arity*]:
op-lso-bex ≡ λ$_2$*P s*. *SP op-lso-bex*$(λ$_2$*x*. *P*$*x*)$*s* **by** (*auto intro*!: *eq-reflection ext*)
lemma *op-lso-bex-monadify*[*sepref-monadify-comb*]:
EVAL$(*op-lso-bex*$(λ$_2$*x*. *P x*)$*s*) ≡ (≫=) $(*EVAL*$*s*)$(λ$_2$*s*. *mop-lso-bex*$(λ$_2$*x*. *EVAL* $ *P x*)$*s*) **by** *simp*

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 −
{ fix *b*
have *nfoldli l* (*Not*) (λ*x* -. *P x*) *b* ≤ *ASSERT* (∀ *x*∈*set l*. ∃ *y*. *P x* = *RETURN y*) ≫ *RETURN* ((∃ *x*∈*set l*. *P x* = *RETURN True*) ∨ *b*)
apply (*induction l arbitrary*: *b*)
applyS *simp*
applyS (*clarsimp simp add*: *pw-le-iff refine-pw-simps*; *blast*)
done
} from *this*[*of False*] **show** *?thesis* **by** (*simp add*: *lso-abex-def*)
qed

locale *lso-bex-impl-loc* =
fixes *Pi* **and** *P* :: *'a* ⇒ *bool nres*
fixes *li* :: *'c list* **and** *l* :: *'a list*
fixes *A* :: *'a* ⇒ *'c* ⇒ *assn*
fixes *F* :: *assn*

assumes *Prl*: ⋀*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*)
using *hn-refine-frame*[*OF Prl*[*OF assms*(*2*)], *of* Γ *F'*] *assms*(*1*)
by (*simp add*: *assn-assoc*)

schematic-goal *lso-bex-impl*:
hn-refine (*hn-ctxt* (*list-assn A*) *l li* ∗ *F*) (*?c*) (*F* ∗ *hn-ctxt* (*list-assn A*) *l li*) *bool-assn* (*lso-abex P l*)
unfolding *lso-abex-def*[*abs-def*]
by *sepref*
end
concrete-definition *lso-bex-impl* **uses** *lso-bex-impl-loc.lso-bex-impl*

lemma *hn-lso-bex*[*sepref-prep-comb-rule*,*sepref-comb-rules*]:
assumes *FR*: Γ ⟹$_t$ *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*)
apply (*rule hn-refine-cons-pre*[*OF FR*])
apply (*clarsimp simp*: *hn-ctxt-def lso-assn-def hr-comp-def in-br-conv hnr-pre-ex-conv*)
apply (*rule hn-refine-preI*)
apply (*drule mod-starD*; *clarsimp*)
apply (*rule hn-refine-ref*[*OF lso-abex-to-set*])
proof −
fix *l* **assume** [*simp*]: *s*=*set l*

from *Prl* **have** *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*)
by *simp*

show *hn-refine* (*list-assn A l li* ∗ *F*) (*lso-bex-impl Pi li*) (∃$_A$*ba*. *F* ∗ *list-assn A ba li* ∗ ↑ (*set l* = *set ba*)) *bool-assn* (*lso-abex P l*)
apply (*rule hn-refine-cons*[*OF - lso-bex-impl.refine*])
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*)
done
qed

end

3.3 Multiset Interface

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

3.3.1 Additions to Multiset Theory

lemma *rel-mset-Plus-gen*:

assumes *rel-mset A m m′*
assumes *rel-mset A n n′*
shows *rel-mset A* (*m*+*n*) (*m′*+*n′*)
using *assms*
by *induction* (*auto simp*: *algebra-simps dest*: *rel-mset-Plus*)

lemma *rel-mset-single*:
assumes *A x y*
shows *rel-mset A* {#*x*#} {#*y*#}
unfolding *rel-mset-def*
apply (*rule exI*[**where** *x*=[*x*]])
apply (*rule exI*[**where** *x*=[*y*]])
using *assms* **by** *auto*

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*#})
unfolding *rel-mset-def*
proof *clarsimp*
fix *ml nl*
assume *A*: *A x y*
assume *R*: *list-all2 A ml nl*
show ∃ *ml′*. *mset ml′* = *mset ml* − {#*x*#} ∧
(∃ *nl′*. *mset nl′* = *mset nl* − {#*y*#} ∧ *list-all2 A ml′ nl′*)
proof (*cases x*∈*set ml*)
case *False*
have *y* ∉ *set nl* **using** *A R*
apply (*auto simp*: *in-set-conv-decomp list-all2-append2 list-all2-Cons2*)
using *False BIU*[*unfolded bi-unique-alt-def*]
apply (*auto dest*: *left-uniqueD*)
done
with *False R* **show** *?thesis* **by** (*auto simp*: *diff-single-trivial in-multiset-in-set*)
next
case *True*
then obtain *ml1 ml2* **where** [*simp*]: *ml*=*ml1*@*x*#*ml2* **by** (*auto simp*: *in-set-conv-decomp*)
then obtain *nl1 nl2* **where** [*simp*]: *nl*=*nl1*@*y*#*nl2*
and *LA*: *list-all2 A ml1 nl1 list-all2 A ml2 nl2*
using *A R*
apply (*auto simp*: *in-set-conv-decomp list-all2-append1 list-all2-Cons1*)
using *BIU*[*unfolded bi-unique-alt-def*]
apply (*auto dest*: *right-uniqueD*)
done
have
mset (*ml1*@*ml2*) = *mset ml* − {#*x*#}
mset (*nl1*@*nl2*) = *mset nl* − {#*y*#}
using *R*
by (*auto simp*: *algebra-simps add-implies-diff union-assoc*)
moreover have *list-all2 A* (*ml1*@*ml2*) (*nl1*@*nl2*)
by (*rule list-all2-appendI*) *fact*+

ultimately show *?thesis* **by** *blast*
qed
qed

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'*)
using *assms*(*3*,*2*)
apply (*induction R*≡*A - - rule*: *rel-mset-induct*)
apply (*auto dest*: *rel-mset-Minus*[*OF BIU*] *simp*: *algebra-simps*)
done

lemma *pcr-count*:
assumes *bi-unique A*
shows *rel-fun* (*rel-mset A*) (*rel-fun A* (=)) *count count*
apply (*intro rel-funI*)
unfolding *rel-mset-def*
apply *clarsimp*
subgoal for *x y xs ys*
apply (*rotate-tac*,*induction xs ys rule*: *list-all2-induct*)
using *assms*
by (*auto simp*: *bi-unique-alt-def left-uniqueD right-uniqueD*)
done

3.3.2 Parametricity Setup

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

lemma *rel2p-mset*[*rel2p*]: *rel2p* (⟨*A*⟩*mset-rel*) = *rel-mset* (*rel2p A*)
by (*simp add*: *mset-rel-def*)

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

lemma *mset-rel-empty*[*simp*]:
(*a*,{#})∈⟨*A*⟩*mset-rel* ⟷ *a*={#}
({#},*b*)∈⟨*A*⟩*mset-rel* ⟷ *b*={#}
by (*auto simp*: *mset-rel-def p2rel-def rel-mset-def*)

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

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

apply (*rule rel2pD*)
apply (*simp add*: *rel2p*)

apply (*intro rel-funI*)
by (*rule rel-mset-Plus-gen*)

lemma *param-mset-add*[*param*]: (*add-mset*, *add-mset*) ∈ *A* → ⟨*A*⟩*mset-rel* → ⟨*A*⟩*mset-rel*
apply (*rule rel2pD*)
apply (*simp add*: *rel2p*)
apply (*intro rel-funI*)
by (*rule rel-mset-Plus*)

lemma *param-mset-minus*[*param*]: ⟦*single-valued A*; *single-valued* (A^{-1})⟧
⟹ ((−), (−)) ∈ ⟨*A*⟩*mset-rel* → ⟨*A*⟩*mset-rel* → ⟨*A*⟩*mset-rel*
apply (*rule rel2pD*)
apply (*simp add*: *rel2p*)
apply (*intro rel-funI*)
apply (*rule rel-mset-Minus-gen*)
subgoal apply (*unfold IS-LEFT-UNIQUE-def*[*symmetric*])
by (*simp add*: *prop2p bi-unique-alt-def*)
apply (*simp*; *fail*)
apply (*simp*; *fail*)
done

lemma *param-count*[*param*]: ⟦*single-valued A*; *single-valued* (A^{-1})⟧ ⟹ (*count*,*count*)∈⟨*A*⟩*mset-rel*
→ *A* → *nat-rel*
apply (*rule rel2pD*)
apply (*simp add*: *prop2p rel2p*)
apply (*rule pcr-count*)
apply (*simp add*: *bi-unique-alt-def*)
done

lemma *param-set-mset*[*param*]:
shows (*set-mset*, *set-mset*) ∈ ⟨*A*⟩*mset-rel* → ⟨*A*⟩*set-rel*
apply (*rule rel2pD*; *simp add*: *rel2p*)
by (*rule multiset.set-transfer*)

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

lemma *mset-is-empty-param*[*param*]: (*mset-is-empty*,*mset-is-empty*) ∈ ⟨*A*⟩*mset-rel*
→ *bool-rel*
unfolding *mset-rel-def mset-is-empty-def*[*abs-def*]
by (*auto simp*: *p2rel-def rel-mset-def intro*: *nres-relI*)

3.3.3 Operations

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

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

unfolding *mset-is-empty-def*[*symmetric*]
apply (*rule frefI*)
by *parametricity*

sepref-decl-op *mset-insert*: *add-mset* :: $A \to \langle A\rangle$*mset-rel* $\to \langle A\rangle$*mset-rel* .

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}) .

sepref-decl-op *mset-plus*: (+)::- *multiset* ⇒ - :: $\langle A\rangle$*mset-rel* $\to \langle A\rangle$*mset-rel* $\to$ $\langle A\rangle$*mset-rel* .
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}) .

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

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}) .

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*
unfolding *mset-is-empty-def*[*symmetric*]
apply (*intro frefI nres-relI*)
apply (*refine-vcg SPEC-refine*)
apply1 (*rule ccontr*; *clarsimp*)
applyS (*metis msed-rel-invL rel2p-def rel2p-mset union-ac*(*2*))
applyS *parametricity*
done

3.3.4 Patterns

lemma [*def-pat-rules*]:
$\{\#\} \equiv$ *op-mset-empty*
add-mset $\equiv$ *op-mset-insert*
$(=)$ \$*b*\$$\{\#\} \equiv$ *op-mset-is-empty*\$*b*
$(=)$ \$$\{\#\}$\$*b* $\equiv$ *op-mset-is-empty*\$*b*
$(+)$ \$*a*\$*b* $\equiv$ *op-mset-plus*\$*a*\$*b*
$(-)$ \$*a*\$*b* $\equiv$ *op-mset-minus*\$*a*\$*b*
by (*auto intro*!: *eq-reflection simp*: *algebra-simps*)

lemma [*def-pat-rules*]:
$(+)$ \$*b*\$(*add-mset*\$*x*\$$\{\#\}) \equiv$ *op-mset-insert*\$*x*\$*b*
$(+)$ \$(*add-mset*\$*x*\$$\{\#\})$\$*b* $\equiv$ *op-mset-insert*\$*x*\$*b*
$(-)$ \$*b*\$(*add-mset*\$*x*\$$\{\#\}) \equiv$ *op-mset-delete*\$*x*\$*b*

(<) 0(count$a$x) ≡ op-mset-contains$x$a
(∈) x(set-mset$a) ≡ op-mset-contains$x$a
by (*auto intro*!: *eq-reflection simp*: *algebra-simps*)

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*
unfolding *op-custom-empty-def* **by** *simp-all*

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
    unfolding mop-prio-pop-min-def[abs-def]
   apply (clarsimp simp: mop-prio-pop-min-def nres-rel-def pw-le-iff refine-pw-simps)
    apply (safe; simp)
  proof goal-cases
    case (1 m n x)

    assume (m,n)∈⟨A⟩mset-rel
      and x∈#m
      and P': ∀ x'∈set-mset m. prio' x ≤ prio' x'
    hence R: rel-mset (rel2p A) m n by (simp add: mset-rel-def p2rel-def)
   from multi-member-split[OF ‹x∈#m›] obtain m' where [simp]: m=m'+{#x#}
by auto

    from msed-rel-invL[OF R[simplified]] obtain n' y where
    [simp]: n=n'+{#y#} and [param, simp]: (x,y)∈A and R': (m',n')∈⟨A⟩mset-rel
      by (auto simp: rel2p-def mset-rel-def p2rel-def)
    have ∀ y'∈set-mset n. prio y ≤ prio y'
    proof
      fix y' assume y'∈set-mset n
      then obtain x' where [param]: (x',y')∈A and x'∈set-mset m
        using R
        by (metis insert-DiffM msed-rel-invR rel2pD union-single-eq-member)
      with P' have prio' x ≤ prio' x' by blast
      moreover have (prio' x ≤ prio' x', prio y ≤ prio y') ∈ bool-rel
        by parametricity
      ultimately show prio y ≤ prio y' by simp
    qed
    thus
       ∃ a. (x, a) ∈ A ∧ (m − {#x#}, n − {#a#}) ∈ ⟨A⟩mset-rel ∧ a ∈# n ∧
(∀ v'∈set-mset n. prio a ≤ prio v')
      using R' by (auto intro!: exI[where x=n'] exI[where x=y])
  qed

  lemma param-mop-prio-peek-min[param]:
    assumes [param]: (prio',prio) ∈ 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
    unfolding mop-prio-peek-min-def[abs-def]
    apply (clarsimp
```

simp: *mop-prio-pop-min-def nres-rel-def pw-le-iff refine-pw-simps*
)
apply (*safe*; *simp?*)
proof −
fix *m n x*
assume (*m*,*n*)∈⟨*A*⟩*mset-rel*
and *x*∈#*m*
and *P′*: ∀ *x′*∈*set-mset m*. *prio′ x* ≤ *prio′ x′*
hence *R*: *rel-mset* (*rel2p A*) *m n* **by** (*simp add*: *mset-rel-def p2rel-def*)
from *multi-member-split*[*OF* ‹*x*∈#*m*›] **obtain** *m′* **where** [*simp*]: *m*=*m′*+{#*x*#}
by *auto*

from *msed-rel-invL*[*OF R*[*simplified*]] **obtain** *n′ y* **where**
[*simp*]: *n*=*n′*+{#*y*#} **and** [*param*, *simp*]: (*x*,*y*)∈*A* **and** *R′*: (*m′*,*n′*)∈⟨*A*⟩*mset-rel*
by (*auto simp*: *rel2p-def mset-rel-def p2rel-def*)

have ∀ *y′*∈*set-mset n*. *prio y* ≤ *prio y′*
proof
fix *y′* **assume** *y′*∈*set-mset n*
then obtain *x′* **where** [*param*]: (*x′*,*y′*)∈*A* **and** *x′*∈*set-mset m*
using *R*
by (*metis msed-rel-invR mset-contains-eq rel2pD union-mset-add-mset-left union-single-eq-member*)
with *P′* **have** *prio′ x* ≤ *prio′ x′* **by** *blast*
moreover have (*prio′ x* ≤ *prio′ x′*, *prio y* ≤ *prio y′*) ∈ *bool-rel*
by *parametricity*
ultimately show *prio y* ≤ *prio y′* **by** *simp*
qed
thus ∃ *y*. (*x*, *y*) ∈ *A* ∧ *y* ∈# *n* ∧ (∀ *v′*∈*set-mset n*. *prio y* ≤ *prio v′*)
using *R′* **by** (*auto intro*!: *exI*[**where** *x*=*y*])
qed

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 *goal-cases*
case *1*
hence [*param*]: (*prio*,*prio*)∈*A* → *Id*
by (*auto simp*: *IS-BELOW-ID-def*)
show *?case*
apply (*rule fref-ncI*)
apply *parametricity*
by *auto*
qed

sepref-decl-op (*no-def*,*no-mop*) *prio-peek-min*:
PR-CONST (*mop-prio-peek-min prio*) :: ⟨*A*⟩*mset-rel* →$_f$ ⟨*A*⟩*nres-rel*
where *IS-BELOW-ID A*
proof *goal-cases*
case *1*
hence [*param*]: (*prio*,*prio*)∈*A* → *Id*
by (*auto simp*: *IS-BELOW-ID-def*)
show *?case*
apply (*rule fref-ncI*)
apply *parametricity*
by *auto*
qed
end

3.4.2 Patterns

lemma [*def-pat-rules*]:
mop-prio-pop-min$*prio* ≡ *UNPROTECT* (*mop-prio-pop-min prio*)
mop-prio-peek-min$*prio* ≡ *UNPROTECT* (*mop-prio-peek-min prio*)
by *auto*

end

3.5 Multisets by Lists

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

3.5.1 Abstract Operations

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

lemma *lms-empty-aref*: ([],*op-mset-empty*) ∈ *list-mset-rel*
unfolding *list-mset-rel-def* **by** (*auto simp*: *in-br-conv*)

lemma *lms-is-empty-aref*: (*is-Nil*,*op-mset-is-empty*) ∈ *list-mset-rel* → *bool-rel*
unfolding *list-mset-rel-def* **by** (*auto simp*: *in-br-conv split*: *list.splits*)

lemma *lms-insert-aref*: ((#), *op-mset-insert*) ∈ *Id* → *list-mset-rel* → *list-mset-rel*
unfolding *list-mset-rel-def* **by** (*auto simp*: *in-br-conv*)

lemma *lms-union-aref*: ((@), *op-mset-plus*) ∈ *list-mset-rel* → *list-mset-rel* → *list-mset-rel*
unfolding *list-mset-rel-def* **by** (*auto simp*: *in-br-conv*)

lemma *lms-pick-aref*: $(\lambda x\#l \Rightarrow RETURN\ (x,l),\ mop\text{-}mset\text{-}pick) \in list\text{-}mset\text{-}rel \rightarrow \langle Id \times_r list\text{-}mset\text{-}rel\rangle nres\text{-}rel$
unfolding *list-mset-rel-def mop-mset-pick-alt*[*abs-def*]
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*)
done

definition *list-contains x l* $\equiv$ *list-ex* ((=) *x*) *l*
lemma *lms-contains-aref*: (*list-contains*, *op-mset-contains*) $\in Id \rightarrow list\text{-}mset\text{-}rel \rightarrow bool\text{-}rel$
unfolding *list-mset-rel-def list-contains-def*[*abs-def*]
by (*auto simp*: *in-br-conv list-ex-iff in-multiset-in-set*)

lemma *lms-remove-aref*: (*remove1*, *op-mset-delete*) $\in Id \rightarrow list\text{-}mset\text{-}rel \rightarrow list\text{-}mset\text{-}rel$
unfolding *list-mset-rel-def* **by** (*auto simp*: *in-br-conv*)

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

lemma *mset-list-count*[*simp*]: *list-count x ys* = *count* (*mset ys*) *x*
by (*induction ys*) *auto*

lemma *lms-count-aref*: (*list-count*, *op-mset-count*) $\in Id \rightarrow list\text{-}mset\text{-}rel \rightarrow nat\text{-}rel$

unfolding *list-mset-rel-def* **by** (*auto simp*: *in-br-conv*)

lemma *lms-minus-aref*: (*minus-list-mset*,*op-mset-minus*) $\in list\text{-}mset\text{-}rel \rightarrow list\text{-}mset\text{-}rel \rightarrow list\text{-}mset\text{-}rel$
unfolding *list-mset-rel-def* **by** (*auto simp*: *in-br-conv*)

3.5.2 Declaration of Implementations

definition *list-mset-assn A* $\equiv$ *pure* (*list-mset-rel O* $\langle$*the-pure A*$\rangle$*mset-rel*)
declare *list-mset-assn-def*[*symmetric*,*fcomp-norm-unfold*]
lemma [*safe-constraint-rules*]: *is-pure* (*list-mset-assn A*) **unfolding** *list-mset-assn-def* **by** *simp*

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

definition [*simp*]: *op-list-mset-empty* $\equiv$ *op-mset-empty*
lemma *lms-fold-custom-empty*:
{#} = *op-list-mset-empty*
op-mset-empty = *op-list-mset-empty*
by *auto*

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*] .
sepref-decl-impl *lms-insert*: *lms-insert-aref*[*sepref-param*] .
sepref-decl-impl *lms-union*: *lms-union-aref*[*sepref-param*] .

— Some extra work is required for nondetermistic ops
lemma *lms-pick-aref'*:
(λ*x*#*l* ⇒ *return* (*x*,*l*), *mop-mset-pick*) ∈ (*pure list-mset-rel*)^k →$_a$ *prod-assn id-assn* (*pure list-mset-rel*)
apply (*simp only*: *prod-assn-pure-conv*)
apply *sepref-to-hoare*
apply (*sep-auto simp*: *refine-pw-simps list-mset-rel-def in-br-conv algebra-simps eintros del*: *exI*)
done
sepref-decl-impl (*ismop*) *lms-pick*: *lms-pick-aref'* .
sepref-decl-impl *lms-contains*: *lms-contains-aref*[*sepref-param*] .
sepref-decl-impl *lms-remove*: *lms-remove-aref*[*sepref-param*] .
sepref-decl-impl *lms-count*: *lms-count-aref*[*sepref-param*] .
sepref-decl-impl *lms-minus*: *lms-minus-aref*[*sepref-param*] .

end
theory *IICF-List-MsetO*
imports *../Intf/IICF-Multiset*
begin

definition *lmso-assn A* ≡ *hr-comp* (*list-assn A*) (*br mset* (λ*-. True*))
lemmas [*fcomp-norm-unfold*] = *lmso-assn-def*[*symmetric*]

lemma *lmso-is-pure*[*safe-constraint-rules*]: *is-pure A* ⟹ *is-pure* (*lmso-assn A*)
unfolding *lmso-assn-def* **by** *safe-constraint*

lemma *lmso-empty-aref*: (*uncurry0* (*RETURN* []), *uncurry0* (*RETURN op-mset-empty*)) ∈ *unit-rel* →$_f$ ⟨*br mset* (λ*-. True*)⟩*nres-rel*
by (*auto intro*!: *frefI nres-relI simp*: *in-br-conv*)

lemma *lmso-is-empty-aref*: (*RETURN o List.null*, *RETURN o op-mset-is-empty*) ∈ *br mset* (λ*-. True*) →$_f$ ⟨*bool-rel*⟩*nres-rel*
by (*auto intro*!: *frefI nres-relI simp*: *in-br-conv split*: *list.split*)

lemma *lmso-insert-aref*: (*uncurry* (*RETURN oo* (#)), *uncurry* (*RETURN oo op-mset-insert*)) ∈ (*Id* ×$_r$ *br mset* (λ*-. True*)) →$_f$ ⟨*br mset* (λ*-. True*)⟩*nres-rel*
by (*auto intro*!: *frefI nres-relI simp*: *in-br-conv*)

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

lemma *hd-tl-opt*[*sepref-opt-simps*]: *hd-tl l* = (*case l of* (*x*#*xs*) ⇒ (*x*,*xs*) | - ⇒ *CODE-ABORT* (λ-. (*hd l*, *tl l*)))
by (*auto split*: *list.split*)

lemma *lmso-pick-aref*: (*RETURN o hd-tl*,*op-mset-pick*) ∈ [λ*m*. *m*≠{#}]$_f$ *br mset* (λ-. *True*) → ⟨*Id* ×$_r$ *br mset* (λ-. *True*)⟩*nres-rel*
by (*auto intro*!: *frefI nres-relI simp*: *in-br-conv pw-le-iff refine-pw-simps neq-Nil-conv algebra-simps*)

lemma *hd-tl-hnr*: (*return o hd-tl*,*RETURN o hd-tl*) ∈ [λ*l*. ¬*is-Nil l*]$_a$ (*list-assn A*)^d → *prod-assn A* (*list-assn A*)
apply *sepref-to-hoare*
subgoal for *l li* **by** (*cases l*; *cases li*; *sep-auto*)
done

sepref-decl-impl (*no-register*) *lmso-empty*: *hn-Nil*[*to-hfref*] **uses** *lmso-empty-aref*
.
definition [*simp*]: *op-lmso-empty* ≡ *op-mset-empty*
sepref-register *op-lmso-empty*
lemma *lmso-fold-custom-empty*:
{#} = *op-lmso-empty*
op-mset-empty = *op-lmso-empty*
mop-mset-empty = *RETURN op-lmso-empty*
by *auto*
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*
apply *sepref-to-hoare*
subgoal for *l li* **by** (*cases l*; *cases li*; *sep-auto*)
done

sepref-decl-impl *lmso-is-empty*: *list-null-hnr* **uses** *lmso-is-empty-aref* .

sepref-decl-impl *lmso-insert*: *hn-Cons*[*to-hfref*] **uses** *lmso-insert-aref* .

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*

$\rightarrow_f$ $\langle Id\rangle nres\text{-}rel$
by (*auto intro*!: *frefI nres-relI*)

sepref-decl-impl *lmso-pick*: *hd-tl-hnr*[*FCOMP lmso-pick-aref*] **uses** *op-mset-pick-dummy-param*
by *simp*
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*) ∈ $\langle A\rangle$*list-rel* → *A* → *nat-rel*
unfolding *index-def*[*abs-def*] *find-index-def*
apply (*subgoal-tac* (((=), (=)) ∈ *A* → *A* → *bool-rel*))
apply *parametricity*
by (*simp add*: *pres-eq-iff-svb*)

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*
)
unfolding *swap-def*
by *auto*

lemma *swap-set*[*simp*]: ⟦ *i* < *length l*; *j*<*length l* ⟧ ⟹ *set* (*swap l i j*) = *set l*
unfolding *swap-def*
by *auto*

lemma *swap-multiset*[*simp*]: ⟦ *i* < *length l*; *j*<*length l* ⟧ ⟹ *mset* (*swap l i j*) = *mset l*
unfolding *swap-def*
by (*auto simp*: *mset-swap*)

lemma *swap-length*[*simp*]: *length* (*swap l i j*) = *length l*
unfolding *swap-def*
by *auto*

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

unfolding *swap-def* **by** *auto*

lemma *distinct-swap*[*simp*]:
⟦*i*<*length l*; *j*<*length l*⟧ ⟹ *distinct* (*swap l i j*) = *distinct l*
unfolding *swap-def*
by *auto*

lemma *map-swap*: ⟦*i*<*length l*; *j*<*length l*⟧
⟹ *map f* (*swap l i j*) = *swap* (*map f l*) *i j*
unfolding *swap-def*
by (*auto simp add*: *map-update*)

lemma *swap-param*[*param*]: ⟦ *i*<*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*
unfolding *swap-def*
by *parametricity*

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*
apply *rule* **apply** *clarsimp*
unfolding *swap-def*
apply *parametricity*
by *simp-all*

lemma *param-list-null*[*param*]: (*List.null*,*List.null*) ∈ ⟨*A*⟩*list-rel* → *bool-rel*
proof −
have *1*: *List.null* = (λ[] ⇒ *True* | - ⇒ *False*)
apply (*rule ext*) **subgoal for** *l* **by** (*cases l*) *simp-all*
done
show *?thesis* **unfolding** *1* **by** *parametricity*
qed

3.5.4 Operations

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

$\times_r\ A \to \langle A\rangle$*list-rel* .
context notes [*simp*] = *List.null-iff* [*symmetric*] **and** [*simp del*] = *List.null-iff*
begin
sepref-decl-op *list-hd*: *hd* :: $[\lambda l.\ l{\neq}[]]_f\ \langle A\rangle$*list-rel* $\to A$.
sepref-decl-op *list-tl*: *tl* :: $[\lambda l.\ l{\neq}[]]_f\ \langle A\rangle$*list-rel* $\to \langle A\rangle$*list-rel* .
sepref-decl-op *list-last*: *last* :: $[\lambda l.\ l{\neq}[]]_f\ \langle A\rangle$*list-rel* $\to A$.
sepref-decl-op *list-butlast*: *butlast* :: $[\lambda l.\ l{\neq}[]]_f\ \langle A\rangle$*list-rel* $\to \langle A\rangle$*list-rel* .
end
sepref-decl-op *list-contains*: $\lambda x\ l.\ x{\in}$*set* $l :: A \to \langle A\rangle$*list-rel* $\to$ *bool-rel*
where *single-valued A single-valued* (A^{-1}) .
sepref-decl-op *list-swap*: *swap* :: $[\lambda((l,i),j).\ i{<}$*length* $l \wedge j{<}$*length* $l]_f\ (\langle A\rangle$*list-rel* $\times_r$ *nat-rel*$) \times_r$ *nat-rel* $\to \langle A\rangle$*list-rel* .
sepref-decl-op *list-rotate1*: *rotate1* :: $\langle A\rangle$*list-rel* $\to \langle A\rangle$*list-rel* .
sepref-decl-op *list-rev*: *rev* :: $\langle A\rangle$*list-rel* $\to \langle A\rangle$*list-rel* .
sepref-decl-op *list-index*: *index* :: $\langle A\rangle$*list-rel* $\to A \to$ *nat-rel*
where *single-valued A single-valued* (A^{-1}) .

3.5.5 Patterns

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

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-rel* $\Longrightarrow x'{\neq}[] \longleftrightarrow x{\neq}[]$ **by** *auto*
lemma *list-rel-pres-length*[*fcomp-prenorm-simps*]: $(x',x){\in}\langle A\rangle$*list-rel* $\Longrightarrow$ *length* x'

= *length x* **by** (*rule list-rel-imp-same-length*)

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-assn*^{k} →$_a$ *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*
    unfolding *op-custom-empty-def* **by** *simp-all*

  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
  }
  unfolding *mop-list-swap-def*
  by (*auto simp*: *pw-eq-iff refine-pw-simps swap-def*)

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* **by** (*auto simp*: *valid-def*)
  lemma *valid0*[*simp*]: *¬valid h 0* **by** (*auto simp*: *valid-def*)
  lemma *valid-glen*[*simp*]: *i>length h ⟹ ¬valid h i* **by** (*auto simp*: *valid-def*)

  lemma *valid-len*[*simp*]: *h≠[] ⟹ valid h* (*length h*) **by** (*auto simp*: *valid-def*)

  lemma *validI*: *0<i ⟹ i≤length h ⟹ valid h i*
    by (*auto simp*: *valid-def*)

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

Navigating the tree

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

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

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

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

  lemma *parent-left-id*[*simp*]: *parent* (*left i*) = *i*
    unfolding *parent-def left-def*
    by *auto*

  lemma *parent-right-id*[*simp*]: *parent* (*right i*) = *i*
    unfolding *parent-def right-def*

by *auto*

lemma *child-of-parentD*:
has-parent l i $\Longrightarrow$ *left* (*parent i*) = *i* $\vee$ *right* (*parent i*) = *i*
unfolding *parent-def left-def right-def valid-def*
by *auto*

lemma *rc-imp-lc*: $[\![$*valid h i*; *has-right h i*$]\!] \Longrightarrow$ *has-left h i*
by (*auto simp*: *valid-def left-def right-def*)

lemma *plr-corner-cases*[*simp*]:
assumes *0<i*
shows
i≠parent i
i≠left i
i≠right i
parent i $\neq$ *i*
left i $\neq$ *i*
right i $\neq$ *i*
using *assms*
by (*auto simp*: *parent-def left-def right-def*)

lemma *i-eq-parent-conv*[*simp*]: *i=parent i* $\longleftrightarrow$ *i=0*
by (*auto simp*: *parent-def*)

Heap Property

The heap property states, that every node's priority is greater or equal to its parent's priority

definition *heap-invar* :: *'a heap* $\Rightarrow$ *bool*
where *heap-invar l*
$\equiv \forall$ *i*. *valid l i* $\longrightarrow$ *has-parent l i* $\longrightarrow$ *pparent l i* $\leq$ *prio-of l i*

definition *heap-rel1* $\equiv$ *br* α *heap-invar*

lemma *heap-invar-empty*[*simp*]: *heap-invar* []
by (*auto simp*: *heap-invar-def*)

function *heap-induction-scheme* :: *nat* $\Rightarrow$ *unit* **where**
heap-induction-scheme i = (
if i>1 then heap-induction-scheme (*parent i*) *else* ())
by *pat-completeness auto*

termination
apply (*relation less-than*)
apply (*auto simp*: *parent-def*)
done

```
lemma
  heap-parent-le: ⟦heap-invar l; valid l i; has-parent l i⟧
    ⟹ pparent l i ≤ prio-of l i
  unfolding heap-invar-def
  by auto

lemma heap-min-prop:
  assumes H: heap-invar h
  assumes V: valid h i
  shows prio-of h (Suc 0) ≤ prio-of h i
proof (cases i>1)
  case False with V show ?thesis
    by (auto simp: valid-def intro: Suc-lessI)
next
  case True
  from V have i≤length h valid h (Suc 0) by (auto simp: valid-def)
  with True show ?thesis
    apply (induction i rule: heap-induction-scheme.induct)
    apply (rename-tac i)
    apply (case-tac parent i = Suc 0)
    apply (rule order-trans[rotated])
    apply (rule heap-parent-le[OF H])
    apply (auto simp: valid-def) [3]

    apply (rule order-trans)
    apply (rprems)
    apply (auto simp: parent-def) [4]
    apply (rule heap-parent-le[OF H])
    apply (auto simp: valid-def parent-def)
    done
qed
```

Obviously, the heap property can also be stated in terms of children, i.e., each node's priority is smaller or equal to it's children's priority.

```
definition children-ge h p i ≡
  (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
  unfolding heap-invar-def
  unfolding heap-invar'-def children-ge-def
  apply rule
  apply auto []
  apply clarsimp
  apply (frule child-of-parentD)
  apply auto []
```

done

3.6.2 Basic Operations

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

Val-Of

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

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

Update

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

lemma *update-valid*[*simp*]: *valid* (*update h i v*) *j* ⟷ *valid h j*
by (*auto simp*: *update-def valid-def*)

lemma *val-of-update*[*simp*]: ⟦*update-pre h i v*; *valid h j*⟧ ⟹ *val-of* (*update h i v*) *j* = (
if i=*j then v else val-of h j*)
unfolding *update-def val-of-def*
by (*auto simp*: *nth-list-update valid-def*)

lemma *length-update*[*simp*]: *length* (*update l i v*) = *length l*
by (*auto simp*: *update-def*)

Exchange

Exchange two elements

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

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

lemma exch-op-alt: exch-op l i j = do {
  vi ← val-of-op l i;
  vj ← val-of-op l j;
  l ← update-op l i vj;
  l ← update-op l j vi;
  RETURN l }
  by (auto simp: exch-op-def swap-def val-of-op-def update-op-def
    pw-eq-iff refine-pw-simps)

lemma exch-op-correct[refine-vcg]:
  exch-pre l i j ⟹ exch-op l i j ≤ SPEC (λr. r = exch l i j)
  unfolding exch-op-def
  apply refine-vcg
  apply (auto simp: exch-def valid-def)
  done

lemma valid-exch[simp]: valid (exch l i j) k = valid l k
  unfolding exch-def by (auto simp: valid-def)

lemma val-of-exch[simp]: ⟦valid l i; valid l j; valid l k⟧ ⟹
  val-of (exch l i j) k = (
    if k=i then val-of l j
    else if k=j then val-of l i
    else val-of l k
  )
  unfolding exch-def val-of-def valid-def
  by (auto)

lemma exch-eq[simp]: exch h i i = h
  by (auto simp: exch-def)

lemma α-exch[simp]: ⟦valid l i; valid l j⟧
  ⟹ α (exch l i j) = α l
  unfolding exch-def valid-def
  by (auto)
```

lemma *length-exch*[*simp*]: *length* (*exch l i j*) = *length l*
by (*auto simp*: *exch-def*)

Butlast

Remove last element

abbreviation *butlast-pre l* ≡ *l*≠[]
definition *butlast-op* :: *'a heap* ⇒ *'a heap nres*
where *butlast-op l* ≡ *mop-list-butlast l*
lemma *butlast-op-correct*[*refine-vcg*]:
butlast-pre l ⟹ *butlast-op l* ≤ *SPEC* (λ*r*. *r* = *butlast l*)
unfolding *butlast-op-def* **by** (*refine-vcg*; *auto*)

lemma *valid-butlast-conv*[*simp*]: *valid* (*butlast h*) *i* ⟷ *valid h i* ∧ *i* < *length h*
by (*auto simp*: *valid-def*)

lemma *valid-butlast*: *valid* (*butlast h*) *i* ⟹ *valid h i*
by (*cases h rule*: *rev-cases*) (*auto simp*: *valid-def*)

lemma *val-of-butlast*[*simp*]: ⟦*valid h i*; *i*<*length h*⟧
⟹ *val-of* (*butlast h*) *i* = *val-of h i*
by (*auto simp*: *valid-def val-of-def nth-butlast*)

lemma *val-of-butlast'*[*simp*]:
valid (*butlast h*) *i* ⟹ *val-of* (*butlast h*) *i* = *val-of h i*
by (*cases h rule*: *rev-cases*) (*auto simp*: *valid-def val-of-def nth-append*)

lemma α*-butlast*[*simp*]: ⟦ *length h* ≠ *0* ⟧
⟹ α (*butlast h*) = α *h* − {# *val-of h* (*length h*)#}
apply (*cases h rule*: *rev-cases*)
apply (*auto simp*: *val-of-def*)
done

lemma *heap-invar-butlast*[*simp*]: *heap-invar h* ⟹ *heap-invar* (*butlast h*)
apply (*cases h* = [])
apply *simp*
apply (*auto simp*: *heap-invar-def dest*: *valid-butlast*)
done

Append

definition *append-op* :: *'a heap* ⇒ *'a* ⇒ *'a heap nres*
where *append-op l v* ≡ *mop-list-append l v*
lemma *append-op-correct*[*refine-vcg*]:
append-op l v ≤ *SPEC* (λ*r*. *r* = *l*@[*v*])
unfolding *append-op-def* **by** (*refine-vcg*; *auto*)

lemma *valid-append*[*simp*]: *valid* (*l*@[*v*]) *i* ⟷ *valid l i* ∨ *i* = *length l* + *1*
by (*auto simp*: *valid-def*)

lemma *val-of-append*[*simp*]: *valid* (*l*@[*v*]) *i* ⟹
val-of (*l*@[*v*]) *i* = (*if valid l i then val-of l i else v*)
unfolding *valid-def val-of-def* **by** (*auto simp*: *nth-append*)

lemma *α-append*[*simp*]: *α* (*l*@[*v*]) = *α l* + {#*v*#}
by *auto*

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*
unfolding *swim-invar-def* **by** *simp*

lemma *swim-invar-exit1*: $\neg$*has-parent h i* $\Longrightarrow$ *swim-invar h i* $\Longrightarrow$ *heap-invar h*
unfolding *heap-invar-def swim-invar-def* **by** *auto*

lemma *swim-invar-exit2*: *pparent h i* $\leq$ *prio-of h i* $\Longrightarrow$ *swim-invar h i* $\Longrightarrow$ *heap-invar h*
unfolding *heap-invar-def swim-invar-def* **by** *auto*

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*)
unfolding *swim-invar-def*
apply *safe*
apply (*simp add*: *h'-def HPI*)

using *HPI VIOLATED INV*
unfolding *swim-invar-def h'-def*
apply *auto* []

using *HPI VIOLATED INV*
unfolding *swim-invar-def h'-def*
apply *auto*
by (*metis order-trans*)

lemma *swim-invar-decr*:
assumes *INV*: *heap-invar h*
assumes *V*: *valid h i*
assumes *DECR*: *prio v* $\leq$ *prio-of h i*
shows *swim-invar* (*update h i v*) *i*
using *INV V DECR*
apply (*auto simp*: *swim-invar-def heap-invar-def intro*: *dual-order.trans*)
done

lemma *swim-op-correct*[*refine-vcg*]:
⟦*swim-invar h i*⟧ $\Longrightarrow$
swim-op h i $\leq$ *SPEC* ($\lambda h'$. α *h'* = α *h* $\wedge$ *heap-invar h'* $\wedge$ *length h'* = *length h*)
unfolding *swim-op-def*
using [[*goals-limit* = *1*]]
apply (*refine-vcg RECT-rule*[**where**
pre=λ(*hh*,*i*).
swim-invar hh i
$\wedge$ α *hh* = α *h*
$\wedge$ *length hh* = *length h* **and**
V = *inv-image less-than snd*
])

apply (*auto*) []
apply (*auto*) []
apply (*auto*) []
apply (*auto*) []
apply (*auto simp*: *swim-invar-valid*) []
apply (*auto*) []
apply (*auto*) []
apply (*auto*) []

apply *rprems*
  apply (*auto simp*: *swim-invar-pres*) []
  apply (*auto simp*: *parent-def valid-def*) []

apply (*auto*) []
apply (*auto simp*: *swim-invar-exit2*) []
apply (*auto*) []
apply (*auto*) []
apply (*auto simp*: *swim-invar-exit1*) []
apply (*auto*) []
done

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 (λ*sink* (*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
  unfolding sink-invar-def by auto

lemma sink-invar-exit: ⟦sink-invar l i; children-ge l (prio-of l i) i⟧
  ⟹ heap-invar' l
  unfolding heap-invar'-def sink-invar-def
  by auto

lemma sink-aux1: ¬ (2*i ≤ length h) ⟹ ¬has-left h i ∧ ¬has-right h i
  unfolding valid-def left-def right-def by auto

lemma sink-invar-pres1:
  assumes sink-invar h i
  assumes has-left h i has-right h i
  assumes prio-of h i ≥ pleft h i
  assumes pleft h i ≥ pright h i
  shows sink-invar (exch h i (right i)) (right i)
  using assms
  unfolding sink-invar-def
  apply auto
  apply (auto simp: children-ge-def)
  done

lemma sink-invar-pres2:
  assumes sink-invar h i
  assumes has-left h i has-right h i
  assumes prio-of h i ≥ pleft h i
  assumes pleft h i ≤ pright h i
  shows sink-invar (exch h i (left i)) (left i)
  using assms
  unfolding sink-invar-def
  apply auto
  apply (auto simp: children-ge-def)
  done

lemma sink-invar-pres3:
```

assumes *sink-invar h i*
assumes *has-left h i has-right h i*
assumes *prio-of h i* $\geq$ *pright h i*
assumes *pleft h i* $\leq$ *pright h i*
shows *sink-invar* (*exch h i* (*left i*)) (*left i*)
using *assms*
unfolding *sink-invar-def*
apply *auto*
apply (*auto simp*: *children-ge-def*)
done

lemma *sink-invar-pres4*:
assumes *sink-invar h i*
assumes *has-left h i has-right h i*
assumes *prio-of h i* $\geq$ *pright h i*
assumes *pleft h i* $\geq$ *pright h i*
shows *sink-invar* (*exch h i* (*right i*)) (*right i*)
using *assms*
unfolding *sink-invar-def*
apply *auto*
apply (*auto simp*: *children-ge-def*)
done

lemma *sink-invar-pres5*:
assumes *sink-invar h i*
assumes *has-left h i* $\neg$*has-right h i*
assumes *prio-of h i* $\geq$ *pleft h i*
shows *sink-invar* (*exch h i* (*left i*)) (*left i*)
using *assms*
unfolding *sink-invar-def*
apply *auto*
apply (*auto simp*: *children-ge-def*)
done

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

lemma *sink-invar-incr*:
assumes *INV*: *heap-invar h*
assumes *V*: *valid h i*
assumes *INCR*: *prio v* $\geq$ *prio-of h i*
shows *sink-invar* (*update h i v*) *i*
using *INV V INCR*
apply (*auto simp*: *sink-invar-def*)

apply (*auto simp*: *children-ge-def heap-invar-def*) []
apply (*auto simp*: *children-ge-def heap-invar-def intro*: *order-trans*) []
apply (*frule spec*[**where** *x*=*left i*])
apply *auto* []
apply (*frule spec*[**where** *x*=*right i*])
apply *auto* []
done

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

unfolding *sink-op-def heap-eq-heap'*
using [[*goals-limit* = *1*]]

apply (*refine-vcg RECT-rule*[**where**
pre=λ(*hh*,*i*). *sink-invar hh i* ∧ α *hh* = α *h* ∧ *length hh* = *length h* **and**
V = *measure* (λ(*l*,*i*). *length l* − *i*)
])

apply (*auto*) []
apply (*auto*) []
apply (*auto*) []
apply (*auto*) []
apply (*auto simp*: *sink-invar-valid*) []
apply (*auto simp*: *valid-def left-def right-def*) []

apply *rprems*
apply (*auto intro*: *sink-invar-pres*) []
apply (*auto simp*: *valid-def left-def right-def*) []

apply *rprems*
apply (*auto intro*: *sink-invar-pres*) []
apply (*auto simp*: *valid-def left-def right-def*) []

apply (*auto*) []

apply *clarsimp*
apply (*rule sink-invar-exit*, *assumption*) []
apply (*auto simp*: *children-ge-def*) []

apply (*auto*) []

apply *rprems*
apply (*auto intro*: *sink-invar-pres*) []
apply (*auto simp*: *valid-def left-def right-def*) []

apply (*auto*) []

apply *clarsimp*
apply (*rule sink-invar-exit*, *assumption*) []
apply (*auto simp*: *children-ge-def*) []

apply (*auto*) []

apply (*auto*) []

apply *clarsimp*
apply (*rule sink-invar-exit*, *assumption*) []
apply (*auto simp*: *children-ge-def*) []

apply (*auto*) []
done

lemma *sink-op-swim-rule*:
swim-invar h i $\Longrightarrow$ *sink-op h i* $\leq$ *SPEC* ($\lambda h'$. h'=h)
apply (*frule swim-invar-valid*)
unfolding *sink-op-def*
apply (*subst RECT-unfold*, *refine-mono*)
apply (*fold sink-op-def*)
apply *refine-vcg*
apply (*simp-all*)
apply (*auto simp add*: *valid-def left-def right-def dest*: *swim-invar-valid*) []
apply (*auto simp*: *swim-invar-def*) []
apply (*auto simp*: *swim-invar-def*) []
apply (*auto simp*: *swim-invar-def*) []
apply (*auto simp*: *swim-invar-def*) []
apply (*auto simp*: *swim-invar-def*) []
apply (*auto simp*: *swim-invar-def*) []
done

definition *sink-op-opt*
— Sink operation as presented in Sedgewick et al. Algs4 reference implementation
where
sink-op-opt h k $\equiv$ *RECT* (λD (*h*,*k*). *do* {
ASSERT (*k*>*0* $\wedge$ *k*$\leq$*length h*);
let len = *length h*;
if (*2*∗*k* $\leq$ *len*) *then do* {
let j = *2*∗*k*;
pj $\leftarrow$ *prio-of-op h j*;

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

```
      pj ← 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
  unfolding sink-op-opt-def sink-op-def
  apply (fo-rule arg-cong fun-cong)+
  apply (intro ext)
  unfolding sink-op-def[symmetric]
  apply (simp cong: if-cong split del: if-split add: Let-def)

  apply (auto simp: valid-def left-def right-def prio-of-op-def val-of-op-def
    val-of-def less-imp-diff-less ASSERT-same-eq-conv nz-le-conv-less) []
  done
```

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
  apply (cases rule: linear[of prio v prio-of h i, THEN disjE])
  apply (blast dest: swim-invar-decr[OF assms])
  apply (blast dest: sink-invar-incr[OF assms])
  done

lemma heap-invar-imp-swim-invar: ⟦heap-invar h; valid h i⟧ ⟹ swim-invar h
i
  unfolding heap-invar-def swim-invar-def
  by (auto intro: order-trans)
```

lemma *repair-correct*[*refine-vcg*]:
assumes *heap-invar h* **and** *valid h i*
shows *repair-op* (*update h i v*) *i* ≤ *SPEC* (λ*h'*.
heap-invar h' ∧ α *h'* = α (*update h i v*) ∧ *length h'* = *length h*)
apply (*rule update-sink-swim-cases*[*of h i v*, *OF assms*])
unfolding *repair-op-def*
apply (*refine-vcg sink-op-swim-rule*)
apply *auto* [*4*]
apply (*refine-vcg*)
using *assms*(*2*)
apply (*auto intro*: *heap-invar-imp-swim-invar simp*: *valid-def*) []
apply *auto* [*3*]
done

3.6.4 Operations

Empty

abbreviation (*input*) *empty* :: *'a heap* — The empty heap
where *empty* ≡ []
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*)
unfolding *empty-op-def* **apply** *refine-vcg* **by** *auto*

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* = {#})
unfolding *is-empty-op-def*
apply *refine-vcg* **by** *auto*

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*))
unfolding *swim-invar-def heap-invar-def valid-def parent-def val-of-def*

by (*fastforce simp*: *nth-append*)

lemma
(*insert-op*,*RETURN oo op-mset-insert*) ∈ *Id* → *heap-rel1* → ⟨*heap-rel1*⟩*nres-rel*
unfolding *insert-op-def*[*abs-def*] *heap-rel1-def o-def*
by *refine-vcg* (*auto simp*: *swim-invar-insert in-br-conv*)

lemma *insert-op-correct*:
heap-invar h ⟹ *insert-op v h* ≤ *SPEC* (λ*h'*. *heap-invar h'* ∧ α *h'* = α *h* + {#*v*#})
unfolding *insert-op-def*
by (*refine-vcg*) (*auto simp*: *swim-invar-insert*)
lemmas [*refine-vcg*] = *insert-op-correct*

Pop minimum element

definition *pop-min-op* :: *'a heap* ⇒ (*'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*
by (*auto simp*: *left-def*)

lemma *right-one-conv*[*simp*]: *right j* = *Suc 0* ⟷ *j*=*0*
by (*auto simp*: *right-def*)

lemma *parent-one-conv*[*simp*]: *parent* (*Suc 0*) = *0*
by (*auto simp*: *parent-def*)

lemma *sink-invar-init*:
assumes *I*: *heap-invar h*
assumes *NE*: *length h* > *1*
shows *sink-invar* (*butlast* (*exch h* (*Suc 0*) (*length h*))) (*Suc 0*)
proof −
from *NE* **have** *V*: *valid h* (*Suc 0*) *valid h* (*length h*)
apply −
apply (*auto simp*: *valid-def neq-Nil-conv*) []

```
    by (cases h) (auto simp: valid-def)

  show ?thesis using I
    unfolding heap-eq-heap' heap-invar'-def sink-invar-def
    apply (intro impI conjI allI)
    using NE apply (auto simp: V valid-butlast-conv) []
    apply (auto simp add: children-ge-def V NE valid-butlast-conv) []
    apply (auto simp add: children-ge-def V NE valid-butlast-conv) []
    done
qed

lemma in-set-conv-val: v ∈ set h ⟷ (∃ i. valid h i ∧ v = val-of h i)
  apply (rule iffI)
  apply (clarsimp simp add: valid-def val-of-def in-set-conv-nth)
  apply (rule-tac x=Suc i in exI; auto)
  apply (clarsimp simp add: valid-def val-of-def in-set-conv-nth)
  apply (rule-tac x=i − Suc 0 in exI; auto)
  done


lemma pop-min-op-correct:
  assumes heap-invar h α h ≠ {#}
  shows pop-min-op h ≤ SPEC (λ(v,h'). heap-invar h' ∧
    v ∈# α h ∧ α h' = α h − {#v#} ∧ (∀ v'∈set-mset (α h). prio v ≤ prio v'))
proof −
  note [simp del] = length-greater-0-conv
  note LG = length-greater-0-conv[symmetric]

  from assms show ?thesis
    unfolding pop-min-op-def
    apply refine-vcg
    apply (simp-all add: sink-invar-init LG)
    apply (auto simp: valid-def) []
    apply (cases h; auto simp: val-of-def) []
    apply (auto simp: in-set-conv-val simp: heap-min-prop) []
    apply auto []
    apply (cases h; auto simp: val-of-def) []
    apply auto []
    apply (cases h; auto simp: val-of-def) []
    done
qed

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

Peek minimum element

```
definition peek-min-op :: 'a heap ⇒ 'a nres where
  peek-min-op h ≡ do {
    ASSERT (heap-invar h);
```

ASSERT (*valid h 1*);
val-of-op h 1
}

lemma *peek-min-op-correct*:
assumes *heap-invar h* α *h* $\neq$ {#}
shows *peek-min-op h* $\leq$ *SPEC* (λ*v*.
v $\in$# α *h* $\wedge$ ($\forall$ *v'*$\in$*set-mset* (α *h*). *prio v* $\leq$ *prio v'*))
unfolding *peek-min-op-def*
apply *refine-vcg*
using *assms*
apply *clarsimp-all*
apply (*auto simp*: *valid-def*) []
apply (*cases h*; *auto simp*: *val-of-def*) []
apply (*auto simp*: *in-set-conv-val simp*: *heap-min-prop*) []
done

lemmas *peek-min-op-correct'*[*refine-vcg*] = *peek-min-op-correct*

3.6.5 Operations as Relator-Style Refinement

lemma *empty-op-refine*: (*empty-op*,*RETURN op-mset-empty*)$\in$$\langle$*heap-rel1*$\rangle$*nres-rel*
apply (*rule nres-relI*)
apply (*rule order-trans*)
apply (*rule empty-op-correct*)
apply (*auto simp*: *heap-rel1-def br-def pw-le-iff refine-pw-simps*)
done

lemma *is-empty-op-refine*: (*is-empty-op*,*RETURN o op-mset-is-empty*) $\in$ *heap-rel1* $\rightarrow$ $\langle$*bool-rel*$\rangle$*nres-rel*
apply (*intro nres-relI fun-relI*; *simp*)
apply *refine-vcg*
apply (*auto simp*: *heap-rel1-def br-def*)
done

lemma *insert-op-refine*: (*insert-op*,*RETURN oo op-mset-insert*) $\in$ *Id* $\rightarrow$ *heap-rel1* $\rightarrow$ $\langle$*heap-rel1*$\rangle$*nres-rel*
apply (*intro nres-relI fun-relI*; *simp*)
apply (*refine-vcg RETURN-as-SPEC-refine*)
apply (*auto simp*: *heap-rel1-def br-def pw-le-iff refine-pw-simps*)
done

lemma *pop-min-op-refine*:
(*pop-min-op*, *PR-CONST* (*mop-prio-pop-min prio*)) $\in$ *heap-rel1* $\rightarrow$ $\langle$*Id* $\times_r$ *heap-rel1*$\rangle$*nres-rel*
apply (*intro fun-relI nres-relI*)
unfolding *mop-prio-pop-min-def PR-CONST-def*
apply (*refine-vcg SPEC-refine*)
apply (*auto simp*: *heap-rel1-def br-def*)

done

lemma *peek-min-op-refine*:
(*peek-min-op*, *PR-CONST* (*mop-prio-peek-min prio*)) ∈ *heap-rel1* → ⟨*Id*⟩*nres-rel*
apply (*intro fun-relI nres-relI*)
unfolding *mop-prio-peek-min-def PR-CONST-def*
apply (*refine-vcg RES-refine*)
apply (*auto simp*: *heap-rel1-def br-def*)
done

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*
apply (*subst id-take-nth-drop*[*OF assms*(*1*)])
apply (*rule assms*(*2*))
apply (*subst id-take-nth-drop*[*OF assms*(*1*)])
apply *simp*
apply (*simp add*: *assms*(*1*))
done

private 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*
apply (*rule entails-preI*)
apply (*clarsimp*)
apply (*cases i* < *length l*; *cases i* < *length li*; (*sep-auto simp*: *list-update-beyond dest*!: *list-assn-aux-eqlen-simp*;*fail*)*?*)
apply (*erule id-take-nth-drop-rl*)
apply (*erule id-take-nth-drop-rl*)
apply (*sep-auto simp add*: *list-update-append*)
done

private lemma *list-set-hd-tl-aux*:
a ≠ [] ⟹ *list-assn R a c* ⟹$_A$ *R* (*hd a*) (*hd c*) ∗ *true*
a ≠ [] ⟹ *list-assn R a c* ⟹$_A$ *list-assn R* (*tl a*) (*tl c*) ∗ *true*
by (*cases c*; *cases a*; *sep-auto*; *fail*)+

private 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*
by (*cases c rule*: *rev-cases*; *cases a rule*: *rev-cases*; *sep-auto*; *fail*)+

private 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*
by (*auto simp*: *swap-def list-update-append nth-append*)

private 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*
apply (*subst list-assn-aux-len*; *clarsimp*)
apply (*cases i*=*j*; (*sep-auto*;*fail*)*?*)
apply (*rule id-take-nth-drop-rl*[**where** *l*=*l* **and** *i*=*i*]; *simp*)
apply (*rule id-take-nth-drop-rl*[**where** *l*=*l* **and** *i*=*j*]; *simp*)
apply (*erule list-match-lel-lel*; *simp*)
apply (*split-list-according li l*; *sep-auto*)
apply (*split-list-according li l*; *sep-auto*)
done

private lemma *list-rotate1-aux*: *list-assn A a c* $\Longrightarrow_A$ *list-assn A* (*rotate1 a*) (*rotate1 c*) $*$ *true*
by (*cases a*; *cases c*; *sep-auto*)

private lemma *list-rev-aux*: *list-assn A a c* $\Longrightarrow_A$ *list-assn A* (*rev a*) (*rev c*) $*$ *true*
apply (*subst list-assn-aux-len*; *clarsimp*)
apply (*induction rule*: *list-induct2*)
apply *sep-auto*
apply *sep-auto*
apply (*erule ent-frame-fwd*, *frame-inference*)
apply *sep-auto*
done

lemma *mod-starE*:
assumes *h* $\models$ *A*$*$*B*
obtains *h1 h2* **where** *h1*$\models$*A h2*$\models$*B*
using *assms* **by** (*auto simp*: *mod-star-conv*)

private lemma *CONSTRAINT-is-pureE*:
assumes *CONSTRAINT is-pure A*
obtains *R* **where** *A*=*pure R*
using *assms* **by** (*auto simp*: *is-pure-conv*)

private 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*)) ∈ *unit-assn*^{k} →$_a$ (*list-assn A*) **by** *solve*
lemma *HOL-list-is-empty-hnr*[*sepref-fr-rules*]: (*return* ∘ *op-list-is-empty*, *RETURN* ∘ *op-list-is-empty*) ∈ (*list-assn A*)^k →$_a$ *bool-assn* **by** *solve*
lemma *HOL-list-prepend-hnr*[*sepref-fr-rules*]: (*uncurry* (*return* ∘∘ *op-list-prepend*), *uncurry* (*RETURN* ∘∘ *op-list-prepend*)) ∈ A^d *a* (*list-assn A*)^d →$_a$ *list-assn A* **by** *solve*
lemma *HOL-list-append-hnr*[*sepref-fr-rules*]: (*uncurry* (*return* ∘∘ *op-list-append*), *uncurry* (*RETURN* ∘∘ *op-list-append*)) ∈ (*list-assn A*)^d *a* A^d →$_a$ *list-assn A* **by** *solve*
lemma *HOL-list-concat-hnr*[*sepref-fr-rules*]: (*uncurry* (*return* ∘∘ *op-list-concat*), *uncurry* (*RETURN* ∘∘ *op-list-concat*)) ∈ (*list-assn A*)^d *a* (*list-assn A*)^d →$_a$ *list-assn A* **by** *solve*
lemma *HOL-list-length-hnr*[*sepref-fr-rules*]: (*return* ∘ *op-list-length*, *RETURN* ∘ *op-list-length*) ∈ (*list-assn A*)^k →$_a$ *nat-assn* **by** *solve*
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* **by** *solve*
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* **by** *solve*
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* **by** *solve*
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* **by** *solve*
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* **by** *solve*
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* **by** *solve*
lemma *HOL-list-rotate1-hnr*[*sepref-fr-rules*]: (*return* ∘ *op-list-rotate1*, *RETURN* ∘ *op-list-rotate1*) ∈ (*list-assn A*)^d →$_a$ *list-assn A* **by** *solve*
lemma *HOL-list-rev-hnr*[*sepref-fr-rules*]: (*return* ∘ *op-list-rev*, *RETURN* ∘ *op-list-rev*) ∈ (*list-assn A*)^d →$_a$ *list-assn A* **by** *solve*

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 $\rightarrow_a$ *list-assn A* **by** *solve*
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* **by** *solve*

private lemma *bool-by-paramE*: ⟦ *a*; (*a*,*b*)∈*Id* ⟧ ⟹ *b* **by** *simp*
private lemma *bool-by-paramE'*: ⟦ *a*; (*b*,*a*)∈*Id* ⟧ ⟹ *b* **by** *simp*

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 $\rightarrow_a$ *bool-assn*
apply *solve-dbg*
apply (*erule bool-by-paramE*[**where** *a*=-∈*set* -]) **apply** *parametricity*
apply (*erule bool-by-paramE'*[**where** *a*=-∈*set* -]) **apply** *parametricity*
done

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*
  apply *unfold-locales*
  apply (*sep-auto intro*!: *hfrefI hn-refineI*)
  by *simp*

schematic-goal
  notes [*sepref-fr-rules*] = *HOL-list-empty-hnr*
  shows
  hn-refine (*emp*) (*?c*::*?'c Heap*) *?Γ' ?R* (*do* {
    x ← *RETURN* [*1*,*2*,*3*::*nat*];
    let x2 = *op-list-append x 5*;
    ASSERT (*length x* = *4*);
    let x = *op-list-swap x 1 2*;
    x ← *mop-list-swap x 1 2*;
    RETURN (*x*@*x*)
  })
    by *sepref*

end
theory *IICF-Array-List*
imports
  ../Intf/IICF-List
  Separation-Logic-Imperative-HOL.Array-Blit
begin

  type-synonym *'a array-list* = *'a Heap.array* × *nat*

  definition *is-array-list l* ≡ λ(*a*,*n*). ∃*A l'*. *a* ↦*a l'* ∗ ↑(*n* ≤ *length l'* ∧ *l* = *take n l'* ∧ *length l'*>*0*)

  lemma *is-array-list-prec*[*safe-constraint-rules*]: *precise is-array-list*
    unfolding *is-array-list-def*[*abs-def*]
    apply(*rule preciseI*)
    apply(*simp split*: *prod.splits*)
   using *preciseD snga-prec* **by** *fastforce*

  definition *initial-capacity* ≡ *16*::*nat*

```
definition minimum-capacity ≡ 16::nat

definition arl-empty ≡ do {
  a ← Array.new initial-capacity default;
  return (a,0)
}

definition arl-empty-sz init-cap ≡ do {
  a ← Array.new (max init-cap minimum-capacity) default;
  return (a,0)
}

definition arl-append ≡ 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 []>
    by (sep-auto simp: arl-empty-def is-array-list-def initial-capacity-def)

  lemma arl-empty-sz-rule[sep-heap-rules]: < emp > arl-empty-sz N <is-array-list
[]>
    by (sep-auto simp: arl-empty-sz-def is-array-list-def minimum-capacity-def)

  lemma arl-copy-rule[sep-heap-rules]: < is-array-list l a > arl-copy a <λr. is-array-list
l a ∗ is-array-list l r>
    by (sep-auto simp: arl-copy-def is-array-list-def)

  lemma arl-append-rule[sep-heap-rules]:
    < is-array-list l a >
      arl-append a x
    <λa. is-array-list (l@[x]) a >t
    by (sep-auto
      simp: arl-append-def is-array-list-def take-update-last neq-Nil-conv
      split: prod.splits nat.split)

  lemma arl-length-rule[sep-heap-rules]:
    <is-array-list l a>
      arl-length a
    <λr. is-array-list l a ∗ ↑(r=length l)>
    by (sep-auto simp: arl-length-def is-array-list-def)

  lemma arl-is-empty-rule[sep-heap-rules]:
    <is-array-list l a>
      arl-is-empty a
    <λr. is-array-list l a ∗ ↑(r⟷(l=[]))>
    by (sep-auto simp: arl-is-empty-def is-array-list-def)

  lemma arl-last-rule[sep-heap-rules]:
    l≠[] ⟹
    <is-array-list l a>
      arl-last a
    <λr. is-array-list l a ∗ ↑(r=last l)>
    by (sep-auto simp: arl-last-def is-array-list-def last-take-nth-conv)
```

lemma *arl-butlast-rule*[*sep-heap-rules*]:
l≠[] ⟹
<*is-array-list l a*>
arl-butlast a
<*is-array-list* (*butlast l*)>$_t$
proof −
assume [*simp*]: *l*≠[]

have [*simp*]: ⋀*x*. *min* (*x*−*Suc 0*) ((*x*−*Suc 0*)∗*2*) = *x*−*Suc 0* **by** *auto*

show *?thesis*
by (*sep-auto*
split: *prod.splits*
simp: *arl-butlast-def is-array-list-def butlast-take minimum-capacity-def*)
qed

lemma *arl-get-rule*[*sep-heap-rules*]:
i<*length l* ⟹
<*is-array-list l a*>
arl-get a i
<λ*r*. *is-array-list l a* ∗ ↑(*r*=*l*!*i*)>
by (*sep-auto simp*: *arl-get-def is-array-list-def split*: *prod.split*)

lemma *arl-set-rule*[*sep-heap-rules*]:
i<*length l* ⟹
<*is-array-list l a*>
arl-set a i x
<*is-array-list* (*l*[*i*:=*x*])>
by (*sep-auto simp*: *arl-set-def is-array-list-def split*: *prod.split*)

definition *arl-assn A* ≡ *hr-comp is-array-list* (⟨*the-pure A*⟩*list-rel*)
lemmas [*safe-constraint-rules*] = *CN-FALSEI*[*of is-pure arl-assn A* **for** *A*]

lemma *arl-assn-comp*: *is-pure A* ⟹ *hr-comp* (*arl-assn A*) (⟨*B*⟩*list-rel*) = *arl-assn* (*hr-comp A B*)
unfolding *arl-assn-def*
by (*auto simp*: *hr-comp-the-pure hr-comp-assoc list-rel-compp*)

lemma *arl-assn-comp′*: *hr-comp* (*arl-assn id-assn*) (⟨*B*⟩*list-rel*) = *arl-assn* (*pure B*)
by (*simp add*: *arl-assn-comp*)

context
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*
by *sep-auto*
sepref-decl-impl (*no-register*) *arl-empty*: *arl-empty-hnr-aux* .

lemma *arl-empty-sz-hnr-aux*: (*uncurry0* (*arl-empty-sz N*),*uncurry0* (*RETURN op-list-empty*)) ∈ *unit-assn*^{k} →$_a$ *is-array-list*
by *sep-auto*
sepref-decl-impl (*no-register*) *arl-empty-sz*: *arl-empty-sz-hnr-aux* .

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*
by *sep-auto*
sepref-decl-impl *arl-copy*: *arl-copy-hnr-aux* .

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*
by *sep-auto*
sepref-decl-impl *arl-append*: *arl-append-hnr-aux* .

lemma *arl-length-hnr-aux*: (*arl-length*,*RETURN o op-list-length*) ∈ *is-array-list*^{k} →$_a$ *nat-assn*
by *sep-auto*
sepref-decl-impl *arl-length*: *arl-length-hnr-aux* .

lemma *arl-is-empty-hnr-aux*: (*arl-is-empty*,*RETURN o op-list-is-empty*) ∈ *is-array-list*^{k} →$_a$ *bool-assn*
by *sep-auto*
sepref-decl-impl *arl-is-empty*: *arl-is-empty-hnr-aux* .

lemma *arl-last-hnr-aux*: (*arl-last*,*RETURN o op-list-last*) ∈ [*pre-list-last*]$_a$ *is-array-list*^{k} → *id-assn*
by *sep-auto*
sepref-decl-impl *arl-last*: *arl-last-hnr-aux* .

lemma *arl-butlast-hnr-aux*: (*arl-butlast*,*RETURN o op-list-butlast*) ∈ [*pre-list-butlast*]$_a$ *is-array-list*^{d} → *is-array-list*
by *sep-auto*
sepref-decl-impl *arl-butlast*: *arl-butlast-hnr-aux* .

lemma *arl-get-hnr-aux*: (*uncurry arl-get*,*uncurry* (*RETURN oo op-list-get*)) ∈ [λ(*l*,*i*). *i*<*length l*]$_a$ (*is-array-list*^{k} ∗$_a$ *nat-assn*^{k}) → *id-assn*
by *sep-auto*
sepref-decl-impl *arl-get*: *arl-get-hnr-aux* .

lemma *arl-set-hnr-aux*: (*uncurry2 arl-set*,*uncurry2* (*RETURN ooo op-list-set*)) ∈ [λ((*l*,*i*),-). *i*<*length l*]$_a$ (*is-array-list*^{d} *$*_a$ *nat-assn*^{k} *$*_a$ *id-assn*^{k}) → *is-array-list*
by *sep-auto*
sepref-decl-impl *arl-set*: *arl-set-hnr-aux* .

sepref-definition *arl-swap* **is** *uncurry2 mop-list-swap* :: ((*arl-assn id-assn*)^d *$*_a$ *nat-assn*^{k} *$*_a$ *nat-assn*^{k} →$_a$ *arl-assn id-assn*)
unfolding *gen-mop-list-swap*[*abs-def*]
by *sepref*
sepref-decl-impl (*ismop*) *arl-swap*: *arl-swap.refine* .
end

interpretation *arl*: *list-custom-empty arl-assn A arl-empty op-arl-empty*
apply *unfold-locales*
apply (*rule arl-empty-hnr*)
by (*auto simp*: *op-arl-empty-def*)

lemma [*def-pat-rules*]: *op-arl-empty-sz*$*N* ≡ *UNPROTECT* (*op-arl-empty-sz N*)
by *simp*
interpretation *arl-sz*: *list-custom-empty arl-assn A arl-empty-sz N PR-CONST* (*op-arl-empty-sz N*)
apply *unfold-locales*
apply (*rule arl-empty-sz-hnr*)
by (*auto simp*: *op-arl-empty-sz-def*)

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* .
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* **.**

  abbreviation *rel* ≡ *arl-assn id-assn*

  sepref-register *prio*
  lemma [*sepref-import-param*]: (*prio*,*prio*) ∈ *Id* → *Id* **by** *simp*

  lemma [*sepref-import-param*]:
    ((≤), (≤)::*'b* ⇒ -) ∈ *Id* → *Id* → *bool-rel*
    ((<), (<)::*'b* ⇒ -) ∈ *Id* → *Id* → *bool-rel*
    by *simp-all*

  sepref-register
    update-op
    val-of-op
    PR-CONST prio-of-op
    exch-op
    valid
    length::*'a list* ⇒ -
    append-op
    butlast-op

    PR-CONST sink-op
    PR-CONST swim-op
    PR-CONST repair-op

  lemma [*def-pat-rules*]:
    heapstruct.prio-of-op$*prio* ≡ *PR-CONST prio-of-op*
    heapstruct.sink-op$*prio* ≡ *PR-CONST sink-op*
    heapstruct.swim-op$*prio* ≡ *PR-CONST swim-op*
    heapstruct.repair-op$*prio* ≡ *PR-CONST repair-op*
    by *simp-all*

end

context
  fixes *prio* :: *'a*::{*heap*,*default*} ⇒ *'b*::*linorder*
begin

  interpretation *heapstruct-impl prio* **.**

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* :: $rel^d *_a nat\text{-}assn^k *_a id\text{-}assn^k \rightarrow_a rel$
unfolding *update-op-def*[*abs-def*]
by *sepref*
lemmas [*sepref-fr-rules*] = *update-impl.refine*

sepref-definition *val-of-impl* **is** *uncurry val-of-op* :: $rel^k *_a nat\text{-}assn^k \rightarrow_a id\text{-}assn$
unfolding *val-of-op-def*[*abs-def*]
by *sepref*
lemmas [*sepref-fr-rules*] = *val-of-impl.refine*

sepref-definition *exch-impl* **is** *uncurry2 exch-op* :: $rel^d *_a nat\text{-}assn^k *_a nat\text{-}assn^k \rightarrow_a rel$
unfolding *exch-op-def*[*abs-def*]
by *sepref*
lemmas [*sepref-fr-rules*] = *exch-impl.refine*

sepref-definition *valid-impl* **is** *uncurry* (*RETURN oo valid*) :: $rel^k *_a nat\text{-}assn^k \rightarrow_a bool\text{-}assn$
unfolding *valid-def*[*abs-def*]
by *sepref*
lemmas [*sepref-fr-rules*] = *valid-impl.refine*

sepref-definition *prio-of-impl* **is** *uncurry* (*PR-CONST prio-of-op*) :: $rel^k *_a nat\text{-}assn^k \rightarrow_a id\text{-}assn$
unfolding *prio-of-op-def*[*abs-def*] *PR-CONST-def*
by *sepref*
lemmas [*sepref-fr-rules*] = *prio-of-impl.refine*

sepref-definition *swim-impl* **is** *uncurry* (*PR-CONST swim-op*) :: $rel^d *_a nat\text{-}assn^k \rightarrow_a rel$
unfolding *swim-op-def*[*abs-def*] *parent-def PR-CONST-def*
by *sepref*

lemmas [*sepref-fr-rules*] = *swim-impl.refine*

sepref-definition *sink-impl* **is** *uncurry* (*PR-CONST sink-op*) :: $rel^d *_a nat\text{-}assn^k \rightarrow_a rel$
unfolding *sink-op-opt-def*[*abs-def*] *sink-op-opt-eq*[*symmetric*,*abs-def*] *PR-CONST-def*
by *sepref*
lemmas [*sepref-fr-rules*] = *sink-impl.refine*

lemmas [*fcomp-norm-unfold*] = *heap-rel-def*[*symmetric*]

sepref-definition *empty-impl* **is** *uncurry0 empty-op* :: $unit\text{-}assn^k \rightarrow_a rel$

unfolding *empty-op-def arl.fold-custom-empty*
by *sepref*

sepref-decl-impl (*no-register*) *heap-empty*: *empty-impl.refine*[*FCOMP empty-op-refine*]
.

sepref-definition *is-empty-impl* **is** *is-empty-op* :: $rel^k \rightarrow_a$ *bool-assn*
unfolding *is-empty-op-def*[*abs-def*]
by *sepref*

sepref-decl-impl *heap-is-empty*: *is-empty-impl.refine*[*FCOMP is-empty-op-refine*]
.

sepref-definition *insert-impl* **is** *uncurry insert-op* :: $id\text{-}assn^k *_a rel^d \rightarrow_a rel$
unfolding *insert-op-def*[*abs-def*] *append-op-def*
by *sepref*
sepref-decl-impl *heap-insert*: *insert-impl.refine*[*FCOMP insert-op-refine*] .

sepref-definition *pop-min-impl* **is** *pop-min-op* :: $rel^d \rightarrow_a$ *prod-assn id-assn rel*
unfolding *pop-min-op-def*[*abs-def*] *butlast-op-def*
by *sepref*

sepref-decl-impl (*no-mop*) *heap-pop-min*: *pop-min-impl.refine*[*FCOMP pop-min-op-refine*]
.

sepref-definition *peek-min-impl* **is** *peek-min-op* :: $rel^k \rightarrow_a$ *id-assn*
unfolding *peek-min-op-def*[*abs-def*]
by *sepref*
sepref-decl-impl (*no-mop*) *heap-peek-min*: *peek-min-impl.refine*[*FCOMP peek-min-op-refine*]
.

end

definition [*simp*]: *heap-custom-empty* ≡ *op-mset-empty*
interpretation *heap*: *mset-custom-empty*
heap-rel prio A empty-impl heap-custom-empty **for** *prio A*
apply *unfold-locales*
apply (*rule heap-empty-hnr*)
by *simp*

3.7.3 Regression Test

export-code *empty-impl is-empty-impl insert-impl pop-min-impl peek-min-impl*
checking *SML*

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-assn (id-assn::'a::{default,heap} ⇒ -))^k →_a arl-assn
id-assn
  unfolding sort-by-prio-def[abs-def]
  by sepref
end
definition sort-impl-nat ≡ sort-impl (id::nat⇒nat)

export-code sort-impl checking SML

ML ‹
  @{code sort-impl-nat} (map @{code nat-of-integer} [4,1,7,2,3,9,8,62]) ()
›

hide-const sort-impl sort-impl-nat
hide-fact sort-impl-def sort-impl-nat-def sort-impl.refine

end
```

3.8 Map Interface

```
theory IICF-Map
imports ../../Sepref
begin
```

3.8.1 Parametricity for Maps

```
definition [to-relAPP]: map-rel K V ≡ (K → ⟨V⟩option-rel)
  ∩ { (mi,m). dom mi ⊆ Domain K ∧ dom m ⊆ Range K }


lemma bi-total-map-rel-eq:
  ⟦IS-RIGHT-TOTAL K; IS-LEFT-TOTAL K⟧ ⟹ ⟨K,V⟩map-rel = K → ⟨V⟩option-rel
  unfolding map-rel-def IS-RIGHT-TOTAL-def IS-LEFT-TOTAL-def
  by (auto dest: fun-relD)

lemma map-rel-Id[simp]: ⟨Id,Id⟩map-rel = Id
  unfolding map-rel-def by auto

lemma map-rel-empty1-simp[simp]:
  (Map.empty,m)∈⟨K,V⟩map-rel ⟷ m=Map.empty
  apply (auto simp: map-rel-def)
```

by (*meson RangeE domIff option-rel-simp*(*1*) *subsetCE tagged-fun-relD-none*)

lemma *map-rel-empty2-simp*[*simp*]:
(*m*,*Map.empty*)∈⟨*K*, *V*⟩*map-rel* ⟷ *m*=*Map.empty*
apply (*auto simp*: *map-rel-def*)
by (*meson Domain.cases domIff fun-relD2 option-rel-simp*(*2*) *subset-eq*)

lemma *map-rel-obtain1*:
assumes *1*: (*m*,*n*)∈⟨*K*, *V*⟩*map-rel*
assumes *2*: *n l* = *Some w*
obtains *k v* **where** *m k* = *Some v* (*k*,*l*)∈*K* (*v*,*w*)∈*V*
using *1* **unfolding** *map-rel-def*
proof *clarsimp*
assume *R*: (*m*, *n*) ∈ *K* → ⟨*V*⟩*option-rel*
assume *dom n* ⊆ *Range K*
with *2* **obtain** *k* **where** (*k*,*l*)∈*K* **by** *auto*
moreover from *fun-relD*[*OF R this*] **have** (*m k*, *n l*) ∈ ⟨*V*⟩*option-rel* **.**
with *2* **obtain** *v* **where** *m k* = *Some v* (*v*,*w*)∈*V* **by** (*cases m k*; *auto*)
ultimately show *thesis* **by** − (*rule that*)
qed

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*
using *1* **unfolding** *map-rel-def*
proof *clarsimp*
assume *R*: (*m*, *n*) ∈ *K* → ⟨*V*⟩*option-rel*
assume *dom m* ⊆ *Domain K*
with *2* **obtain** *l* **where** (*k*,*l*)∈*K* **by** *auto*
moreover from *fun-relD*[*OF R this*] **have** (*m k*, *n l*) ∈ ⟨*V*⟩*option-rel* **.**
with *2* **obtain** *w* **where** *n l* = *Some w* (*v*,*w*)∈*V* **by** (*cases n l*; *auto*)
ultimately show *thesis* **by** − (*rule that*)
qed

lemma *param-dom*[*param*]: (*dom*,*dom*)∈⟨*K*, *V*⟩*map-rel* → ⟨*K*⟩*set-rel*
apply (*clarsimp simp*: *set-rel-def*; *safe*)
apply (*erule* (*1*) *map-rel-obtain2*; *auto*)
apply (*erule* (*1*) *map-rel-obtain1*; *auto*)
done

3.8.2 Interface Type

sepref-decl-intf (′*k*,′*v*) *i-map* **is** ′*k* ⇀ ′*v*

lemma [*synth-rules*]: ⟦*INTF-OF-REL K TYPE*(′*k*); *INTF-OF-REL V TYPE*(′*v*)⟧

⟹ *INTF-OF-REL* (⟨*K*, *V*⟩*map-rel*) *TYPE*((′*k*,′*v*) *i-map*) **by** *simp*

3.8.3 Operations

sepref-decl-op *map-empty*: *Map.empty* :: $\langle K,V\rangle$*map-rel* .

sepref-decl-op *map-is-empty*: (=) *Map.empty* :: $\langle K,V\rangle$*map-rel* → *bool-rel*
apply (*rule fref-ncI*)
apply *parametricity*
apply (*rule fun-relI*; *auto*)
done

sepref-decl-op *map-update*: $\lambda k\ v\ m.\ m(k\mapsto v)$ :: $K \to V \to \langle K,V\rangle$*map-rel* → $\langle K,V\rangle$*map-rel*
where *single-valued K single-valued* (K^{-1})
apply (*rule fref-ncI*)
apply *parametricity*
unfolding *map-rel-def*
apply (*intro fun-relI*)
apply (*elim IntE*; *rule IntI*)
apply (*intro fun-relI*)
apply *parametricity*
apply (*simp add*: *pres-eq-iff-svb*)
apply *auto*
done

sepref-decl-op *map-delete*: $\lambda k\ m.$ *fun-upd m k None* :: $K \to \langle K,V\rangle$*map-rel* → $\langle K,V\rangle$*map-rel*
where *single-valued K single-valued* (K^{-1})
apply (*rule fref-ncI*)
apply *parametricity*
unfolding *map-rel-def*
apply (*intro fun-relI*)
apply (*elim IntE*; *rule IntI*)
apply (*intro fun-relI*)
apply *parametricity*
apply (*simp add*: *pres-eq-iff-svb*)
apply *auto*
done

sepref-decl-op *map-lookup*: $\lambda k\ (m::'k\rightharpoonup'v).\ m\ k :: K \to \langle K,V\rangle$*map-rel* → $\langle V\rangle$*option-rel*
apply (*rule fref-ncI*)
apply *parametricity*
unfolding *map-rel-def*
apply (*intro fun-relI*)
apply (*elim IntE*)
apply *parametricity*
done

lemma *in-dom-alt*: $k\in$*dom m* ⟷ ¬*is-None* (*m k*) **by** (*auto split*: *option.split*)

sepref-decl-op *map-contains-key*: $\lambda k\ m.\ k\in$*dom m* :: $K \to \langle K,V\rangle$*map-rel* →

bool-rel
unfolding *in-dom-alt*
apply (*rule fref-ncI*)
apply *parametricity*
unfolding *map-rel-def*
apply (*elim IntE*)
apply *parametricity*
done

3.8.4 Patterns

lemma *pat-map-empty*[*pat-rules*]: λ_2-. *None* ≡ *op-map-empty* **by** *simp*

lemma *pat-map-is-empty*[*pat-rules*]:
(=) $*m*$(λ_2-. *None*) ≡ *op-map-is-empty*$*m*
(=) $($\lambda_2$-. *None*)$*m* ≡ *op-map-is-empty*$*m*
(=) $(*dom*$*m*)$*{} ≡ *op-map-is-empty*$*m*
(=) $*{}*$(*dom*$*m*) ≡ *op-map-is-empty*$*m*
unfolding *atomize-eq*
by (*auto dest*: *sym*)

lemma *pat-map-update*[*pat-rules*]:
fun-upd$*m*$*k*$(*Some*$*v*) ≡ *op-map-update*$'*k*$'*v*$'*m*
by *simp*
lemma *pat-map-lookup*[*pat-rules*]: *m*$*k* ≡ *op-map-lookup*$'*k*$'*m*
by *simp*

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*
by (*simp-all add*: *map-upd-eq-restrict*)

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*
by (*auto intro*!: *eq-reflection*)

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*
unfolding *op-custom-empty-def* **by** *simp-all*

end

end

3.9 Priority Maps

theory *IICF-Prio-Map*
imports *IICF-Map*
begin

This interface inherits from maps, and adds some operations

lemma *uncurry-fun-rel-conv*:
(*uncurry f*, *uncurry g*) ∈ $A \times_r B \to R \longleftrightarrow (f,g) \in A \to B \to R$
by (*auto simp*: *uncurry-def dest*!: *fun-relD intro*: *prod-relI*)

lemma *uncurry0-fun-rel-conv*:
(*uncurry0 f*, *uncurry0 g*) ∈ *unit-rel* → $R \longleftrightarrow (f,g) \in R$
by (*auto dest*!: *fun-relD*)

lemma *RETURN-rel-conv0*: (*RETURN f*, *RETURN g*)∈⟨*A*⟩*nres-rel* $\longleftrightarrow (f,g) \in A$
by (*auto simp*: *nres-rel-def*)

lemma *RETURN-rel-conv1*: (*RETURN o f*, *RETURN o g*)∈*A* → ⟨*B*⟩*nres-rel* $\longleftrightarrow (f,g) \in A \to B$
by (*auto simp*: *nres-rel-def dest*!: *fun-relD*)

lemma *RETURN-rel-conv2*: (*RETURN oo f*, *RETURN oo g*)∈*A* → *B* → ⟨*R*⟩*nres-rel* $\longleftrightarrow (f,g) \in A \to B \to R$
by (*auto simp*: *nres-rel-def dest*!: *fun-relD*)

lemma *RETURN-rel-conv3*: (*RETURN ooo f*, *RETURN ooo g*)∈*A*→*B*→*C* → ⟨*R*⟩*nres-rel* $\longleftrightarrow (f,g) \in A \to B \to C \to R$
by (*auto simp*: *nres-rel-def dest*!: *fun-relD*)

lemmas *fref2param-unfold* =
uncurry-fun-rel-conv uncurry0-fun-rel-conv
RETURN-rel-conv0 RETURN-rel-conv1 RETURN-rel-conv2 RETURN-rel-conv3

lemmas *param-op-map-update*[*param*] = *op-map-update.fref*[*THEN fref-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}) .

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}) .

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}) .

context
fixes *prio* :: $'v \Rightarrow 'p$::*linorder*
begin
sepref-decl-op *pm-decrease-key*: *op-map-update*
:: $[\lambda((k,v),m).\ k \in dom\ m \wedge prio\ v \leq prio\ (the\ (m\ k))]_f\ (K \times_r V) \times_r \langle K,V\rangle map\text{-}rel \to \langle K,(V::('v \times 'v)\ set)\rangle map\text{-}rel$
where *single-valued K single-valued* (K^{-1}) *IS-BELOW-ID V*
proof *goal-cases*
case *1*
have [*param*]: $((\leq),(\leq)) \in Id \to Id \to bool\text{-}rel$ **by** *simp*
from *1* **show** *?case*
apply (*parametricity add*: *param-and-cong1*)
apply (*auto simp*: *IS-BELOW-ID-def map-rel-def dest*!: *fun-relD*)
done
qed

sepref-decl-op *pm-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 *goal-cases*
case *1*
have [*param*]: $((\leq),(\leq)) \in Id \to Id \to bool\text{-}rel$ **by** *simp*
from *1* **show** *?case*
apply (*parametricity add*: *param-and-cong1*)
apply (*auto simp*: *IS-BELOW-ID-def map-rel-def dest*!: *fun-relD*)
done
qed

lemma *IS-BELOW-ID-D*: $(a,b) \in R \Longrightarrow IS\text{-}BELOW\text{-}ID\ R \Longrightarrow a = b$ **by** (*auto simp*: *IS-BELOW-ID-def*)

sepref-decl-op *pm-peek-min*: $\lambda m.\ SPEC\ (\lambda(k,v).$
$m\ k = Some\ v \wedge (\forall k'\ v'.\ m\ k' = Some\ v' \longrightarrow prio\ v \leq prio\ v'))$
:: $[Not\ o\ op\text{-}map\text{-}is\text{-}empty]_f\ \langle K,V\rangle map\text{-}rel \to K \times_r (V::('v \times 'v)\ set)$

```
    where IS-BELOW-ID V
    apply (rule frefI)
    apply (intro nres-relI)
    apply (clarsimp simp: pw-le-iff refine-pw-simps)
    apply (rule map-rel-obtain2, assumption, assumption)
    apply1 (intro exI conjI allI impI; assumption?)
  proof −
    fix x y k' v' b w
    assume (x, y) ∈ ⟨K, V⟩map-rel y k' = Some v'
    then obtain k v where (k,k')∈K (v,v')∈V x k = Some v
      by (rule map-rel-obtain1)

    assume IS-BELOW-ID V (b, w) ∈ V
    with ‹(v,v')∈V› have [simp]: b=w v=v' by (auto simp: IS-BELOW-ID-def)

    assume ∀ k' v'. x k' = Some v' ⟶ prio b ≤ prio v'
    with ‹x k = Some v› show prio w ≤ prio v'
      by auto
  qed

  sepref-decl-op pm-pop-min: λm. SPEC (λ((k,v),m').
      m k = Some v
    ∧ m' = op-map-delete k m
    ∧ (∀ k' v'. m k' = Some v' ⟶ prio v ≤ prio v')
  ) :: [Not o op-map-is-empty]ₓ ⟨K,V⟩map-rel → (K×ᵣ(V::('v×'v) set))×ᵣ⟨K,V⟩map-rel
    where single-valued K single-valued (K⁻¹) IS-BELOW-ID V
    apply (rule frefI)
    apply (intro nres-relI)
    apply (clarsimp simp: pw-le-iff refine-pw-simps simp del: op-map-delete-def)
    apply (rule map-rel-obtain2, assumption, assumption)
    apply (intro exI conjI allI impI; assumption?)
    applyS parametricity
  proof −
    fix x y k' v' b w
    assume (x, y) ∈ ⟨K, V⟩map-rel y k' = Some v'
    then obtain k v where (k,k')∈K (v,v')∈V x k = Some v
      by (rule map-rel-obtain1)

    assume IS-BELOW-ID V (b, w) ∈ V
    with ‹(v,v')∈V› have [simp]: b=w v=v' by (auto simp: IS-BELOW-ID-def)

    assume ∀ k' v'. x k' = Some v' ⟶ prio b ≤ prio v'
    with ‹x k = Some v› show prio w ≤ prio v'
      by auto
  qed
 end

end
```

3.10 Priority Maps implemented with List and Map

theory *IICF-Abs-Heapmap*
imports *IICF-Abs-Heap HOL−Library.Rewrite ../../Intf/IICF-Prio-Map*
begin

type-synonym $('k,'v)$ *ahm* = $'k$ *list* × $('k \rightharpoonup 'v)$

3.10.1 Basic Setup

First, we define a mapping to list-based heaps

definition *hmr-α* :: $('k,'v)$ *ahm* ⇒ $'v$ *heap* **where**
hmr-α ≡ λ(*pq*,*m*). *map* (*the o m*) *pq*

definition *hmr-invar* ≡ λ(*pq*,*m*). *distinct pq* ∧ *dom m* = *set pq*

definition *hmr-rel* ≡ *br hmr-α hmr-invar*

lemmas *hmr-rel-defs* = *hmr-rel-def br-def hmr-α-def hmr-invar-def*

lemma *hmr-empty-invar*[*simp*]: *hmr-invar* ([],*Map.empty*)
by (*auto simp*: *hmr-invar-def*)

locale *hmstruct* = *h*: *heapstruct prio* **for** *prio* :: $'v$ ⇒ $'b$::*linorder*
begin

Next, we define a mapping to priority maps.

definition *heapmap-α* :: $('k,'v)$ *ahm* ⇒ $('k \rightharpoonup 'v)$ **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* **by** (*simp add*: *RELATES-def*)

lemma *h-heap-invarI*[*simp*]: *heapmap-invar hm* ⟹ *h.heap-invar* (*hmr-α hm*)

by (*simp add*: *heapmap-invar-def*)

lemma *hmr-invarI*[*simp*]: *heapmap-invar hm* ⟹ *hmr-invar hm*
unfolding *heapmap-invar-def* **by** *blast*

lemma *set-hmr-α*[*simp*]: *hmr-invar hm* ⟹ *set* (*hmr-α hm*) = *ran* (*heapmap-α hm*)
apply (*clarsimp simp*: *hmr-α-def hmr-invar-def heapmap-α-def*
eq-commute[*of dom - set* -] *ran-def*)
apply *force*
done

lemma *in-h-hmr-α-conv*[*simp*]: *hmr-invar hm* ⟹ *x* ∈# *h.α* (*hmr-α hm*) ⟷ *x* ∈ *ran* (*heapmap-α hm*)
by (*force simp*: *hmr-α-def hmr-invar-def heapmap-α-def in-multiset-in-set ran-is-image*)

3.10.2 Basic Operations

In this section, we define the basic operations on heapmaps, and their relations to heaps and maps.

Length

Length of the list that represents the heap

definition *hm-length* :: (′*k*,′*v*) *ahm* ⇒ *nat* **where**
hm-length ≡ λ(*pq*,-). *length pq*

lemma *hm-length-refine*: (*hm-length*, *length*) ∈ *hmr-rel* → *nat-rel*
apply (*intro fun-relI*)
unfolding *hm-length-def*
by (*auto simp*: *hmr-rel-defs*)

lemma *hm-length-hmr-α*[*simp*]: *length* (*hmr-α hm*) = *hm-length hm*
by (*auto simp*: *hm-length-def hmr-α-def split*: *prod.splits*)

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*
apply (*intro fun-relI*)
unfolding *hm-valid-def h.valid-def*
by (*parametricity add*: *hm-length-refine*)

lemma *hm-valid-hmr-α*[*simp*]: *h.valid* (*hmr-α hm*) = *hm-valid hm*
by (*intro ext*) (*auto simp*: *h.valid-def hm-valid-def*)

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*)
unfolding *hm-valid-def hm-length-def hm-key-of-op-def hm-key-of-def*
by (*auto split*: *prod.splits simp*: *pw-eq-iff refine-pw-simps*)

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*))
by (*auto*
simp: *hmr-α-def h.val-of-def heapmap-α-def hm-key-of-def hm-valid-def hm-length-def*
split: *prod.splits*)

lemma *hm-α-key-ex*[*simp*]:
⟦*hmr-invar hm*; *hm-valid hm i*⟧ ⟹ (*heapmap-α hm* (*hm-key-of hm i*) ≠ *None*)
unfolding *heapmap-invar-def hmr-invar-def hm-valid-def heapmap-α-def*
hm-key-of-def hm-length-def
by (*auto split*: *prod.splits*)

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*)
≫ *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*
unfolding *hm-exch-op-def h.exch-op-def h.val-of-op-def h.update-op-def*
apply *simp*

apply *refine-vcg*
apply (*auto simp*: *hm-valid-def map-swap hm-length-def hmr-rel-defs*)
done

lemma *hm-exch-op-refine*: (*hm-exch-op*,*h.exch-op*) ∈ *hmr-rel* → *nat-rel* → *nat-rel* → ⟨*hmr-rel*⟩*nres-rel*
apply (*intro fun-relI nres-relI*)
unfolding *hm-exch-op-def h.exch-op-def h.val-of-op-def h.update-op-def*
apply *simp*
apply *refine-vcg*
apply (*auto simp*: *hm-valid-def map-swap hm-length-def hmr-rel-defs*)
done

lemmas *hm-exch-op-refine'*[*refine*] = *hm-exch-op-refine*[*param-fo*, *THEN nres-relD*]

definition *hm-exch* :: ('*k*,'*v*) *ahm* ⇒ *nat* ⇒ *nat* ⇒ ('*k*,'*v*) *ahm*
where *hm-exch* ≡ λ(*pq*,*m*) *i j*. (*swap pq* (*i*−*1*) (*j*−*1*),*m*)

lemma *hm-exch-op-α-correct*: *hm-exch-op hm i j* ≤$_n$ *SPEC* (λ*hm'*.
hm-valid hm i ∧ *hm-valid hm j* ∧ *hm'*=*hm-exch hm i j*
)
unfolding *hm-exch-op-def*
apply *refine-vcg*
apply (*vc-solve simp*: *hm-valid-def hm-length-def heapmap-α-def solve*: *asm-rl*)
apply (*auto simp add*: *hm-key-of-def hm-exch-def swap-def*) []
done

lemma *hm-exch-α*[*simp*]: *heapmap-α* (*hm-exch hm i j*) = (*heapmap-α hm*)
by (*auto simp*: *heapmap-α-def hm-exch-def split*: *prod.splits*)
lemma *hm-exch-valid*[*simp*]: *hm-valid* (*hm-exch hm i j*) = *hm-valid hm*
by (*intro ext*) (*auto simp*: *hm-valid-def hm-length-def hm-exch-def split*: *prod.splits*)
lemma *hm-exch-length*[*simp*]: *hm-length* (*hm-exch hm i j*) = *hm-length hm*
by (*auto simp*: *hm-length-def hm-exch-def split*: *prod.splits*)

lemma *hm-exch-same*[*simp*]: *hm-exch hm i i* = *hm*
by (*auto simp*: *hm-exch-def split*: *prod.splits*)

lemma *hm-key-of-exch-conv*[*simp*]:
⟦*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
)
unfolding *hm-exch-def hm-valid-def hm-length-def hm-key-of-def*
by (*auto split*: *prod.splits*)

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*
by *simp-all*

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*)
by (*force simp*: *hm-valid-def heapmap-α-def hmr-invar-def hm-index-def hm-length-def Suc-le-eq*)

lemma *hm-index-key-of*[*simp*]: ⟦*hmr-invar hm*; *heapmap-α hm k* ≠ *None*⟧ ⟹ *hm-key-of hm* (*hm-index hm k*) = *k*
by (*force*
simp: *hm-valid-def heapmap-α-def hmr-invar-def hm-index-def hm-length-def hm-key-of-def Suc-le-eq*)

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*)
using *assms* **unfolding** *hm-index-op-def*
apply *refine-vcg*
apply (*auto simp*: *heapmap-α-def hmr-invar-def hm-index-def index-nth-id*)
done
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
  unfolding hm-update-op-def h.update-op-def
  apply refine-vcg
 by (auto simp: hmr-rel-defs map-distinct-upd-conv hm-valid-def hm-length-def)

lemma hm-update-op-refine: (hm-update-op, h.update-op) ∈ hmr-rel → nat-rel
→ Id → ⟨hmr-rel⟩nres-rel
  apply (intro fun-relI nres-relI)
  unfolding hm-update-op-def h.update-op-def mop-list-get-alt mop-list-set-alt
  apply refine-vcg
 apply (auto simp: hmr-rel-defs map-distinct-upd-conv hm-valid-def hm-length-def)
  done

lemmas [refine] = hm-update-op-refine[param-fo, THEN nres-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 (λhm'. heapmap-α hm'
= (heapmap-α hm)(k↦v))
  using assms
  unfolding hm-update-op-def
  apply refine-vcg
  apply (force simp: heapmap-rel-defs hmr-rel-defs hm-index-def)
  done
```

Butlast

Remove last element

```
definition hm-butlast-op :: ('k,'v) ahm ⇒ ('k,'v) ahm nres where
  hm-butlast-op ≡ λ(pq,m). do {
    ASSERT (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
  supply [simp del] = map-upd-eq-restrict
  apply (intro fun-relI nres-relI)
  unfolding hm-butlast-op-def h.butlast-op-def
  apply simp
  apply refine-vcg
  apply (clarsimp-all simp: hmr-rel-defs map-butlast distinct-butlast)
```

apply (*auto simp*: *neq-Nil-rev-conv*) []
done

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*))
unfolding *hm-butlast-op-def*
apply *refine-vcg*
apply (*auto simp*: *heapmap-α-def hm-key-of-def hm-length-def*)
done

Append

Append new element at end of heap

definition *hm-append-op* :: ('*k*,'*v*) *ahm* ⇒ '*k* ⇒ '*v* ⇒ ('*k*,'*v*) *ahm nres*
where *hm-append-op* ≡ λ(*pq*,*m*) *k v*. *do* {
ASSERT (*k* ∉ *dom m*);
ASSERT (*hmr-invar* (*pq*,*m*));
pq ← *mop-list-append pq k*;
let m = *m* (*k* ↦ *v*);
RETURN (*pq*,*m*)
}

lemma *hm-append-op-invar*: *hm-append-op hm k v* ≤$_n$ *SPEC hmr-invar*
unfolding *hm-append-op-def h.append-op-def*
apply *refine-vcg*
unfolding *heapmap-α-def hmr-rel-defs*
apply *auto*
done

lemma *hm-append-op-refine*: ⟦ *heapmap-α hm k* = *None*; (*hm*,*h*)∈*hmr-rel* ⟧
⟹ (*hm-append-op hm k v*, *h.append-op h v*) ∈ ⟨*hmr-rel*⟩*nres-rel*
apply (*intro fun-relI nres-relI*)
unfolding *hm-append-op-def h.append-op-def*
apply *refine-vcg*
unfolding *heapmap-α-def hmr-rel-defs*
apply *auto*
done

lemmas *hm-append-op-refine'*[*refine*] = *hm-append-op-refine*[*param-fo*, *THEN nres-relD*]

lemma *hm-append-op-α-correct*:
hm-append-op hm k v ≤$_n$ *SPEC* (λ*hm'*. *heapmap-α hm'* = (*heapmap-α hm*) (*k* ↦ *v*))
unfolding *hm-append-op-def*
apply *refine-vcg*

by (*auto simp*: *heapmap-α-def*)

3.10.3 Auxiliary Operations

Auxiliary operations on heapmaps, which are derived from the basic operations, but do not correspond to operations of the priority map interface

We start with some setup

lemma *heapmap-hmr-relI*: $(hm,h)\in$*heapmap-rel* $\Longrightarrow$ $(hm,$*hmr-α* $hm) \in$ *hmr-rel*

by (*auto simp*: *heapmap-rel-defs hmr-rel-defs*)

lemma *heapmap-hmr-relI'*: *heapmap-invar hm* $\Longrightarrow$ $(hm,$*hmr-α* $hm) \in$ *hmr-rel*

by (*auto simp*: *heapmap-rel-defs hmr-rel-defs*)

The basic principle how we prove correctness of our operations: Invariant preservation is shown by relating the operations to operations on heaps. Then, only correctness on the abstraction remains to be shown, assuming the operation does not fail.

lemma *heapmap-nres-relI'*:
assumes $hm \le \Downarrow$*hmr-rel h'*
assumes $h' \le$ *SPEC* (*h.heap-invar*)
assumes $hm \le_n$ *SPEC* (λ*hm'*. *RETURN* (*heapmap-α hm'*) $\le h$)
shows $hm \le \Downarrow$*heapmap-rel h*
using *assms*
unfolding *heapmap-rel-defs hmr-rel-def*
by (*auto simp*: *pw-le-iff pw-leof-iff refine-pw-simps*)

lemma *heapmap-nres-relI''*:
assumes $hm \le \Downarrow$*hmr-rel h'*
assumes $h' \le$ *SPEC* Φ
assumes $\bigwedge$*h'*. Φ *h'* $\Longrightarrow$ *h.heap-invar h'*
assumes $hm \le_n$ *SPEC* (λ*hm'*. *RETURN* (*heapmap-α hm'*) $\le h$)
shows $hm \le \Downarrow$*heapmap-rel h*
apply (*rule heapmap-nres-relI'*)
apply *fact*
apply (*rule order-trans*, *fact*)
apply (*clarsimp*; *fact*)
apply *fact*
done

Val-of

Indexing into the heap

definition *hm-val-of-op* :: ('*k*,'*v*) *ahm* $\Rightarrow$ *nat* $\Rightarrow$ '*v nres* **where**
hm-val-of-op $\equiv$ λ*hm i*. *do* {
k $\leftarrow$ *hm-key-of-op hm i*;

v ← *hm-the-lookup-op hm k*;
RETURN v
}

lemma *hm-val-of-op-refine*: (*hm-val-of-op*,*h.val-of-op*) ∈ (*hmr-rel* → *nat-rel* → ⟨*Id*⟩*nres-rel*)
apply (*intro fun-relI nres-relI*)
unfolding *hm-val-of-op-def h.val-of-op-def*
hm-key-of-op-def hm-key-of-def hm-valid-def hm-length-def
hm-the-lookup-op-def
apply *clarsimp*
apply (*rule refine-IdD*)
apply *refine-vcg*
apply (*auto simp*: *hmr-rel-defs heapmap-α-def*)
done

lemmas [*refine*] = *hm-val-of-op-refine*[*param-fo*, *THEN nres-relD*]

Prio-of

Priority of key

definition *hm-prio-of-op h i* ≡ *do* {*v* ← *hm-val-of-op h i*; *RETURN* (*prio v*)}

lemma *hm-prio-of-op-refine*: (*hm-prio-of-op*, *h.prio-of-op*) ∈ *hmr-rel* → *nat-rel* → ⟨*Id*⟩*nres-rel*
apply (*intro fun-relI nres-relI*)
unfolding *hm-prio-of-op-def h.prio-of-op-def*
apply *refine-rcg*
by *auto*

lemmas *hm-prio-of-op-refine′*[*refine*] = *hm-prio-of-op-refine*[*param-fo*, *THEN nres-relD*]

Swim

definition *hm-swim-op* :: (′*k*,′*v*) *ahm* ⇒ *nat* ⇒ (′*k*,′*v*) *ahm nres* **where**
hm-swim-op h i ≡ *do* {
RECT (λ*swim* (*h*,*i*). *do* {
ASSERT (*hm-valid h i* ∧ *h.swim-invar* (*hmr-α h*) *i*);
if hm-valid h (*h.parent i*) *then do* {
ppi ← *hm-prio-of-op h* (*h.parent i*);
pi ← *hm-prio-of-op h i*;
if (¬*ppi* ≤ *pi*) *then do* {
h ← *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) ∈ hmr-rel → nat-rel →
⟨hmr-rel⟩nres-rel
  apply (intro fun-relI nres-relI)
  unfolding hm-swim-op-def h.swim-op-def
  apply refine-rcg
  apply refine-dref-type
  apply (clarsimp-all simp: hm-valid-refine[param-fo, THEN IdD])
  apply (simp add: hmr-rel-def in-br-conv)
  done

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
  unfolding hm-swim-op-def
  apply (subst (asm) RECT-unfold, refine-mono)
  by (auto simp: refine-pw-simps)

lemma hm-swim-op-α-correct: hm-swim-op hm i ≤n SPEC (λhm'. heapmap-α
hm' = heapmap-α hm)
  apply (rule leof-add-nofailI)
  apply (drule hm-swim-op-nofail-imp-valid)
  unfolding hm-swim-op-def
  apply (rule RECT-rule-leof[where
      pre=λ(hm',i). hm-valid hm' i ∧ heapmap-α hm' = heapmap-α hm
      and V = inv-image less-than snd
      ])
  apply simp
  apply simp

  unfolding hm-prio-of-op-def hm-val-of-op-def
    hm-exch-op-def hm-key-of-op-def hm-the-lookup-op-def
  apply (refine-vcg)
  apply (vc-solve simp add: hm-valid-def hm-length-def)
  apply rprems
  apply (vc-solve simp: heapmap-α-def h.parent-def)
  done
```

Sink

```
definition hm-sink-op
where
  hm-sink-op h k ≡ RECT (λD (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*
apply (*intro fun-relI nres-relI*)
unfolding *hm-sink-op-def h.sink-op-opt-eq*[*symmetric*] *h.sink-op-opt-def*
apply *refine-rcg*
apply *refine-dref-type*

unfolding *hmr-rel-def heapmap-rel-def*
apply (*clarsimp-all simp*: *in-br-conv*)
done

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*
unfolding *hm-sink-op-def*
apply (*subst* (*asm*) *RECT-unfold*, *refine-mono*)
by (*auto simp*: *refine-pw-simps hm-valid-def*)

lemma *hm-sink-op-α-correct*: *hm-sink-op hm i* ≤$_n$ *SPEC* (λ*hm'*. *heapmap-α hm'* = *heapmap-α hm*)
apply (*rule leof-add-nofailI*)
apply (*drule hm-sink-op-nofail-imp-valid*)
unfolding *hm-sink-op-def*
apply (*rule RECT-rule-leof*[**where**
  pre=λ(*hm'*,*i*). *hm-valid hm' i* ∧ *heapmap-α hm'* = *heapmap-α hm* ∧ *hm-length hm'* = *hm-length hm*
  and *V* = *measure* (λ(*hm'*,*i*). *hm-length hm'* − *i*)
  ])

apply *simp*
apply *simp*

unfolding *hm-prio-of-op-def hm-val-of-op-def hm-exch-op-def*
hm-key-of-op-def hm-the-lookup-op-def
apply (*refine-vcg*)
apply (*vc-solve simp add*: *hm-valid-def hm-length-def*)
apply *rprems*
apply (*vc-solve simp*: *heapmap-α-def h.parent-def split*: *prod.splits*)
apply (*auto*)
done

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*
apply (*intro fun-relI nres-relI*)
unfolding *hm-repair-op-def h.repair-op-def*
by *refine-rcg*

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* ≤$_n$ *SPEC* (λ*hm'*. *heapmap-α hm'* = *heapmap-α hm*)
unfolding *hm-repair-op-def*
apply (*refine-vcg*
hm-swim-op-α-correct[*THEN leof-trans*]
hm-sink-op-α-correct[*THEN leof-trans*])
by *auto*

3.10.4 Operations

In this section, we define the operations that implement the priority-map interface

Empty

definition *hm-empty-op* :: (*'k*,*'v*) *ahm nres*
where *hm-empty-op* ≡ *RETURN* ([],*Map.empty*)

lemma *hm-empty-aref*: (*hm-empty-op*,*RETURN op-map-empty*) ∈ ⟨*heapmap-rel*⟩*nres-rel*

unfolding *hm-empty-op-def*
by (*auto simp*: *heapmap-rel-defs hmr-rel-defs intro*: *nres-relI*)

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*)
unfolding *hm-insert-op-def h.insert-op-def*
apply *refine-rcg*
by (*auto simp*: *hmr-rel-def br-def*)

lemma *hm-insert-op-aref*:
(*hm-insert-op*,*mop-map-update-new*) ∈ *Id* → *Id* → *heapmap-rel* → ⟨*heapmap-rel*⟩*nres-rel*
apply (*intro fun-relI nres-relI*)
unfolding *mop-map-update-new-alt*
apply (*rule ASSERT-refine-right*)
apply (*rule heapmap-nres-relI''*[*OF hm-insert-op-refine h.insert-op-correct*])
apply (*unfold heapmap-rel-def in-br-conv*; *clarsimp*)
apply (*erule heapmap-hmr-relI*)
apply (*unfold heapmap-rel-def in-br-conv*; *clarsimp*)
apply (*unfold heapmap-rel-def in-br-conv*; *clarsimp*)
unfolding *hm-insert-op-def*
apply (*refine-vcg*
hm-append-op-α-correct[*THEN leof-trans*]
hm-swim-op-α-correct[*THEN leof-trans*])
apply (*unfold heapmap-rel-def in-br-conv*; *clarsimp*)
done

Is-Empty

lemma *hmr-α-empty-iff*[*simp*]:
hmr-invar hm ⟹ *hmr-α hm* = [] ⟷ *heapmap-α hm* = *Map.empty*
by (*auto*
simp: *hmr-α-def heapmap-invar-def heapmap-α-def hmr-invar-def*
split: *prod.split*)

definition *hm-is-empty-op* :: (*'k*,*'v*) *ahm* ⇒ *bool nres* **where**
hm-is-empty-op ≡ 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*
apply (*intro fun-relI nres-relI*)
unfolding *hm-is-empty-op-def h.is-empty-op-def*
apply *refine-rcg*
apply (*auto simp*: *hmr-rel-defs*) []
apply (*parametricity add*: *hm-length-refine*)
done

lemma *hm-is-empty-op-aref*: (*hm-is-empty-op*, *RETURN o op-map-is-empty*) ∈ *heapmap-rel* → ⟨*bool-rel*⟩*nres-rel*
apply (*intro fun-relI nres-relI*)
unfolding *hm-is-empty-op-def*
apply *refine-vcg*
apply (*auto simp*: *hmr-rel-defs heapmap-rel-defs hm-length-def*)
done

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*
apply (*intro fun-relI nres-relI*)
unfolding *hm-lookup-op-def heapmap-rel-def in-br-conv*
apply *refine-vcg*
apply *simp-all*
done

Contains-Key

definition *hm-contains-key-op* ≡ 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*
apply (*intro fun-relI nres-relI*)
unfolding *hm-contains-key-op-def heapmap-rel-defs*
apply *refine-vcg*
by (*auto*)

Decrease-Key

definition *hm-decrease-key-op* ≡ λ*k v hm*. *do* {
ASSERT (*heapmap-invar hm*);

ASSERT (*heapmap-α hm k* ≠ *None* ∧ *prio v* ≤ *prio* (*the* (*heapmap-α hm k*)));
i ← *hm-index-op hm k*;
hm ← *hm-update-op hm i v*;
hm-swim-op hm i
}

definition (**in** *heapstruct*) *decrease-key-op i v h* ≡ *do* {
ASSERT (*valid h i* ∧ *prio v* ≤ *prio-of h i*);
h ← *update-op h i v*;
swim-op h i
}

lemma (**in** *heapstruct*) *decrease-key-op-invar*:
⟦*heap-invar h*; *valid h i*; *prio v* ≤ *prio-of h i*⟧ ⟹ *decrease-key-op i v h* ≤ *SPEC heap-invar*
unfolding *decrease-key-op-def*
apply *refine-vcg*
by (*auto simp*: *swim-invar-decr*)

lemma *index-op-inline-refine*:
assumes *heapmap-invar hm*
assumes *heapmap-α hm k* ≠ *None*
assumes *f* (*hm-index hm k*) ≤ *m*
shows *do* {*i* ← *hm-index-op hm k*; *f i*} ≤ *m*
using *hm-index-op-correct*[*of hm k*] *assms*
by (*auto simp*: *pw-le-iff refine-pw-simps*)

lemma *hm-decrease-key-op-refine*:
⟦(*hm*,*h*)∈*hmr-rel*; (*hm*,*m*)∈*heapmap-rel*; *m k* = *Some v*⟧
⟹ *hm-decrease-key-op k v hm* ≤⇓*hmr-rel* (*h.decrease-key-op* (*hm-index hm k*) *v h*)
unfolding *hm-decrease-key-op-def h.decrease-key-op-def*

apply (*refine-rcg index-op-inline-refine*)
unfolding *hmr-rel-def heapmap-rel-def in-br-conv*
apply (*clarsimp-all*)
done

lemma *hm-index-op-inline-leof*:
assumes *f* (*hm-index hm k*) ≤$_n$ *m*
shows *do* {*i* ← *hm-index-op hm k*; *f i*} ≤$_n$ *m*
using *hm-index-op-correct*[*of hm k*] *assms* **unfolding** *hm-index-op-def*
by (*auto simp*: *pw-le-iff pw-leof-iff refine-pw-simps split*: *prod.splits*)

lemma *hm-decrease-key-op-α-correct*:
heapmap-invar hm ⟹ *hm-decrease-key-op k v hm* ≤$_n$ *SPEC* (λ*hm′*. *heapmap-α hm′* = (*heapmap-α hm*)(*k*↦*v*))

unfolding *hm-decrease-key-op-def*
apply (*refine-vcg*
hm-update-op-α-correct[*THEN leof-trans*]
hm-swim-op-α-correct[*THEN leof-trans*]
hm-index-op-inline-leof
)
apply *simp-all*
done

lemma *hm-decrease-key-op-aref*:
(*hm-decrease-key-op*, *PR-CONST* (*mop-pm-decrease-key prio*)) ∈ *Id* → *Id* → *heapmap-rel* → ⟨*heapmap-rel*⟩*nres-rel*
unfolding *PR-CONST-def*
apply (*intro fun-relI nres-relI*)
apply (*frule heapmap-hmr-relI*)
unfolding *mop-pm-decrease-key-alt*
apply (*rule ASSERT-refine-right*; *clarsimp*)
apply (*rule heapmap-nres-relI'*)
apply (*rule hm-decrease-key-op-refine*; *assumption*)
unfolding *heapmap-rel-def hmr-rel-def in-br-conv*
apply (*rule h.decrease-key-op-invar*; *simp*; *fail*)
apply (*refine-vcg hm-decrease-key-op-α-correct*[*THEN leof-trans*]; *simp*; *fail*)
done

Increase-Key

definition *hm-increase-key-op* ≡ λ*k v hm*. *do* {
ASSERT (*heapmap-invar hm*);
ASSERT (*heapmap-α hm k* ≠ *None* ∧ *prio v* ≥ *prio* (*the* (*heapmap-α hm k*)));
i ← *hm-index-op hm k*;
hm ← *hm-update-op hm i v*;
hm-sink-op hm i
}

definition (**in** *heapstruct*) *increase-key-op i v h* ≡ *do* {
ASSERT (*valid h i* ∧ *prio v* ≥ *prio-of h i*);
h ← *update-op h i v*;
sink-op h i
}

lemma (**in** *heapstruct*) *increase-key-op-invar*:
⟦*heap-invar h*; *valid h i*; *prio v* ≥ *prio-of h i*⟧ ⟹ *increase-key-op i v h* ≤ *SPEC heap-invar*
unfolding *increase-key-op-def*
apply *refine-vcg*
by (*auto simp*: *sink-invar-incr*)

lemma *hm-increase-key-op-refine*:

⟦(*hm*,*h*)∈*hmr-rel*; (*hm*,*m*)∈*heapmap-rel*; *m k* = *Some v*′⟧
⟹ *hm-increase-key-op k v hm* ≤⇓*hmr-rel* (*h.increase-key-op* (*hm-index hm k*) *v h*)
unfolding *hm-increase-key-op-def h.increase-key-op-def*

apply (*refine-rcg index-op-inline-refine*)
unfolding *hmr-rel-def heapmap-rel-def in-br-conv*
apply (*clarsimp-all*)
done

lemma *hm-increase-key-op-α-correct*:
heapmap-invar hm ⟹ *hm-increase-key-op k v hm* ≤$_n$ *SPEC* (λ*hm*′. *heapmap-α hm*′ = (*heapmap-α hm*)(*k*↦*v*))
unfolding *hm-increase-key-op-def*
apply (*refine-vcg*
hm-update-op-α-correct[*THEN leof-trans*]
hm-sink-op-α-correct[*THEN leof-trans*]
hm-index-op-inline-leof)
apply *simp-all*
done

lemma *hm-increase-key-op-aref*:
(*hm-increase-key-op*, *PR-CONST* (*mop-pm-increase-key prio*)) ∈ *Id* → *Id* → *heapmap-rel* → ⟨*heapmap-rel*⟩*nres-rel*
unfolding *PR-CONST-def*
apply (*intro fun-relI nres-relI*)
apply (*frule heapmap-hmr-relI*)
unfolding *mop-pm-increase-key-alt*
apply (*rule ASSERT-refine-right*; *clarsimp*)
apply (*rule heapmap-nres-relI*′)
apply (*rule hm-increase-key-op-refine*; *assumption*)
unfolding *heapmap-rel-def hmr-rel-def in-br-conv*
apply (*rule h.increase-key-op-invar*; *simp*; *fail*)
apply (*refine-vcg hm-increase-key-op-α-correct*[*THEN leof-trans*]; *simp*)
done

Change-Key

definition *hm-change-key-op* ≡ λ*k v hm*. *do* {
ASSERT (*heapmap-invar hm*);
ASSERT (*heapmap-α hm k* ≠ *None*);
i ← *hm-index-op hm k*;
hm ← *hm-update-op hm i v*;
hm-repair-op hm i
}

definition (**in** *heapstruct*) *change-key-op i v h* ≡ *do* {
ASSERT (*valid h i*);
h ← *update-op h i v*;

repair-op h i
}

lemma (**in** *heapstruct*) *change-key-op-invar*:
⟦*heap-invar h*; *valid h i*⟧ ⟹ *change-key-op i v h* ≤ *SPEC heap-invar*
unfolding *change-key-op-def*
apply (*refine-vcg*)
apply *hypsubst*
apply *refine-vcg*
by (*auto simp*: *sink-invar-incr*)

lemma *hm-change-key-op-refine*:
⟦(*hm*,*h*)∈*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*)
unfolding *hm-change-key-op-def h.change-key-op-def*

apply (*refine-rcg index-op-inline-refine*)
unfolding *hmr-rel-def heapmap-rel-def in-br-conv*
apply (*clarsimp-all*)
done

lemma *hm-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*))
unfolding *hm-change-key-op-def*
apply (*refine-vcg*
hm-update-op-α-correct[*THEN leof-trans*]
hm-repair-op-α-correct[*THEN leof-trans*]
hm-index-op-inline-leof)
unfolding *heapmap-rel-def in-br-conv*
apply *simp*
apply *simp*
done

lemma *hm-change-key-op-aref*:
(*hm-change-key-op*, *mop-map-update-ex*) ∈ *Id* → *Id* → *heapmap-rel* → ⟨*heapmap-rel*⟩*nres-rel*
apply (*intro fun-relI nres-relI*)
apply (*frule heapmap-hmr-relI*)
unfolding *mop-map-update-ex-alt*
apply (*rule ASSERT-refine-right*; *clarsimp*)
apply (*rule heapmap-nres-relI*')
apply (*rule hm-change-key-op-refine*; *assumption*)
unfolding *heapmap-rel-def hmr-rel-def in-br-conv*
apply (*rule h.change-key-op-invar*; *simp*; *fail*)
apply ((*refine-vcg hm-change-key-op-α-correct*[*THEN leof-trans*]; *simp*))
done

Set

Realized as generic algorithm!

```
  lemma (in −) op-pm-set-gen-impl: RETURN ooo op-map-update = (λk v m.
do {
    c ← RETURN (op-map-contains-key k m);
    if c then
      mop-map-update-ex k v m
    else
      mop-map-update-new k v m
  })
    apply (intro ext)
  unfolding op-map-contains-key-def mop-map-update-ex-def mop-map-update-new-def
    by simp

  definition hm-set-op k v hm ≡ do {
    c ← hm-contains-key-op k hm;
    if c then
      hm-change-key-op k v hm
    else
      hm-insert-op k v hm
  }

  lemma hm-set-op-aref:
      (hm-set-op, RETURN ooo op-map-update) ∈ Id → Id → heapmap-rel →
⟨heapmap-rel⟩nres-rel
    unfolding op-pm-set-gen-impl
    apply (intro fun-relI nres-relI)
    unfolding hm-set-op-def o-def
    apply (refine-rcg
      hm-contains-key-op-aref[param-fo, unfolded o-def, THEN nres-relD]
      hm-change-key-op-aref[param-fo, THEN nres-relD]
      hm-insert-op-aref[param-fo, THEN nres-relD]
      )
    by auto
```

Pop-Min

```
  definition hm-pop-min-op :: ('k,'v) ahm ⇒ (('k×'v) × ('k,'v) ahm) nres where
    hm-pop-min-op hm ≡ do {
      ASSERT (heapmap-invar hm);
      ASSERT (hm-valid hm 1);
      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
  apply (intro fun-relI nres-relI)
  unfolding hm-pop-min-op-def h.pop-min-op-def

  unfolding ignore-snd-refine-conv hm-the-lookup-op-def hm-key-of-op-unfold
  apply (simp cong: if-cong add: Let-def)
  apply (simp add: unused-bind-conv h.val-of-op-def refine-pw-simps)


  apply refine-rcg
  unfolding hmr-rel-def in-br-conv
  apply (unfold heapmap-invar-def;simp)
  apply (auto simp: in-br-conv)
  done
```

We demonstrate two different approaches for proving correctness here. The first approach uses the relation to plain heaps only to establish the invariant.

The second approach also uses the relation to heaps to establish correctness of the result.

The first approach seems to be more robust against badly set up simpsets, which may be the case in early stages of development.

Assuming a working simpset, the second approach may be less work, and the proof may look more elegant.

First approach Transfer heapmin-property to heapmap-domain

```
lemma heapmap-min-prop:
  assumes INV: heapmap-invar hm
  assumes V': heapmap-α hm k = Some v'
  assumes NE: hm-valid hm (Suc 0)
  shows prio (the (heapmap-α hm (hm-key-of hm (Suc 0)))) ≤ prio v'
proof −
  — Transform into the domain of heaps
  obtain pq m where [simp]: hm=(pq,m) by (cases hm)

  from NE have [simp]: pq≠[] by (auto simp: hm-valid-def hm-length-def)

  have CNV-LHS: prio (the (heapmap-α hm (hm-key-of hm (Suc 0))))
    = h.prio-of (hmr-α hm) (Suc 0)
    by (auto simp: heapmap-α-def hm-key-of-def hmr-α-def h.val-of-def)

  from INV have INV': h.heap-invar (hmr-α hm)
```

unfolding *heapmap-invar-def* **by** *auto*

from *V' INV* **obtain** *i* **where** *IDX*: *h.valid* (*hmr-α hm*) *i*
and *CNV-RHS*: *prio v' = h.prio-of* (*hmr-α hm*) *i*
apply (*clarsimp simp*: *heapmap-α-def heapmap-invar-def hmr-invar-def hmr-α-def*
h.valid-def h.val-of-def)
by (*metis* (*no-types*, *opaque-lifting*) *Suc-leI comp-apply diff-Suc-Suc*
diff-zero domI index-less-size-conv neq0-conv nth-index nth-map
old.nat.distinct(*2*) *option.sel*)

from *h.heap-min-prop*[*OF INV' IDX*] **show** *?thesis*
unfolding *CNV-LHS CNV-RHS* **.**
qed

With the above lemma, the correctness proof is straightforward

lemma *hm-pop-min-α-correct*: *hm-pop-min-op hm* $\leq_n$ *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'*))
unfolding *hm-pop-min-op-def hm-key-of-op-unfold hm-the-lookup-op-def*
apply (*refine-vcg*
hm-exch-op-α-correct[*THEN leof-trans*]
hm-butlast-op-α-correct[*THEN leof-trans*]
hm-sink-op-α-correct[*THEN leof-trans*]
)
apply (*auto simp*: *heapmap-min-prop*)
done

lemma *heapmap-nres-rel-prodI*:
assumes *hmx* ≤ ⇓(*UNIV* ×$_r$ *hmr-rel*) *h'x*
assumes *h'x* ≤ *SPEC* (λ(-,*h'*). *h.heap-invar h'*)
assumes *hmx* $\leq_n$ *SPEC* (λ(*r*,*hm'*). *RETURN* (*r*,*heapmap-α hm'*) ≤ ⇓(*R*×$_r$*Id*) *hx*)
shows *hmx* ≤ ⇓(*R*×$_r$*heapmap-rel*) *hx*
using *assms*
unfolding *heapmap-rel-def hmr-rel-def br-def heapmap-invar-def*
apply (*auto simp*: *pw-le-iff pw-leof-iff refine-pw-simps*; *blast*)
done

lemma *hm-pop-min-op-aref*: (*hm-pop-min-op*, *PR-CONST* (*mop-pm-pop-min prio*)) ∈ *heapmap-rel* → ⟨(*Id*×$_r$*Id*)×$_r$*heapmap-rel*⟩*nres-rel*
unfolding *PR-CONST-def*
apply (*intro fun-relI nres-relI*)
apply (*frule heapmap-hmr-relI*)
unfolding *mop-pm-pop-min-alt*
apply (*intro ASSERT-refine-right*)
apply (*rule heapmap-nres-rel-prodI*)

apply (*rule hm-pop-min-op-refine*[*param-fo*, *THEN nres-relD*]; *assumption*)
unfolding *heapmap-rel-def hmr-rel-def in-br-conv*
apply (*refine-vcg*; *simp*)
apply (*refine-vcg hm-pop-min-α-correct*[*THEN leof-trans*]; *simp split*: *prod.splits*)
done

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*)
unfolding *hm-kv-of-op-def h.val-of-op-def kvi-rel-def*
hm-key-of-op-unfold hm-the-lookup-op-def
apply *simp*
apply *refine-vcg*
using *assms*
by (*auto*
simp: *hm-valid-def hm-length-def hmr-rel-defs heapmap-α-def hm-key-of-def*
split: *prod.splits*)

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*)
unfolding *hm-pop-min-op′-def h.pop-min-op-def*

unfolding *ignore-snd-refine-conv*

apply *refine-rcg*
unfolding *hmr-rel-def heapmap-rel-def*
apply (*unfold heapmap-invar-def*; *simp add*: *in-br-conv*)
apply (*simp-all add*: *in-br-conv*)
done

lemma *heapmap-nres-rel-prodI'*:
assumes $hmx \le \Downarrow(S \times_r hmr\text{-}rel)\ h'x$
assumes $h'x \le SPEC\ \Phi$
assumes $\bigwedge h'\ r.\ \Phi\ (r,h') \Longrightarrow h.heap\text{-}invar\ h'$
assumes $hmx \le_n SPEC\ (\lambda(r,hm').\ (\exists r'.\ (r,r')\in S \wedge \Phi\ (r',hmr\text{-}\alpha\ hm')) \wedge hmr\text{-}invar\ hm' \longrightarrow RETURN\ (r,heapmap\text{-}\alpha\ hm') \le \Downarrow(R\times_r Id)\ hx)$
shows $hmx \le \Downarrow(R\times_r heapmap\text{-}rel)\ hx$
using *assms*
unfolding *heapmap-rel-def hmr-rel-def heapmap-invar-def*
apply (*auto*
simp: *pw-le-iff pw-leof-iff refine-pw-simps in-br-conv*
)
by *meson*

lemma *ex-in-kvi-rel-conv*:
$(\exists r'.\ (r,r')\in kvi\text{-}rel\ hm\ i \wedge \Phi\ r') \longleftrightarrow (fst\ r = hm\text{-}key\text{-}of\ hm\ i \wedge \Phi\ (snd\ r))$
unfolding *kvi-rel-def*
apply (*cases r*)
apply *auto*
done

lemma *hm-pop-min-aref'*: $(hm\text{-}pop\text{-}min\text{-}op', mop\text{-}pm\text{-}pop\text{-}min\ prio) \in heapmap\text{-}rel \to \langle(Id\times_r Id) \times_r heapmap\text{-}rel\rangle nres\text{-}rel$
apply (*intro fun-relI nres-relI*)
apply (*frule heapmap-hmr-relI*)
unfolding *mop-pm-pop-min-alt*
apply (*intro ASSERT-refine-right*)
apply (*rule heapmap-nres-rel-prodI'*)
apply (*erule hm-pop-min-op-refine'*)

apply (*unfold heapmap-rel-def hmr-rel-def in-br-conv*) []
apply (*rule h.pop-min-op-correct*)
apply *simp*
apply *simp*

apply *simp*

apply (*clarsimp simp*: *ex-in-kvi-rel-conv split*: *prod.splits*)
unfolding *hm-pop-min-op'-def hm-kv-of-op-def hm-key-of-op-unfold*

```
    hm-the-lookup-op-def
  apply (refine-vcg
    hm-exch-op-α-correct[THEN leof-trans]
    hm-butlast-op-α-correct[THEN leof-trans]
    hm-sink-op-α-correct[THEN leof-trans]
    )
  unfolding heapmap-rel-def hmr-rel-def in-br-conv
  apply (auto intro: ranI)
 done
```

Remove

```
definition hm-remove-op k hm ≡ do {
  ASSERT (heapmap-invar hm);
  ASSERT (k ∈ dom (heapmap-α hm));
  i ← hm-index-op hm k;
  let l = hm-length hm;
  hm ← hm-exch-op hm i l;
  hm ← hm-butlast-op hm;
  if i ≠ l then
    hm-repair-op hm i
  else
    RETURN hm
}

definition (in heapstruct) remove-op i h ≡ do {
  ASSERT (heap-invar h);
  ASSERT (valid h i);
  let l = length h;
  h ← exch-op h i l;
  h ← butlast-op h;
  if i ≠ l then
    repair-op h i
  else
    RETURN h
}

lemma (in −) swap-empty-iff[iff]: swap l i j = [] ⟷ l=[]
  by (auto simp: swap-def)

lemma (in heapstruct)
  butlast-exch-last: butlast (exch h i (length h)) = update (butlast h) i (last h)
  unfolding exch-def update-def
  apply (cases h rule: rev-cases)
  apply (auto simp: swap-def butlast-list-update list-update-beyond)
  done

lemma (in heapstruct) remove-op-invar:
  ⟦ heap-invar h; valid h i ⟧ ⟹ remove-op i h ≤ SPEC heap-invar
```

unfolding *remove-op-def*
apply *refine-vcg*
apply (*auto simp*: *valid-def*) []
apply (*auto simp*: *valid-def exch-def*) []
apply (*simp add*: *butlast-exch-last*)
apply *refine-vcg*
apply *auto* []
apply *auto* []
apply (*auto simp*: *valid-def*) []
apply *auto* []
apply *auto* []
done

lemma *hm-remove-op-refine*[*refine*]:
⟦ (*hm*,*m*)∈*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*)
unfolding *hm-remove-op-def h.remove-op-def heapmap-rel-def*

apply (*refine-rcg index-op-inline-refine*)
unfolding *hmr-rel-def*
apply (*auto simp*: *in-br-conv*)
done

lemma *hm-remove-op-α-correct*:
hm-remove-op k hm ≤$_n$ *SPEC* (λ*hm'*. *heapmap-α hm'* = (*heapmap-α hm*)(*k*:=*None*))

unfolding *hm-remove-op-def*
apply (*refine-vcg*
hm-exch-op-α-correct[*THEN leof-trans*]
hm-butlast-op-α-correct[*THEN leof-trans*]
hm-repair-op-α-correct[*THEN leof-trans*]
hm-index-op-inline-leof
)
apply (*auto*; *fail*)

apply *clarsimp*
apply (*rewrite at hm-index - k* = *hm-length -* **in** *asm eq-commute*)
apply (*auto*; *fail*)
done

lemma *hm-remove-op-aref*:
(*hm-remove-op*,*mop-map-delete-ex*) ∈ *Id* → *heapmap-rel* → ⟨*heapmap-rel*⟩*nres-rel*
apply (*intro fun-relI nres-relI*)
unfolding *mop-map-delete-ex-alt*
apply (*rule ASSERT-refine-right*)
apply (*frule heapmap-hmr-relI*)
apply (*rule heapmap-nres-relI'*)
apply (*rule hm-remove-op-refine*; *assumption?*)
apply (*unfold heapmap-rel-def in-br-conv*; *auto*)

unfolding *heapmap-rel-def hmr-rel-def in-br-conv*
apply (*refine-vcg h.remove-op-invar*; *clarsimp*; *fail*)
apply (*refine-vcg hm-remove-op-α-correct*[*THEN leof-trans*]; *simp*; *fail*)
done

Peek-Min

definition *hm-peek-min-op* :: ('*k*,'*v*) *ahm* ⇒ ('*k*×'*v*) *nres* **where**
hm-peek-min-op hm ≡ *hm-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*
unfolding *PR-CONST-def*
apply (*intro fun-relI nres-relI*)
proof −
fix *hm* **and** *m* :: '*k* ⇀ '*v*
assume *A*: (*hm*,*m*)∈*heapmap-rel*

from *A* **have** [*simp*]: *h.heap-invar* (*hmr-α hm*) *hmr-invar hm m*=*heapmap-α hm*
unfolding *heapmap-rel-def in-br-conv heapmap-invar-def*
by *simp-all*

have *hm-peek-min-op hm* ≤ ⇓ (*kvi-rel hm 1*) (*h.peek-min-op* (*hmr-α hm*))
unfolding *hm-peek-min-op-def h.peek-min-op-def*
apply (*refine-rcg hm-kv-op-refine*)
using *A*
apply (*simp add*: *heapmap-hmr-relI*)
done
also have ⟦*hmr-α hm* ≠ []⟧ ⟹ (*h.peek-min-op* (*hmr-α hm*))
≤ *SPEC* (λ*v*. *v*∈*ran* (*heapmap-α hm*) ∧ (∀ *v*'∈*ran* (*heapmap-α hm*). *prio v* ≤ *prio v*'))
apply *refine-vcg*
by *simp-all*
finally show *hm-peek-min-op hm* ≤ ⇓ (*Id* ×$_r$ *Id*) (*mop-pm-peek-min prio m*)

unfolding *mop-pm-peek-min-alt*
apply (*simp add*: *pw-le-iff refine-pw-simps hm-peek-min-op-def hm-kv-of-op-def*

hm-key-of-op-unfold hm-the-lookup-op-def)
apply (*fastforce simp*: *kvi-rel-def ran-def*)
done

qed

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*
apply *rule*
unfolding *is-array-def*
apply *prec-extract-eqs*
by *simp*

definition *array-assn* **where** *array-assn A* ≡ *hr-comp is-array* (⟨*the-pure A*⟩*list-rel*)
lemmas [*safe-constraint-rules*] = *CN-FALSEI*[*of is-pure array-assn A* **for** *A*]

definition [*simp*,*code-unfold*]: *heap-array-empty* ≡ *Array.of-list* []
definition [*simp*,*code-unfold*]: *heap-array-set p i v* ≡ *Array.upd i v p*

context
notes [*fcomp-norm-unfold*] = *array-assn-def*[*symmetric*]
notes [*intro*!] = *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*
by *sep-auto*
sepref-decl-impl (*no-register*) *array-empty*: *array-empty-hnr-aux* .

lemma *array-replicate-hnr-aux*:
(*uncurry Array.new*, *uncurry* (*RETURN oo op-list-replicate*))
∈ *nat-assn*^{k} ∗$_a$ *id-assn*^{k} →$_a$ *is-array*
by (*sep-auto*)
sepref-decl-impl (*no-register*) *array-replicate*: *array-replicate-hnr-aux* .

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*

by (*auto simp*: *op-array-replicate-def intro*!: *ext*)
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*
unfolding *list-assn-pure-conv*
by (*sep-auto*)
sepref-decl-impl (*no-register*) *array-of-list*: *array-of-list-hnr-aux* .

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*
by (*auto intro*!: *ext*)
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*
by *sep-auto*
sepref-decl-impl *array-copy*: *array-copy-hnr-aux* .

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*
by *sep-auto*
sepref-decl-impl *array-get*: *array-get-hnr-aux* .

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*

by *sep-auto*
sepref-decl-impl *array-set*: *array-set-hnr-aux* .

lemma *array-length-hnr-aux*: (*Array.len*,*RETURN o op-list-length*) ∈ *is-array*^{k} →$_a$ *nat-assn*
by *sep-auto*
sepref-decl-impl *array-length*: *array-length-hnr-aux* .

end

definition [*simp*]: *op-array-empty* ≡ *op-list-empty*
interpretation *array*: *list-custom-empty array-assn A heap-array-empty op-array-empty*
apply *unfold-locales*
apply (*rule array-empty-hnr*[*simplified pre-list-empty-def*])
by (*auto*)

end
theory *IICF-MS-Array-List*
imports
../Intf/IICF-List
Separation-Logic-Imperative-HOL.Array-Blit
Separation-Logic-Imperative-HOL.Default-Insts
begin

type-synonym *'a ms-array-list = 'a Heap.array × nat*

definition *is-ms-array-list ms l ≡ λ(a,n). ∃ A l'. a ↦a l' ∗ ↑(n ≤ length l' ∧ l = take n l' ∧ ms=length l')*

lemma *is-ms-array-list-prec*[*safe-constraint-rules*]: *precise* (*is-ms-array-list ms*)
unfolding *is-ms-array-list-def*[*abs-def*]
apply(*rule preciseI*)
apply(*simp split*: *prod.splits*)
using *preciseD snga-prec* **by** *fastforce*

definition *marl-empty-sz maxsize ≡ do* {
a ← Array.new maxsize default;
return (*a*,*0*)
}

definition *marl-append ≡ λ*(*a*,*n*) *x. do* {
a ← Array.upd n x a;
return (*a*,*n*+*1*)
}

definition *marl-length* :: *'a*::*heap ms-array-list ⇒ nat Heap* **where**
marl-length ≡ λ(*a*,*n*). *return* (*n*)

definition *marl-is-empty* :: *'a*::*heap ms-array-list ⇒ bool Heap* **where**
marl-is-empty ≡ λ(*a*,*n*). *return* (*n*=*0*)

definition *marl-last* :: *'a*::*heap ms-array-list ⇒ 'a Heap* **where**
marl-last ≡ λ(*a*,*n*). *do* {
Array.nth a (*n* − *1*)
}

definition *marl-butlast* :: *'a*::*heap ms-array-list ⇒ 'a ms-array-list Heap* **where**
marl-butlast ≡ λ(*a*,*n*). *do* {
return (*a*,*n* − *1*)
}

definition *marl-get* :: *'a*::*heap ms-array-list ⇒ nat ⇒ 'a Heap* **where**
marl-get ≡ λ(*a*,*n*) *i. Array.nth a i*

definition *marl-set* :: *'a::heap ms-array-list ⇒ nat ⇒ 'a ⇒ 'a ms-array-list Heap*
where
marl-set ≡ λ(*a*,*n*) *i x*. *do* { *a* ← *Array.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* []>
by (*sep-auto simp*: *marl-empty-sz-def is-ms-array-list-def*)

lemma *marl-append-rule*[*sep-heap-rules*]: *length l* < *N* ⟹
< *is-ms-array-list N l a* >
marl-append a x
<λ*a*. *is-ms-array-list N* (*l*@[*x*]) *a* >$_t$
by (*sep-auto*
simp: *marl-append-def is-ms-array-list-def take-update-last*
split: *prod.splits*)

lemma *marl-length-rule*[*sep-heap-rules*]:
<*is-ms-array-list N l a*>
marl-length a
<λ*r*. *is-ms-array-list N l a* ∗ ↑(*r*=*length l*)>
by (*sep-auto simp*: *marl-length-def is-ms-array-list-def*)

lemma *marl-is-empty-rule*[*sep-heap-rules*]:
<*is-ms-array-list N l a*>
marl-is-empty a
<λ*r*. *is-ms-array-list N l a* ∗ ↑(*r*⟷(*l*=[]))>
by (*sep-auto simp*: *marl-is-empty-def is-ms-array-list-def*)

lemma *marl-last-rule*[*sep-heap-rules*]:
l≠[] ⟹
<*is-ms-array-list N l a*>
marl-last a
<λ*r*. *is-ms-array-list N l a* ∗ ↑(*r*=*last l*)>
by (*sep-auto simp*: *marl-last-def is-ms-array-list-def last-take-nth-conv*)

lemma *marl-butlast-rule*[*sep-heap-rules*]:
l≠[] ⟹
<*is-ms-array-list N l a*>
marl-butlast a
<*is-ms-array-list N* (*butlast l*)>$_t$
by (*sep-auto*
split: *prod.splits*
simp: *marl-butlast-def is-ms-array-list-def butlast-take*)

lemma *marl-get-rule*[*sep-heap-rules*]:
i<*length l* ⟹
<*is-ms-array-list N l a*>
marl-get a i

<λr. is-ms-array-list N l a ∗ ↑(r=l!i)>
by (*sep-auto simp*: *marl-get-def is-ms-array-list-def split*: *prod.split*)

lemma *marl-set-rule*[*sep-heap-rules*]:
i<*length l* ⟹
<*is-ms-array-list N l a*>
marl-set a i x
<*is-ms-array-list N* (*l*[*i*:=*x*])>
by (*sep-auto simp*: *marl-set-def is-ms-array-list-def split*: *prod.split*)

definition *marl-assn N A* ≡ *hr-comp* (*is-ms-array-list N*) (⟨*the-pure A*⟩*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*) ≡ *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* ≡ *UNPROTECT* (*op-marl-empty-sz N*) **by** *simp*

lemma *marl-fold-custom-empty-sz*:
op-list-empty = *op-marl-empty-sz N*
mop-list-empty = *RETURN* (*op-marl-empty-sz N*)
[] = *op-marl-empty-sz N*
by *auto*

lemma *marl-empty-hnr-aux*: (*uncurry0* (*marl-empty-sz N*), *uncurry0* (*RETURN op-list-empty*)) ∈ *unit-assn*^{k} →$_a$ *is-ms-array-list N*
by *sep-auto*
lemmas *marl-empty-hnr* = *marl-empty-hnr-aux*[*FCOMP op-list-empty.fref*[*of the-pure A* **for** *A*]]
lemmas *marl-empty-hnr-mop* = *marl-empty-hnr*[*FCOMP 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*
using *marl-empty-hnr*
by *simp*

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*

by *sep-auto*
lemmas *marl-append-hnr*[*sepref-fr-rules*] = *marl-append-hnr-aux*[*FCOMP op-list-append.fref*]
lemmas *marl-append-hnr-mop*[*sepref-fr-rules*] = *marl-append-hnr*[*FCOMP mk-mop-rl2-np*[*OF mop-list-append-alt*]]

lemma *marl-length-hnr-aux*: (*marl-length*,*RETURN o op-list-length*) ∈ (*is-ms-array-list N*)^k →$_a$ *nat-assn*
by *sep-auto*
lemmas *marl-length-hnr*[*sepref-fr-rules*] = *marl-length-hnr-aux*[*FCOMP op-list-length.fref*[*of the-pure A* **for** *A*]]
lemmas *marl-length-hnr-mop*[*sepref-fr-rules*] = *marl-length-hnr*[*FCOMP mk-mop-rl1-np*[*OF mop-list-length-alt*]]

lemma *marl-is-empty-hnr-aux*: (*marl-is-empty*,*RETURN o op-list-is-empty*) ∈ (*is-ms-array-list N*)^k →$_a$ *bool-assn*
by *sep-auto*
lemmas *marl-is-empty-hnr*[*sepref-fr-rules*] = *marl-is-empty-hnr-aux*[*FCOMP op-list-is-empty.fref*[*of the-pure A* **for** *A*]]
lemmas *marl-is-empty-hnr-mop*[*sepref-fr-rules*] = *marl-is-empty-hnr*[*FCOMP mk-mop-rl1-np*[*OF mop-list-is-empty-alt*]]

lemma *marl-last-hnr-aux*: (*marl-last*,*RETURN o op-list-last*) ∈ [λ*x*. *x*≠[]]$_a$ (*is-ms-array-list N*)^k → *id-assn*
by *sep-auto*
lemmas *marl-last-hnr*[*sepref-fr-rules*] = *marl-last-hnr-aux*[*FCOMP op-list-last.fref*]
lemmas *marl-last-hnr-mop*[*sepref-fr-rules*] = *marl-last-hnr*[*FCOMP mk-mop-rl1*[*OF mop-list-last-alt*]]

lemma *marl-butlast-hnr-aux*: (*marl-butlast*,*RETURN o op-list-butlast*) ∈ [λ*x*. *x*≠[]]$_a$ (*is-ms-array-list N*)^d → (*is-ms-array-list N*)
by *sep-auto*
lemmas *marl-butlast-hnr*[*sepref-fr-rules*] = *marl-butlast-hnr-aux*[*FCOMP op-list-butlast.fref*[*of the-pure A* **for** *A*]]
lemmas *marl-butlast-hnr-mop*[*sepref-fr-rules*] = *marl-butlast-hnr*[*FCOMP mk-mop-rl1*[*OF mop-list-butlast-alt*]]

lemma *marl-get-hnr-aux*: (*uncurry marl-get*,*uncurry* (*RETURN oo op-list-get*)) ∈ [λ(*l*,*i*). *i*<*length l*]$_a$ ((*is-ms-array-list N*)^k *$_a$ *nat-assn*^{k}) → *id-assn*
by *sep-auto*
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*)
by *sep-auto*
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*])
})
  using *N-sz*
  by *sepref*

end

schematic-goal *hn-refine* (*emp*) (*?c*::*?'c Heap*) *?*Γ′ *?R* (*do* {
  let x = *op-list-empty*;
  RETURN (*x*@[*1*::*nat*])
})
  apply (*subst marl-fold-custom-empty-sz*[**where** *N=10*])
  apply *sepref*
  done

end
theory *IICF-Indexed-Array-List*
imports
  HOL−Library.Rewrite
  ../Intf/IICF-List
  List−Index.List-Index
  IICF-Array
  IICF-MS-Array-List
begin

We implement distinct lists of natural numbers in the range $\{0..<N\}$ by a length counter and two arrays of size N. The first array stores the list, and the second array stores the positions of the elements in the list, or N if the element is not in the list.

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 −
   from card-mono[OF - l-set] have card (set l) ≤ length qp by auto
   with distinct-card[OF l-distinct] show ?thesis by simp
  qed

  lemma idx-len[simp]: i<length l ⟹ l!i < length qp
   using l-set
   by (metis atLeastLessThan-iff nth-mem psubsetD psubsetI)

  lemma l-set-simp[simp]: k∈set l ⟹ k < length qp
   by (auto dest: subsetD[OF l-set])

  lemma qpk-idx: k<length qp ⟹ qp ! k < length l ⟷ k ∈ set l
  proof (rule iffI)
   assume A: k<length qp
   {
    assume qp!k < length l
    hence qp!k < length qp using l-len by simp
    with spec[OF qp-def, of k] A show k∈set l
     by (auto split: if-split-asm)
   }
   {
    assume k∈set l
    thus qp!k<length l
     using qp-def by (auto split: if-split-asm) []
   }
  qed

  lemma lqpk[simp]: k ∈ set l ⟹ l ! (qp ! k) = k
   using spec[OF qp-def, of k] by auto

  lemma ⟦i<length l; j<length l; l!i=l!j⟧ ⟹ i=j
   by (simp add: nth-eq-iff-index-eq)

 lemmas index-swap[simp] = index-swap-if-distinct[folded swap-def, OF l-distinct]


  lemma swap-invar:
   assumes i<length l j<length l
   shows ial-invar (length qp) (swap l i j) (qp[l ! j := i, l ! i := j])
   using assms
```

apply *unfold-locales*
apply *auto* []
apply *auto* []
apply *auto* []
apply (*auto simp*: *simp*: *nth-list-update nth-eq-iff-index-eq index-nth-id*) []
using *qp-def* **apply** *auto* [*2*]
done

end

definition *ial-rel1 maxsize* ≡ *br fst* (*uncurry* (*ial-invar maxsize*))

definition *ial-assn2* :: *nat* ⇒ *nat list* ∗ *nat list* ⇒ - **where**
ial-assn2 maxsize ≡ *prod-assn* (*marl-assn maxsize nat-assn*) (*array-assn nat-assn*)

definition *ial-assn maxsize A* ≡ *hr-comp* (*hr-comp* (*ial-assn2 maxsize*) (*ial-rel1 maxsize*)) (⟨*the-pure A*⟩*list-rel*)
lemmas [*safe-constraint-rules*] = *CN-FALSEI*[*of is-pure ial-assn maxsize A* **for** *maxsize A*]

3.11.1 Empty

definition *op-ial-empty-sz* :: *nat* ⇒ *'a list*
where [*simp*]: *op-ial-empty-sz ms* ≡ *op-list-empty*

lemma [*def-pat-rules*]: *op-ial-empty-sz*$*maxsize* ≡ *UNPROTECT* (*op-ial-empty-sz maxsize*)
by *simp*

context fixes *maxsize* :: *nat* **begin**
sepref-register *PR-CONST* (*op-ial-empty-sz maxsize*)
end

context
fixes *maxsize* :: *nat*
notes [*fcomp-norm-unfold*] = *ial-assn-def*[*symmetric*]
notes [*simp*] = *hn-ctxt-def pure-def*
begin

definition *aial-empty* ≡ *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*

unfolding *aial-empty-def*
apply (*refine-vcg nres-relI*)
apply (*clarsimp simp*: *ial-rel1-def br-def*)
apply *unfold-locales*
apply *auto*
done

context
notes [*id-rules*] = *itypeI*[*Pure.of maxsize TYPE*(*nat*)]
notes [*sepref-import-param*] = *IdI*[*of maxsize*]
begin
sepref-definition *ial-empty* **is** *uncurry0 aial-empty* :: *unit-assn*^{k} →$_a$ *ial-assn2 maxsize*
unfolding *aial-empty-def ial-assn2-def*
using [[*id-debug*]]
by *sepref*
end

sepref-decl-impl (*no-register*) *ial-empty*: *ial-empty.refine*[*FCOMP aial-empty-impl*]
.
lemma *ial-empty-sz-hnr*[*sepref-fr-rules*]:
(*uncurry0 local.ial-empty*, *uncurry0* (*RETURN* (*PR-CONST* (*op-ial-empty-sz maxsize*)))) ∈ *unit-assn*^{k} →$_a$ *ial-assn maxsize A*
using *ial-empty-hnr*[*of A*] **by** *simp*

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*
by (*auto simp*: *ial-rel1-def in-br-conv*)

lemma *aial-swap-impl*:
(*aial-swap*,*mop-list-swap*) ∈ *ial-rel1 maxsize* → *nat-rel* → *nat-rel* → ⟨*ial-rel1 maxsize*⟩*nres-rel*
proof (*intro fun-relI nres-relI*; *clarsimp simp*: *in-ial-rel1-conv*; *refine-vcg*; *clarsimp*)
fix *l qp i j*
assume [*simp*]: *i*<*length l j*<*length l* **and** *ial-invar maxsize l qp*

then interpret *ial-invar maxsize l qp* **by** *simp*

show *aial-swap* (*l*, *qp*) *i j* ≤ *SPEC* (λ*c*. (*c*, *swap l i j*) ∈ *ial-rel1 maxsize*)
unfolding *aial-swap-def*
apply *refine-vcg*
apply (*vc-solve simp add*: *in-ial-rel1-conv swap-def*[*symmetric*] *swap-invar*)
done
qed

sepref-definition *ial-swap* **is**
uncurry2 aial-swap :: (*ial-assn2 maxsize*)^d $*_a$ *nat-assn*^{k} $*_a$ *nat-assn*^{k} $\rightarrow_a$ *ial-assn2 maxsize*
unfolding *aial-swap-def ial-assn2-def*
by *sepref*

sepref-decl-impl (*ismop*) *test*: *ial-swap.refine*[*FCOMP aial-swap-impl*]
uses *mop-list-swap.fref* **.**

3.11.3 Length

definition *aial-length* :: *aial* ⇒ *nat nres*
where *aial-length* ≡ λ(*l*,-). *RETURN* (*op-list-length l*)

lemma *aial-length-impl*: (*aial-length*, *mop-list-length*) ∈ *ial-rel1 maxsize* → ⟨*nat-rel*⟩*nres-rel*
apply (*intro fun-relI nres-relI*)
unfolding *ial-rel1-def in-br-conv aial-length-def*
by *auto*

sepref-definition *ial-length* **is** *aial-length* :: (*ial-assn2 maxsize*)^k $\rightarrow_a$ *nat-assn*
unfolding *aial-length-def ial-assn2-def*
by *sepref*

sepref-decl-impl (*ismop*) *ial-length*: *ial-length.refine*[*FCOMP aial-length-impl*]
.

3.11.4 Index

definition *aial-index* :: *aial* ⇒ *nat* ⇒ *nat nres* **where**
aial-index ≡ λ(*l*,*qp*) *k*. *do* {
ASSERT (*k*∈*set l*);
i ← *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*
apply (*intro fun-relI nres-relI frefI*)
unfolding *ial-rel1-def*

proof (*clarsimp simp*: *in-br-conv*)
fix *l qp k*
assume *ial-invar maxsize l qp*
then interpret *ial-invar maxsize l qp* **.**

assume *k*∈*set l*
then show *aial-index* (*l*,*qp*) *k* ≤ *RETURN* (*index l k*)
unfolding *aial-index-def*
apply (*refine-vcg*)
by (*auto simp*: *qp-def*)
qed

sepref-definition *ial-index* **is** *uncurry aial-index* :: (*ial-assn2 maxsize*)^k $*_a$ *nat-assn*^{k} $\rightarrow_a$ *nat-assn*
unfolding *aial-index-def ial-assn2-def*
by *sepref*

sepref-decl-impl (*ismop*) *ial-index*: *ial-index.refine*[*FCOMP aial-index-impl*] **.**

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*
apply (*intro fun-relI nres-relI*)
unfolding *ial-rel1-def*
proof (*clarsimp simp*: *in-br-conv simp del*: *mop-list-butlast-alt*)
fix *l qp*
assume *ial-invar maxsize l qp*
then interpret *ial-invar maxsize l qp* **.**

{
assume *A*: *l*≠[]
have *ial-invar* (*length qp*) (*butlast l*) (*qp*[*l* ! (*length l* − *Suc 0*) := *length qp*])
apply *standard*
apply *clarsimp-all*
apply (*auto simp*: *distinct-butlast*) []
using *l-set* **apply** (*auto dest*: *in-set-butlastD*) []
using *qp-def A l-distinct*
apply (*auto simp*: *nth-list-update neq-Nil-rev-conv index-append simp del*:

l-distinct)
        done
    } **note** *aux1*=*this*

  show *aial-butlast* (*l*, *qp*) ≤ ⇓ (*br fst* (*uncurry* (*ial-invar maxsize*))) (*mop-list-butlast* *l*)
      unfolding *aial-butlast-def mop-list-butlast-alt*
      apply *refine-vcg*
      apply (*clarsimp-all simp*: *in-br-conv aux1*)
      done
  qed

  sepref-definition *ial-butlast* **is** *aial-butlast* :: (*ial-assn2 maxsize*)^d →$_a$ *ial-assn2* *maxsize*
    unfolding *aial-butlast-def ial-assn2-def* **by** *sepref*

  sepref-decl-impl (*ismop*) *ial-butlast*: *ial-butlast.refine*[*FCOMP aial-butlast-refine*]
.

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*
    apply (*intro frefI nres-relI*)
    unfolding *ial-rel1-def*
  proof (*clarsimp simp*: *in-br-conv*)
    fix *l qp k*
    assume *KLM*: *k*<*maxsize* **and** *KNL*: *k*∉*set l*
    assume *ial-invar maxsize l qp*
    then interpret *ial-invar maxsize l qp* **.**

    from *KLM* **have** *KLL*: *k*<*length qp* **by** *simp*

    note *distinct-card*[*OF l-distinct*, *symmetric*]
    also from *KNL l-set* **have** *set l* ⊆ {*0*..<*k*} ∪ {*Suc k*..<*length qp*}
      by (*auto simp*: *nat-less-le*)
    from *card-mono*[*OF* - *this*] **have** *card* (*set l*) ≤ *card* ...
      by *simp*

also note *card-Un-le*
also have *card* $\{0..<k\}$ + *card* $\{Suc\ k..<length\ qp\} = k + (length\ qp - Suc\ k)$
by *simp*
also have $\ldots < length\ qp$ **using** *KLL* **by** *simp*
finally have *LLEN*: $length\ l < length\ qp$ **.**

have *aux1*[*simp*]: *ial-invar* (*length qp*) (*l* @ [*k*]) (*qp*[*k* := *length l*])
apply *standard*
apply (*clarsimp-all simp*: *KNL KLL*)
using *KLL* **apply** (*auto simp*: *Suc-le-eq LLEN*) []
apply (*auto simp*: *index-append KNL nth-list-update'*)
apply (*simp add*: *qp-def*)
apply (*simp add*: *qp-def*)
done

show *aial-append* (*l*, *qp*) *k* ≤ ⇓ (*br fst* (*uncurry* (*ial-invar maxsize*))) (*RETURN* (*l*@[*k*]))
unfolding *aial-append-def mop-list-append-def*
apply *refine-vcg*
apply (*clarsimp-all simp*: *in-br-conv KLL KNL LLEN*)
done
qed

private lemma *aial-append-impl-aux*: ((*l*, *qp*), *l'*) ∈ *ial-rel1 maxsize* ⟹ *l'*=*l* ∧ *maxsize* = *length qp*
unfolding *ial-rel1-def*
by (*clarsimp simp*: *in-br-conv ial-invar.maxsize-eq*[*symmetric*])

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*
unfolding *aial-append-def ial-assn2-def*
by *sepref*
end

lemma (λ*b*. *b*<*maxsize*, *X*) ∈ *A* → *bool-rel*
apply *auto*
oops

context begin

private lemma *append-fref'*: ⟦*IS-BELOW-ID R*⟧
⟹ (*uncurry mop-list-append*, *uncurry mop-list-append*) ∈ ⟨*R*⟩*list-rel* ×$_r$ *R*

$\rightarrow_f$ $\langle\langle R\rangle$*list-rel*$\rangle$*nres-rel*
by (*rule mop-list-append.fref*)

sepref-decl-impl (*ismop*) *ial-append*: *ial-append.refine*[*FCOMP aial-append-refine*]
uses *append-fref'*
unfolding *IS-BELOW-ID-def*
apply (*parametricity*; *auto simp*: *single-valued-below-Id*)
done
end

3.11.7 Get

definition *aial-get* :: *aial* $\Rightarrow$ *nat* $\Rightarrow$ *nat nres* **where**
aial-get $\equiv$ λ(*l*,*qp*) *i*. *mop-list-get l i*

lemma *aial-get-refine*: (*aial-get*,*mop-list-get*) $\in$ *ial-rel1 maxsize* $\rightarrow$ *nat-rel* $\rightarrow$ $\langle$*nat-rel*$\rangle$*nres-rel*
apply (*intro fun-relI nres-relI*)
unfolding *aial-get-def ial-rel1-def mop-list-get-def in-br-conv*
apply *refine-vcg*
apply *clarsimp-all*
done

sepref-definition *ial-get* **is** *uncurry aial-get* :: (*ial-assn2 maxsize*)^k $*_a$ *nat-assn*^{k} $\rightarrow_a$ *nat-assn*
unfolding *aial-get-def ial-assn2-def* **by** *sepref*

sepref-decl-impl (*ismop*) *ial-get*: *ial-get.refine*[*FCOMP aial-get-refine*] **.**

3.11.8 Contains

definition *aial-contains* :: *nat* $\Rightarrow$ *aial* $\Rightarrow$ *bool nres* **where**
aial-contains $\equiv$ λk (*l*,*qp*). *do* {
if k<*maxsize then do* {
i $\leftarrow$ *mop-list-get qp k*;
RETURN (*i*<*maxsize*)
} *else RETURN False*
}

lemma *aial-contains-refine*: (*uncurry aial-contains*,*uncurry mop-list-contains*)
$\in$ (*nat-rel* $\times_r$ *ial-rel1 maxsize*) $\rightarrow_f$ $\langle$*bool-rel*$\rangle$*nres-rel*
apply (*intro frefI nres-relI*)
unfolding *ial-rel1-def*
proof (*clarsimp simp*: *in-br-conv*)
fix *l qp k*

assume *ial-invar maxsize l qp*
then interpret *ial-invar maxsize l qp* **.**

show *aial-contains k* (*l*, *qp*) $\leq$ *RETURN* (*k*$\in$*set l*)

unfolding *aial-contains-def*
apply *refine-vcg*
by (*auto simp*: *l-len qp-def split*: *if-split-asm*)
qed

context
notes [*id-rules*] = *itypeI*[*Pure.of maxsize TYPE*(*nat*)]
notes [*sepref-import-param*] = *IdI*[*of maxsize*]
begin
sepref-definition *ial-contains* **is** *uncurry aial-contains* :: $nat\text{-}assn^k *_a (ial\text{-}assn2\ maxsize)^k \to_a bool\text{-}assn$
unfolding *aial-contains-def ial-assn2-def* **by** *sepref*
end

sepref-decl-impl (*ismop*) *ial-contains*: *ial-contains.refine*[*FCOMP aial-contains-refine*]
.
end

interpretation *ial-sz*: *list-custom-empty ial-assn N A ial-empty N PR-CONST* (*op-ial-empty-sz N*)
apply *unfold-locales*
apply (*rule ial-empty-sz-hnr* [*unfolded op-ial-empty-sz-def PR-CONST-def*])
by *simp*

end

3.12 Implementation of Heaps by Arrays

theory *IICF-Impl-Heapmap*
imports *IICF-Abs-Heapmap ../IICF-Indexed-Array-List*
begin

Some setup to circumvent the really inefficient implementation of division in the code generator, which has to consider several cases for negative divisors and dividends.

definition [*code-unfold*]:
efficient-nat-div2 n
≡ *nat-of-integer* (*fst* (*Code-Numeral.divmod-abs* (*integer-of-nat n*) *2*))

lemma *efficient-nat-div2*[*simp*]: *efficient-nat-div2 n* = *n div 2*
by (*simp add*: *efficient-nat-div2-def nat-of-integer.rep-eq*)

type-synonym *'v hma* = *nat list* × (*'v list*)
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* **by** *simp*
lemmas [*sepref-import-param*] = *param-prio*
sepref-register *prio*
end

context
fixes *maxsize* :: *nat*
fixes *prio* :: *'v*::*heap* ⇒ *'p*::*linorder*
notes [*map-type-eqs*] = *map-type-eqI*[*Pure.of TYPE*((*nat*,*'v*) *ahm*) *TYPE*(*'v i-hma*)]
begin

interpretation *hmstruct* .
interpretation *hmstruct-impl* .

definition *hm-impl1-α* ≡ λ(*pq*,*ml*).
(*pq*,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*
unfolding *hm-impl1-α-def* **by** (*auto split*: *if-split-asm*)

term *hm-empty-op*
definition *hm-empty-op'* :: *'v hma nres*
where *hm-empty-op'* ≡ *do* {
let pq = *op-ial-empty-sz maxsize*;
let ml = *op-list-empty*;
RETURN (*pq*,*ml*)

}

lemma *hm-empty-op'-refine*: (*hm-empty-op'*, *hm-empty-op*) ∈ ⟨*hm-impl1-rel*⟩*nres-rel*

apply (*intro fun-relI nres-relI*)
unfolding *hm-empty-op'-def hm-empty-op-def hm-impl1-rel-defs*
by (*auto simp*: *in-br-conv*)

definition *hm-length'* :: *'v hma* ⇒ *nat* **where** *hm-length'* ≡ λ(*pq*,*ml*). *length pq*

lemma *hm-length'-refine*: (*RETURN o hm-length'*,*RETURN o hm-length*) ∈ *hm-impl1-rel* → ⟨*nat-rel*⟩*nres-rel*
apply (*intro fun-relI nres-relI*)
unfolding *hm-length'-def hm-length-def hm-impl1-rel-defs*
by (*auto*)

term *hm-key-of-op*
definition *hm-key-of-op'* ≡ λ(*pq*,*ml*) *i*. *ASSERT* (*i*>*0*) ≫ *mop-list-get pq* (*i* − *1*)
lemma *hm-key-of-op'-refine*: (*hm-key-of-op'*, *hm-key-of-op*) ∈ *hm-impl1-rel* → *nat-rel* → ⟨*nat-rel*⟩*nres-rel*
apply (*intro fun-relI nres-relI*)
unfolding *hm-key-of-op'-def hm-key-of-op-def hm-impl1-rel-defs*
by (*auto*)

term *hm-lookup*
definition *hm-lookup-op'* ≡ λ(*pq*,*ml*) *k*. *do* {
if (*k*<*maxsize*) *then do* { — TODO: This check can be eliminated, but this 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*
apply (*intro frefI nres-relI*)
unfolding *hm-lookup-op-def hm-lookup-op'-def o-def uncurry-def*
apply *refine-vcg*
apply (*auto simp*: *hm-impl1-rel-defs heapmap-α-def hmr-invar-def*)
done

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*
apply (*intro frefI nres-relI*)
unfolding *hm-contains-key-op-def hm-contains-key-op′-def o-def uncurry-def PR-CONST-def*
apply *refine-vcg*
apply (*auto simp*: *hm-impl1-rel-defs heapmap-α-def hmr-invar-def*)
done

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*
using *assms*
unfolding *hmr-rel-def hm-impl1-rel-def hm-weak-impl′-rel-def in-br-conv hm-impl1-weak-invar-def hm-impl1-invar-def*
by *auto*

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*
using *assms hm-impl1-relI*
apply (*auto simp*: *pw-le-iff pw-leof-iff refine-pw-simps in-br-conv nres-rel-def*)
apply *blast*
done

lemma *hm-exch-op′-refine*: (*hm-exch-op′*, *hm-exch-op*) ∈ *hm-impl1-rel* → *nat-rel* → *nat-rel* → ⟨*hm-impl1-rel*⟩*nres-rel*
apply (*intro fun-relI hm-impl1-nres-relI*[*OF hm-exch-op-invar*])

unfolding *hm-exch-op'-def hm-exch-op-def*
apply (*auto simp*: *pw-le-iff refine-pw-simps nres-rel-def*
hm-impl1-rel-def in-br-conv split: *prod.splits*)
apply (*auto simp*: *hm-impl1-α-def*)
unfolding *hm-impl1-rel-defs*
apply *auto*
done

term *hm-index-op*

definition *hm-index-op'* ≡ λ(*pq*,*ml*) *k*.
do {
ASSERT (*hm-impl1-invar* (*pq*,*ml*) ∧ *heapmap-α* (*hm-impl1-α* (*pq*,*ml*)) *k* ≠ *None* ∧ *k*∈*set pq*);
i ← *mop-list-index pq k*;
RETURN (*i*+*1*)
}
lemma *hm-index-op'-refine*: (*hm-index-op'*,*hm-index-op*)
∈ *hm-impl1-rel* → *nat-rel* → ⟨*nat-rel*⟩*nres-rel*
apply (*intro fun-relI nres-relI*)
unfolding *hm-index-op'-def hm-index-op-def hm-impl1-rel-defs*
apply (*auto simp*: *pw-le-iff refine-pw-simps heapmap-α-def split*: *if-split-asm*)
done

definition *hm-update-op'* **where**
hm-update-op' ≡ λ(*pq*,*ml*) *i v*. *do* {
ASSERT (*hm-valid* (*hm-impl1-α* (*pq*,*ml*)) *i* ∧ *hm-impl1-invar* (*pq*,*ml*));
k ← *mop-list-get pq* (*i* − *1*);
ml ← *mop-list-set ml k v*;
RETURN (*pq*, *ml*)
}
lemma *hm-update-op'-refine*: (*hm-update-op'*, *hm-update-op*) ∈ *hm-impl1-rel* → *nat-rel* → *Id* → ⟨*hm-impl1-rel*⟩*nres-rel*
apply (*intro fun-relI hm-impl1-nres-relI*[*OF hm-update-op-invar*])
unfolding *hm-update-op'-def hm-update-op-def*
apply (*auto simp*: *pw-le-iff refine-pw-simps nres-rel-def*
hm-impl1-rel-def in-br-conv split: *prod.splits*)
apply (*auto simp*: *hm-impl1-α-def*)
unfolding *hm-impl1-rel-defs*
apply (*auto simp*: *subset-code*(*1*))
done

term *hm-butlast-op*

lemma *hm-butlast-op-invar*: *hm-butlast-op hm* ≤$_n$ *SPEC hmr-invar*
unfolding *hm-butlast-op-def h.butlast-op-def*
apply *refine-vcg*

apply (*clarsimp-all simp*: *hmr-rel-defs map-butlast distinct-butlast*)
apply *safe*

apply (*auto simp*: *in-set-conv-nth nth-butlast*) []
apply (*metis Suc-pred len-greater-imp-nonempty length-greater-0-conv less-antisym*)

apply (*auto dest*: *in-set-butlastD*) []

apply (*metis One-nat-def append-butlast-last-id distinct-butlast last-conv-nth not-distinct-conv-prefix*)
done

definition *hm-butlast-op'* **where**
hm-butlast-op' ≡ λ(*pq*,*ml*). *do* {
ASSERT (*hmr-invar* (*hm-impl1-α* (*pq*,*ml*)));
pq ← *mop-list-butlast pq*;
RETURN (*pq*,*ml*)
}

lemma *set-butlast-distinct-conv*:
⟦*distinct l*⟧ ⟹ *set* (*butlast l*) = *set l* − {*last l*}
by (*cases l rule*: *rev-cases*; *auto*)

lemma *hm-butlast-op'-refine*: (*hm-butlast-op'*, *hm-butlast-op*) ∈ *hm-impl1-rel* → ⟨*hm-impl1-rel*⟩*nres-rel*
apply (*intro fun-relI hm-impl1-nres-relI*[*OF hm-butlast-op-invar*])
unfolding *hm-butlast-op'-def hm-butlast-op-def*
apply (*auto simp*: *pw-le-iff refine-pw-simps nres-rel-def*
hm-impl1-rel-def in-br-conv split: *prod.splits*)
apply (*auto simp*: *hm-impl1-α-def*)
unfolding *hm-impl1-rel-defs*
apply (*auto simp*: *restrict-map-def*) []

defer

apply (*auto dest*: *in-set-butlastD*) []
apply (*auto intro*!: *ext*
simp: *hmr-invar-def set-butlast-distinct-conv last-conv-nth*
dest: *in-set-butlastD*) []
done

definition *hm-append-op'*
where *hm-append-op'* ≡ λ(*pq*,*ml*) *k v*. *do* {
ASSERT (*k* ∉ *set pq* ∧ *k*<*maxsize*);
ASSERT (*hm-impl1-invar* (*pq*,*ml*));
pq ← *mop-list-append pq k*;
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*

apply (*intro frefI hm-impl1-nres-relI*[*OF hm-append-op-invar*])
unfolding *hm-append-op'-def hm-append-op-def*
apply (*auto simp*: *pw-le-iff refine-pw-simps nres-rel-def*
hm-impl1-rel-def in-br-conv split: *prod.splits*)
unfolding *hm-impl1-rel-defs*
apply (*auto simp*: *restrict-map-def hmr-invar-def split*: *prod.splits if-split-asm*)

done

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*
by (*intro fun-relI*) (*simp add*: *h.parent-def*)
lemmas [*sepref-import-param*] = *param-parent*
sepref-register *h.parent*

lemma *param-left*: (*h.left*,*h.left*) ∈ *Id* → *Id* **by** *simp*
lemmas [*sepref-import-param*] = *param-left*
sepref-register *h.left*

lemma *param-right*: (*h.right*,*h.right*) ∈ *Id* → *Id* **by** *simp*
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* **by** *simp*
lemmas [*sepref-import-param*] = *param-prio-le*

lemma *param-prio-lt*: ((<), (<)) ∈ *prio-rel* → *prio-rel* → *bool-rel* **by** *simp*
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*
unfolding *hm-length'-def hm-impl2-rel-def*

by *sepref*
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*
unfolding *hm-key-of-op'-def hm-impl2-rel-def*
by *sepref*
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-impl2-rel*^{k} *a* *nat-assn*^{k} →$_a$ *option-assn id-assn*)
unfolding *hm-lookup-op'-def hm-impl2-rel-def*
by *sepref*
lemmas [*sepref-fr-rules*] =
hm-lookup-impl.refine[*FCOMP hm-lookup-op'-refine*]
sepref-register *hm-lookup*::(*nat*,'*v*) *ahm* ⇒ -

sepref-definition *hm-exch-op-impl* **is** *uncurry2 hm-exch-op'* :: *hm-impl2-rel*^{d} *a* *nat-assn*^{k} *a* *nat-assn*^{k} →$_a$ *hm-impl2-rel*
unfolding *hm-exch-op'-def hm-impl2-rel-def*
by *sepref*
lemmas [*sepref-fr-rules*] = *hm-exch-op-impl.refine*[*FCOMP hm-exch-op'-refine*]
sepref-register *hm-exch-op*::(*nat*,'*v*) *ahm* ⇒ -

sepref-definition *hm-index-op-impl* **is** *uncurry hm-index-op'* :: *hm-impl2-rel*^{k} *a* *id-assn*^{k} →$_a$ *id-assn*
unfolding *hm-index-op'-def hm-impl2-rel-def*
by *sepref*
lemmas [*sepref-fr-rules*] = *hm-index-op-impl.refine*[*FCOMP hm-index-op'-refine*]
sepref-register *hm-index-op*::(*nat*,'*v*) *ahm* ⇒ -

sepref-definition *hm-update-op-impl* **is** *uncurry2 hm-update-op'* :: *hm-impl2-rel*^{d} *a* *id-assn*^{k} *a* *id-assn*^{k} →$_a$ *hm-impl2-rel*
unfolding *hm-update-op'-def hm-impl2-rel-def*
by *sepref*
lemmas [*sepref-fr-rules*] = *hm-update-op-impl.refine*[*FCOMP hm-update-op'-refine*]
sepref-register *hm-update-op*::(*nat*,'*v*) *ahm* ⇒ -

sepref-definition *hm-butlast-op-impl* **is** *hm-butlast-op'* :: *hm-impl2-rel*^{d} →$_a$ *hm-impl2-rel*

unfolding *hm-butlast-op′-def hm-impl2-rel-def* **by** *sepref*
lemmas [*sepref-fr-rules*] = *hm-butlast-op-impl.refine*[*FCOMP hm-butlast-op′-refine*]
sepref-register *hm-butlast-op*::(*nat*,′*v*) *ahm* ⇒ -

sepref-definition *hm-append-op-impl* **is** *uncurry2 hm-append-op′* :: *hm-impl2-rel*^{d} $*_a$ *id-assn*^{k} $*_a$ *id-assn*^{k} $\rightarrow_a$ *hm-impl2-rel*
unfolding *hm-append-op′-def hm-impl2-rel-def*
apply (*rewrite array-fold-custom-replicate*)
by *sepref*
lemmas [*sepref-fr-rules*] = *hm-append-op-impl.refine*[*FCOMP hm-append-op′-refine*]
sepref-register *hm-append-op*::(*nat*,′*v*) *ahm* ⇒ -

3.12.2 Auxiliary Operations

lemmas [*intf-of-assn*] = *intf-of-assnI*[**where** *R*=*hm-impl-rel* :: (*nat*,′*v*) *ahm* ⇒ - **and** ′*a*=′*v i-hma*]

sepref-definition *hm-valid-impl* **is** *uncurry* (*RETURN oo hm-valid*) :: *hm-impl-rel*^{k} $*_a$ *nat-assn*^{k} $\rightarrow_a$ *bool-assn*
unfolding *hm-valid-def*[*abs-def*]
by *sepref*
lemmas [*sepref-fr-rules*] = *hm-valid-impl.refine*
sepref-register *hm-valid*::(*nat*,′*v*) *ahm* ⇒ -

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*
apply (*intro fun-relI nres-relI*)
unfolding *hm-the-lookup-op′-def hm-the-lookup-op-def*
apply *refine-vcg*
apply (*auto simp*: *hm-impl1-rel-defs heapmap-α-def hmr-invar-def split*: *if-split-asm*)
done

sepref-definition *hm-the-lookup-op-impl* **is** *uncurry hm-the-lookup-op′* :: *hm-impl2-rel*^{k} $*_a$ *id-assn*^{k} $\rightarrow_a$ *id-assn*
unfolding *hm-the-lookup-op′-def*[*abs-def*] *hm-impl2-rel-def*
by *sepref*
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* :: $hm\text{-}impl\text{-}rel^k *_a id\text{-}assn^k \rightarrow_a id\text{-}assn$
unfolding *hm-val-of-op-def* **by** *sepref*
lemmas [*sepref-fr-rules*] = *hm-val-of-op-impl.refine*
sepref-register *hm-val-of-op*::(*nat*,'*v*) *ahm* ⇒ -

sepref-definition *hm-prio-of-op-impl* **is** *uncurry* (*PR-CONST hm-prio-of-op*) :: $hm\text{-}impl\text{-}rel^k *_a id\text{-}assn^k \rightarrow_a id\text{-}assn$
unfolding *hm-prio-of-op-def*[*abs-def*] *PR-CONST-def* **by** *sepref*
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*
by *simp*

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*) :: $hm\text{-}impl\text{-}rel^d *_a nat\text{-}assn^k \rightarrow_a hm\text{-}impl\text{-}rel$
unfolding *hm-swim-op-def*[*abs-def*] *PR-CONST-def*
using [[*goals-limit* = *1*]]
by *sepref*
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*
by *simp*

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*) :: $hm\text{-}impl\text{-}rel^d *_a nat\text{-}assn^k \rightarrow_a hm\text{-}impl\text{-}rel$
unfolding *hm-sink-op-def*[*abs-def*] *PR-CONST-def*
by *sepref*
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* **by** *simp*

sepref-definition *hm-repair-op-impl* **is** *uncurry* (*PR-CONST hm-repair-op*) :: $hm\text{-}impl\text{-}rel^d *_a nat\text{-}assn^k \rightarrow_a hm\text{-}impl\text{-}rel$
unfolding *hm-repair-op-def*[*abs-def*] *PR-CONST-def*
by *sepref*
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*
by *simp*

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.
unfolding *hm-rel-def* **by** *simp*

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*) **by** *simp*

lemma *hm-fold-custom-empty-sz*:
op-map-empty = *op-hm-empty-sz sz*
Map.empty = *op-hm-empty-sz sz*
by *auto*

sepref-definition *hm-empty-op-impl* **is** *uncurry0 hm-empty-op'* :: *unit-assn*^{k} →$_a$ *hm-impl2-rel*
unfolding *hm-empty-op'-def hm-impl2-rel-def array.fold-custom-empty*
by *sepref*

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*
unfolding *hm-insert-op-def*
by *sepref*

sepref-definition *hm-is-empty-op-impl* **is** *hm-is-empty-op* :: *hm-impl-rel*^{k} →$_a$ *bool-assn*
unfolding *hm-is-empty-op-def*
by *sepref*

sepref-definition *hm-lookup-op-impl* **is** *uncurry hm-lookup-op* :: *id-assn*^{k}*$_a$*hm-impl-rel*^{k} →$_a$ *option-assn id-assn*
unfolding *hm-lookup-op-def* **by** *sepref*

sepref-definition *hm-contains-key-impl* **is** *uncurry hm-contains-key-op'* :: *id-assn*^{k}*$_a$*hm-impl2-rel*^{k} →$_a$ *bool-assn*
unfolding *hm-contains-key-op'-def hm-impl2-rel-def*
by *sepref*

sepref-definition *hm-decrease-key-op-impl* **is** *uncurry2 hm-decrease-key-op* :: $id\text{-}assn^k *_a id\text{-}assn^k *_a hm\text{-}impl\text{-}rel^d \rightarrow_a hm\text{-}impl\text{-}rel$
unfolding *hm-decrease-key-op-def* **by** *sepref*

sepref-definition *hm-increase-key-op-impl* **is** *uncurry2 hm-increase-key-op* :: $id\text{-}assn^k *_a id\text{-}assn^k *_a hm\text{-}impl\text{-}rel^d \rightarrow_a hm\text{-}impl\text{-}rel$
unfolding *hm-increase-key-op-def* **by** *sepref*

sepref-definition *hm-change-key-op-impl* **is** *uncurry2 hm-change-key-op* :: $id\text{-}assn^k *_a id\text{-}assn^k *_a hm\text{-}impl\text{-}rel^a \rightarrow_a hm\text{-}impl\text{-}rel$
unfolding *hm-change-key-op-def* **by** *sepref*

sepref-definition *hm-pop-min-op-impl* **is** *hm-pop-min-op* :: $hm\text{-}impl\text{-}rel^d \rightarrow_a prod\text{-}assn\ (prod\text{-}assn\ nat\text{-}assn\ id\text{-}assn)\ hm\text{-}impl\text{-}rel$
unfolding *hm-pop-min-op-def*[*abs-def*]
by *sepref*

sepref-definition *hm-remove-op-impl* **is** *uncurry hm-remove-op* :: $id\text{-}assn^k *_a hm\text{-}impl\text{-}rel^d \rightarrow_a hm\text{-}impl\text{-}rel$
unfolding *hm-remove-op-def*[*abs-def*] **by** *sepref*

sepref-definition *hm-peek-min-op-impl* **is** *hm-peek-min-op* :: $hm\text{-}impl\text{-}rel^k \rightarrow_a prod\text{-}assn\ nat\text{-}assn\ id\text{-}assn$
unfolding *hm-peek-min-op-def*[*abs-def*] *hm-kv-of-op-def*
by *sepref*

Setup of Refinements

sepref-decl-impl (*no-register*) *hm-empty*:
hm-empty-op-impl.refine[*FCOMP hm-empty-op'-refine*, *FCOMP hm-empty-aref*]
.

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'*
unfolding *IS-BELOW-ID-def*
apply (*parametricity*; *auto simp*: *single-valued-below-Id*)
done

sepref-decl-impl *hm-is-empty*: *hm-is-empty-op-impl.refine*[*FCOMP hm-is-empty-op-aref*]
.
sepref-decl-impl *hm-lookup*: *hm-lookup-op-impl.refine*[*FCOMP hm-lookup-op-aref*]
.

sepref-decl-impl *hm-contains-key*:
hm-contains-key-impl.refine[*FCOMP hm-contains-key-op'-refine*, *FCOMP hm-contains-key-op-aref*]
.

sepref-decl-impl (*ismop*) *hm-decrease-key*: *hm-decrease-key-op-impl.refine*[*FCOMP hm-decrease-key-op-aref*] .
sepref-decl-impl (*ismop*) *hm-increase-key*: *hm-increase-key-op-impl.refine*[*FCOMP hm-increase-key-op-aref*] .
sepref-decl-impl (*ismop*) *hm-change-key*: *hm-change-key-op-impl.refine*[*FCOMP hm-change-key-op-aref*] .

sepref-decl-impl (*ismop*) *hm-remove*: *hm-remove-op-impl.refine*[*FCOMP hm-remove-op-aref*]
.

sepref-decl-impl (*ismop*) *hm-pop-min*: *hm-pop-min-op-impl.refine*[*FCOMP hm-pop-min-op-aref*]
.
sepref-decl-impl (*ismop*) *hm-peek-min*: *hm-peek-min-op-impl.refine*[*FCOMP hm-peek-min-op-aref*]
.

— Realized as generic algorithm. Note that we use *id-assn* for the elements.
sepref-definition *hm-upd-op-impl* **is** *uncurry2* (*RETURN ooo op-map-update*) ::
$[\lambda((k,\text{-}),\text{-}).\ k<maxsize]_a$ $id\text{-}assn^k$ $*_a$ $id\text{-}assn^k$ $*_a$ $(hm\text{-}rel\ id\text{-}assn\ id\text{-}assn)^d \to hm\text{-}rel$
id-assn id-assn
unfolding *op-pm-set-gen-impl* **by** *sepref*

sepref-decl-impl *hm-upd-op-impl.refine*[*unfolded hm-rel-id-conv*] **uses** *op-map-update-fref'*
unfolding *IS-BELOW-ID-def*
apply (*parametricity*; *auto simp*: *single-valued-below-Id*)
done

end
end

interpretation *hm*: *map-custom-empty PR-CONST* (*op-hm-empty-sz maxsize*)
apply *unfold-locales* **by** *simp*

lemma *op-hm-empty-sz-hnr*[*sepref-fr-rules*]:
(*uncurry0* (*hm-empty-op-impl maxsize*), *uncurry0* (*RETURN* (*PR-CONST* (*op-hm-empty-sz maxsize*)))) ∈ $unit\text{-}assn^k \to_a hm\text{-}rel\ maxsize\ prio\ K\ V$
using *hm-empty-hnr* **by** *simp*

3.12.4 Manual fine-tuning of code-lemmas

declare [[*code drop*: *hm-change-key-op-impl*]]

declare [[*code drop*: *hm-upd-op-impl*]]

declare [[*code drop*: *hm-pop-min-op-impl*]]

context
notes [*simp del*] = *CNV-def efficient-nat-div2*
begin

lemma *nested-case-bind*:
(*case p of* (*a*,*b*) ⇒ *bind* (*case a of* (*a1*,*a2*) ⇒ *m a b a1 a2*) (*f a b*))
= (*case p of* ((*a1*,*a2*),*b*) ⇒ *bind* (*m* (*a1*,*a2*) *b a1 a2*) (*f* (*a1*,*a2*) *b*))
(*case p of* (*a*,*b*) ⇒ *bind* (*case b of* (*b1*,*b2*) ⇒ *m a b b1 b2*) (*f a b*))
= (*case p of* (*a*,*b1*,*b2*) ⇒ *bind* (*m a* (*b1*,*b2*) *b1 b2*) (*f a* (*b1*,*b2*)))
by (*simp-all split*: *prod.splits*)

lemma *it-case*: (*case p of* (*a*,*b*) ⇒ *f p a b*) = (*case p of* (*a*,*b*) ⇒ *f* (*a*,*b*) *a b*)
by (*auto split*: *prod.split*)

lemma *c2l*: (*case p of* (*a*,*b*) ⇒ *bind* (*m a b*) (*f a b*)) =
do { *let* (*a*,*b*) = *p*; *bind* (*m a b*) (*f a b*)} **by** *simp*

lemma *bind-Let*: *do* { *x* ← *do* { *let y* = *v*; (*f y* :: *'a Heap*)}; *g x* } = *do* { *let y*=*v*; *x* ← *f y*; *g x* } **by** *auto*
lemma *bind-case*: *do* { *x* ← (*case y of* (*a*,*b*) ⇒ *f a b*); (*g x* :: *'a Heap*) } = *do* { *let* (*a*,*b*) = *y*; *x* ← *f a b*; *g x* }
by (*auto split*: *prod.splits*)

lemma *bind-case-mvup*: *do* { *x* ← *f*; *case y of* (*a*,*b*) ⇒ *g a b x* }
= *do* { *let* (*a*,*b*) = *y*; *x* ← *f*; (*g a b x* :: *'a Heap*) }
by (*auto split*: *prod.splits*)

lemma *if-case-mvup*: (*if b then case p of* (*x1*,*x2*) ⇒ *f x1 x2 else e*)
= (*case p of* (*x1*,*x2*) ⇒ *if b then f x1 x2 else e*) **by** *auto*

lemma *nested-case*: (*case p of* (*a*,*b*) ⇒ (*case p of* (*c*,*d*) ⇒ *f a b c d*)) =
(*case p of* (*a*,*b*) ⇒ *f a b a b*)
by (*auto split*: *prod.split*)

lemma *split-prod-bound*: (λ*p*. *f p*) = (λ(*a*,*b*). *f* (*a*,*b*)) **by** *auto*

lemma *bpc-conv*: *do* { (*a*,*b*) ← (*m*::(-∗-) *Heap*); *f a b* } = *do* {
ab ← (*m*);
f (*fst ab*) (*snd ab*)
}
apply (*subst* (*2*) *split-prod-bound*)
by *simp*

lemma *it-case-pp*: (*case p of* ((*p1*,*p2*)) ⇒ *case p of* ((*p1'*,*p2'*)) ⇒ *f p1 p2 p1' p2'*)
= (*case p of* ((*p1*,*p2*)) ⇒ *f p1 p2 p1 p2*)
by (*auto split*: *prod.split*)

lemma *it-case-ppp*: (*case p of* ((*p1*,*p2*),*p3*) ⇒ *case p of* ((*p1'*,*p2'*),*p3'*) ⇒ *f p1 p2 p3 p1' p2' p3'*)
= (*case p of* ((*p1*,*p2*),*p3*) ⇒ *f p1 p2 p3 p1 p2 p3*)
by (*auto split*: *prod.split*)

lemma *it-case-pppp*: (*case a1 of*
(((*a*, *b*), *c*), *d*) ⇒
case a1 of
(((*a'*, *b'*), *c'*), *d'*) ⇒ *f a b c d a' b' c' d'*) =
(*case a1 of*
(((*a*, *b*), *c*), *d*) ⇒ *f a b c d a b c d*)
by (*auto split*: *prod.splits*)

private lemmas *inlines* = *hm-append-op-impl-def ial-append-def*
marl-length-def marl-append-def hm-length-impl-def ial-length-def
hm-valid-impl-def hm-prio-of-op-impl-def hm-val-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

declare [[*code drop*: *hm-insert-op-impl*]]

schematic-goal [*code*]: *hm-insert-op-impl maxsize prio hm k v* = *?f*
unfolding *hm-insert-op-impl-def*
apply (*rule CNV-eqD*)
apply (*simp add*: *inlines cong*: *if-cong*)
by (*rule CNV-I*)

schematic-goal *hm-swim-op-impl prio hm i* ≡ *?f*
unfolding *hm-swim-op-impl-def*
apply (*rule eq-reflection*)
apply (*rule CNV-eqD*)
apply (*simp add*: *inlines efficient-nat-div2*
cong: *if-cong*)
by (*rule CNV-I*)

lemma *hm-swim-op-impl-code*: *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)
  unfolding hm-swim-op-impl-def
  apply (rule eq-reflection)
  apply (simp add: inlines efficient-nat-div2 Let-def
    cong: if-cong)
  done

prepare-code-thms hm-swim-op-impl-code

schematic-goal hm-sink-opt-impl-code: hm-sink-op-impl prio hm i ≡ ?f
  unfolding hm-sink-op-impl-def
  apply (rule eq-reflection)
  apply (rule CNV-eqD)
  apply (simp add: inlines
    cong: if-cong)
  by (rule CNV-I)

prepare-code-thms 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 ≡ 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)
          })
  unfolding hm-change-key-op-impl-def
  apply (rule eq-reflection)
  apply (simp add: inlines hm-contains-key-impl-def ial-contains-def
    hm-change-key-op-impl-def hm-index-op-impl-def hm-update-op-impl-def
    ial-index-def
    cong: if-cong split: prod.splits)
  done
```

schematic-goal *hm-set-opt-impl-code*[*code*]: *hm-upd-op-impl maxsize prio hm k v* ≡ *?f*
unfolding *hm-upd-op-impl-def*
apply (*rule eq-reflection*)
apply (*rule CNV-eqD*)
apply (*simp add*: *inlines hm-contains-key-impl-def ial-contains-def hm-change-key-op-impl-def hm-index-op-impl-def hm-update-op-impl-def ial-index-def cong*: *if-cong*)
by (*rule CNV-I*)

schematic-goal *hm-pop-min-opt-impl-code*[*code*]: *hm-pop-min-op-impl prio hm* ≡ *?f*
unfolding *hm-pop-min-op-impl-def*
apply (*rule eq-reflection*)
apply (*rule CNV-eqD*)
apply (*simp add*: *inlines hm-contains-key-impl-def ial-contains-def hm-change-key-op-impl-def hm-index-op-impl-def hm-update-op-impl-def hm-butlast-op-impl-def ial-butlast-def ial-index-def cong*: *if-cong*)
by (*rule CNV-I*)

end

export-code
hm-empty-op-impl
hm-insert-op-impl
hm-is-empty-op-impl
hm-lookup-op-impl
hm-contains-key-impl
hm-decrease-key-op-impl
hm-increase-key-op-impl
hm-change-key-op-impl
hm-upd-op-impl
hm-pop-min-op-impl
hm-remove-op-impl
hm-peek-min-op-impl
checking *SML-imp*

end

3.13 Matrices

theory *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* **unfolding** *mtx-rel-def* **by** *auto*

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)*
by *simp*

3.13.2 Operations

definition *op-mtx-new* :: *'a mtx* ⇒ *'a mtx* **where** [*simp*]: *op-mtx-new c* ≡ *c*

sepref-decl-op (*no-def*) *mtx-new*: *op-mtx-new* :: (*nat-rel*×$_r$*nat-rel* → *A*) → ⟨*A*⟩*mtx-rel*
apply (*rule fref-ncI*) **unfolding** *op-mtx-new-def*[*abs-def*] *mtx-rel-def*
by *parametricity*

lemma *mtx-init-adhoc-frame-match-rule*[*sepref-frame-match-rules*]:
hn-val (*nat-rel*×$_r$*nat-rel* → *A*) *x y* ⟹$_t$ *hn-val* (*nat-rel*×$_r$*nat-rel* → *the-pure* (*pure A*)) *x y*
by *simp*

definition *op-mtx-copy* :: *'a mtx* ⇒ *'a mtx* **where** [*simp*]: *op-mtx-copy c* ≡ *c*

sepref-decl-op (*no-def*) *mtx-copy*: *op-mtx-copy* :: ⟨*A*⟩*mtx-rel* → ⟨*A*⟩*mtx-rel* .

sepref-decl-op *mtx-get*: λ(*c*::*'a mtx*) *ij*. *c ij* :: ⟨*A*⟩*mtx-rel* → (*nat-rel*×$_r$*nat-rel*) → *A*
apply (*rule fref-ncI*) **unfolding** *mtx-rel-def*
by *parametricity*

sepref-decl-op *mtx-set*: *fun-upd*::*'a mtx* ⇒ - :: ⟨*A*⟩*mtx-rel* → (*nat-rel*×$_r$*nat-rel*) → *A* → ⟨*A*⟩*mtx-rel*
apply (*rule fref-ncI*)
unfolding *mtx-rel-def*
proof *goal-cases* **case** *1*
have [*param*]: ((=), (=)) ∈ *nat-rel* ×$_r$ *nat-rel* → *nat-rel* ×$_r$ *nat-rel* → *bool-rel* **by** *simp*
show *?case* **by** *parametricity*
qed

definition *mtx-nonzero* :: - *mtx* ⇒ (*nat*×*nat*) *set* **where** *mtx-nonzero m* ≡ {(*i*,*j*). *m* (*i*,*j*)≠*0*}

sepref-decl-op *mtx-nonzero*: *mtx-nonzero* :: $\langle A\rangle$*mtx-rel* $\to$ $\langle$*nat-rel*$\times_r$*nat-rel*$\rangle$*set-rel*
  where *IS-ID* (*A*::(-$\times$(-::*zero*)) *set*)
proof *goal-cases*
  case *1*
  assume *IS-ID A*
  hence *U*: *A*=*Id* **by** (*simp only*: *IS-ID-def*)
  have [*param*]: ((=),(=))$\in$*A*$\to$*A*$\to$*bool-rel* **using** *U* **by** *simp*
  show *?case*
    apply (*rule fref-ncI*)
    unfolding *mtx-rel-def*
    apply *parametricity*
    unfolding *U* **by** *simp-all*
qed

3.13.3 Patterns

lemma *pat-amtx-get*: *c*$*e*≡*op-mtx-get*$'*c*$'*e* **by** *simp*
lemma *pat-amtx-set*: *fun-upd*$*c*$*e*$*v*≡*op-mtx-set*$'*c*$'*e*$'*v* **by** *simp*

lemmas *amtx-pats*[*pat-rules*] = *pat-amtx-get pat-amtx-set*

3.13.4 Pointwise Operations

Auxiliary Definitions and Lemmas

locale *pointwise-op* =
  fixes *f* :: *'p* $\Rightarrow$ *'s* $\Rightarrow$ *'s*
  fixes *q* :: *'s* $\Rightarrow$ *'p* $\Rightarrow$ *'a*
  assumes *upd-indep1*[*simp*, *intro*]: *p*$\neq$*p'* $\Longrightarrow$ *q* (*f p s*) *p'* = *q s p'*
  assumes *upd-indep2*[*simp*, *intro*]: *p*$\neq$*p'* $\Longrightarrow$ *q* (*f p* (*f p' s*)) *p* = *q* (*f p s*) *p*
begin
  lemma *pointwise-upd-fold*: *distinct ps* $\Longrightarrow$
    q (*fold f ps s*) *p* = (*if p*$\in$*set ps then q* (*f p s*) *p else q s p*)
    by (*induction ps arbitrary*: *s*) *auto*

end

lemma *pointwise-fun-fold*:
  fixes *f* :: *'a* $\Rightarrow$ (*'a* $\Rightarrow$ *'b*) $\Rightarrow$ (*'a* $\Rightarrow$ *'b*)
  fixes *s* :: *'a* $\Rightarrow$ *'b*
  assumes *indep1*: $\bigwedge$*x x' s*. *x* $\neq$ *x'* $\Longrightarrow$ *f x s x'* = *s x'*
  assumes *indep2*: $\bigwedge$*x x' s*. *x* $\neq$ *x'* $\Longrightarrow$ *f x* (*f x' s*) *x* = *f x s x*
  assumes [*simp*]: *distinct xs*
  shows *fold f xs s x* = (*if x* $\in$ *set xs then f x s x else s x*)
proof −
  interpret *pointwise-op f* λ*s*. *s*
    by *unfold-locales fact+*

  show *?thesis*
    using *pointwise-upd-fold*[*of xs s x*]

```
    by auto
  qed

  lemma list-prod-divmod-eq: List.product [0..<M] [0..<N] = map (λi. (i div N,
i mod N)) [0..<N*M]
  proof −
    have [simp]: i < m*n ⟹ (i::nat) div n < m for i m n
    by (metis mult.commute div-eq-0-iff div-mult2-eq gr-implies-not-zero mult-not-zero)

    have [simp]: i<N*M ⟹ N>0 ∧ M>0 for i
      by (cases N; cases M; auto)

    show ?thesis
      by (rule nth-equalityI) (auto simp add: product-nth algebra-simps)
  qed


  lemma nfoldli-prod-divmod-conv:
    nfoldli (List.product [0..<N] [0..<M]) ctd (λ(i,j). f i j) = nfoldli [0..<N*M]
ctd (λi. f (i div M) (i mod M))
    apply (intro ext)
    apply (subst list-prod-divmod-eq)
    apply (simp add: nfoldli-map)
    apply (fo-rule cong)+
    apply (auto simp: algebra-simps)
    done

  lemma nfoldli-prod-divmod-conv':
    nfoldli [0..<M] ctd (λi. nfoldli [0..<N] ctd (f i)) = nfoldli [0..<N*M] ctd (λi.
f (i div N) (i mod N))
    apply (intro ext)
    apply (subst nfoldli-nfoldli-prod-conv)
    by (simp add: nfoldli-prod-divmod-conv algebra-simps)

  lemma foldli-prod-divmod-conv':
    foldli [0..<M] ctd (λi. foldli [0..<N] ctd (f i)) = foldli [0..<N*M] ctd (λi. f
(i div N) (i mod N))
    (is ?lhs=?rhs)
  proof −
    have RETURN (?lhs s) = RETURN (?rhs s) for s
      apply (subst foldli-eq-nfoldli)+
      apply (subst nfoldli-prod-divmod-conv')
      ..
    thus ?thesis by auto
  qed

  lemma fold-prod-divmod-conv': fold (λi. fold (f i) [0..<N]) [0..<M] = fold (λi.
f (i div N) (i mod N)) [0..<N*M]
    using foldli-prod-divmod-conv'[of M λ-. True N f, THEN fun-cong]
```

apply (*intro ext*)
apply (*simp add*: *foldli-foldl foldl-conv-fold*)
done

lemma *mtx-nonzero-cases*[*consumes 0*, *case-names nonzero zero*]:
obtains (*i*,*j*)∈*mtx-nonzero m* | *m* (*i*,*j*) = *0*
by (*auto simp*: *mtx-nonzero-def*)

Unary Pointwise

definition *mtx-pointwise-unop* :: (*nat*×*nat* ⇒ *'a* ⇒ *'a*) ⇒ *'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*) **by** *simp*
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*
apply (*rule ext*)
unfolding *opr-fold-impl-def*
apply (*simp add*: *fold-fold-prod-conv*)
apply (*subst pointwise-fun-fold*)
apply (*auto simp*: *mtx-pointwise-unop-def distinct-product*) [*3*]
apply *clarsimp*
subgoal for *a b*
apply (*cases a b m rule*: *mtx-nonzero-cases*)
using *assms*
apply (*auto simp*: *mtx-pointwise-unop-def*)
done
done

lemma *opr-fold-impl-refine*: (*opr-fold-impl*, *mtx-pointwise-unop f*) ∈ [λ*m*. *mtx-nonzero m* ⊆ {*0*..<*N*}×{*0*..<*M*}]$_f$ *Id* → *Id*
apply (*rule frefI*)

using *opr-fold-impl-eq*
by *auto*

end

locale *mtx-pointwise-unop-gen-impl* = *mtx-pointwise-unop-loc* +
fixes *assn* :: *'a mtx* ⇒ *'i* ⇒ *assn*
fixes *A* :: *'a* ⇒ *'ai* ⇒ *assn*
fixes *get-impl* :: *'i* ⇒ *nat*×*nat* ⇒ *'ai Heap*
fixes *set-impl* :: *'i* ⇒ *nat*×*nat* ⇒ *'ai* ⇒ *'i Heap*
fixes *fi* :: *nat*×*nat* ⇒ *'ai* ⇒ *'ai Heap*
assumes *assn-range*: *rdomp assn m* ⟹ *mtx-nonzero m* ⊆ {*0*..<*N*}×{*0*..<*M*}
assumes *get-impl-hnr*:
(*uncurry get-impl*,*uncurry* (*RETURN oo op-mtx-get*)) ∈ *assn*^{k} *a* (*prod-assn* (*nbn-assn N*) (*nbn-assn M*))^k →$_a$ *A*
assumes *set-impl-hnr*:
(*uncurry2 set-impl*,*uncurry2* (*RETURN ooo op-mtx-set*)) ∈ *assn*^{d} *a* (*prod-assn* (*nbn-assn N*) (*nbn-assn M*))^k *a* *A*^{k} →$_a$ *assn*
assumes *fi-hnr*:
(*uncurry fi*,*uncurry* (*RETURN oo f*)) ∈ (*prod-assn nat-assn nat-assn*)^k *a* *A*^{k} →$_a$ *A*
begin

lemma *this-loc*: *mtx-pointwise-unop-gen-impl N M f assn A get-impl set-impl fi*
by *unfold-locales*

context
notes [[*sepref-register-adhoc f N M*]]
notes [*intf-of-assn*] = *intf-of-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*
unfolding *opr-fold-impl-def*
supply [[*goals-limit* = *1*]]
by *sepref*
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*
unfolding *PR-CONST-def*
apply (*rule hfref-weaken-pre'*[*OF - mtx-pointwise-unnop-fold-impl1.refine*[*OF this-loc*,*FCOMP opr-fold-impl-refine*]])
by (*simp add*: *assn-range*)

end

Binary Pointwise

definition *mtx-pointwise-binop* :: ('*a* ⇒ '*a* ⇒ '*a*) ⇒ '*a mtx* ⇒ '*a mtx* ⇒ '*a mtx*
where
mtx-pointwise-binop f m n ≡ λ(*i*,*j*). *f* (*m*(*i*,*j*)) (*n*(*i*,*j*))
context fixes *f* :: '*a* ⇒ '*a* ⇒ '*a* **begin**
sepref-register *PR-CONST* (*mtx-pointwise-binop f*) :: '*a i-mtx* ⇒ '*a i-mtx* ⇒ '*a i-mtx*
lemma [*def-pat-rules*]: *mtx-pointwise-binop*$*f* ≡ *UNPROTECT* (*mtx-pointwise-binop f*) **by** *simp*
end

locale *mtx-pointwise-binop-loc* =
fixes *N* :: *nat* **and** *M* :: *nat*
fixes *f* :: '*a*::{*zero*} ⇒ '*a* ⇒ '*a*
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*
apply (*rule ext*)
unfolding *opr-fold-impl-def*
apply (*simp add*: *fold-fold-prod-conv*)
apply (*subst pointwise-fun-fold*)
apply (*auto simp*: *mtx-pointwise-binop-def distinct-product*) [*3*]
apply *clarsimp*
subgoal for *a b*
apply (*cases a b m rule*: *mtx-nonzero-cases*; *cases a b n rule*: *mtx-nonzero-cases*)
using *assms*
apply (*auto simp*: *mtx-pointwise-binop-def*)
done
done

lemma *opr-fold-impl-refine*: (*uncurry opr-fold-impl*, *uncurry* (*mtx-pointwise-binop f*)) ∈ [λ(*m*,*n*). *mtx-nonzero m* ⊆ {*0*..<*N*}×{*0*..<*M*} ∧ *mtx-nonzero n* ⊆ {*0*..<*N*}×{*0*..<*M*}]$_f$ *Id*×$_r$*Id* → *Id*
apply (*rule frefI*)
using *opr-fold-impl-eq*
by *auto*

end

locale *mtx-pointwise-binop-gen-impl* = *mtx-pointwise-binop-loc* +
fixes *assn* :: '*a mtx* ⇒ '*i* ⇒ *assn*
fixes *A* :: '*a* ⇒ '*ai* ⇒ *assn*
fixes *get-impl* :: '*i* ⇒ *nat*×*nat* ⇒ '*ai Heap*

fixes *set-impl* :: *'i* ⇒ *nat*×*nat* ⇒ *'ai* ⇒ *'i Heap*
fixes *fi* :: *'ai* ⇒ *'ai* ⇒ *'ai Heap*
assumes *assn-range*: *rdomp assn m* ⟹ *mtx-nonzero m* ⊆ {*0..<N*}×{*0..<M*}
assumes *get-impl-hnr*:
(*uncurry get-impl*,*uncurry* (*RETURN oo op-mtx-get*)) ∈ $assn^k$ ∗$_a$ (*prod-assn* (*nbn-assn N*) (*nbn-assn M*))^k →$_a$ *A*
assumes *set-impl-hnr*:
(*uncurry2 set-impl*,*uncurry2* (*RETURN ooo op-mtx-set*)) ∈ $assn^d$ ∗$_a$ (*prod-assn* (*nbn-assn N*) (*nbn-assn M*))^k ∗$_a$ A^k →$_a$ *assn*
assumes *fi-hnr*:
(*uncurry 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*
by *unfold-locales*

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$ →$_a$ *assn*
unfolding *opr-fold-impl-def*[*abs-def*]
by *sepref*

end

concrete-definition (**in** −) *mtx-pointwise-binop-fold-impl1*
uses *mtx-pointwise-binop-gen-impl.opr-fold-impl1.refine-raw* **is** (*uncurry ?f*,-)∈-
prepare-code-thms (**in** −) *mtx-pointwise-binop-fold-impl1-def*

lemma *op-hnr*[*sepref-fr-rules*]: (*uncurry* (*mtx-pointwise-binop-fold-impl1 N M get-impl set-impl fi*), *uncurry* (*RETURN oo PR-CONST* (*mtx-pointwise-binop f*))) ∈ $assn^d$ ∗$_a$ $assn^k$ →$_a$ *assn*
unfolding *PR-CONST-def*
apply (*rule hfref-weaken-pre'*[*OF - mtx-pointwise-binop-fold-impl1.refine*[*OF this-loc*,*FCOMP opr-fold-impl-refine*]])
apply (*auto dest*: *assn-range*)
done

end

Compare Pointwise

definition *mtx-pointwise-cmpop* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a mtx* ⇒ *'a mtx* ⇒ *bool* **where**

```
    mtx-pointwise-cmpop f g m n ≡ (∀ i j. f (m(i,j)) (n(i,j))) ∧ (∃ i j. g (m(i,j))
(n(i,j)))
  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) by simp
  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
    by (auto simp: mtx-nonzero-def)


  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]) (λs. s≠2) (λ(i,j) s. do {
        if f (m(i,j)) (n(i,j)) then
          if s=0 then
            if g (m(i,j)) (n(i,j)) then RETURN 1 else RETURN s
          else
            RETURN s

        else RETURN 2
      }) (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 −
      have (∀ i<N. ∀ j<M. f (m (i, j)) (n (i, j))) ⟹ f (m (i, j)) (n (i, j))) for i j
        apply (cases i<N; cases j<M)
        using assms by (auto simp: mtx-nonzeroD)
       moreover have g (m (i, j)) (n (i, j)) ⟹ (∃ i<N. ∃ j<M. g (m (i, j)) (n
(i, j))) for i j
        apply (cases i<N; cases j<M)
        using assms by (auto simp: mtx-nonzeroD)
      ultimately have EQ: mtx-pointwise-cmpop f g m n
          ⟷ (∀ i<N. ∀ j<M. f (m(i,j)) (n(i,j))) ∧ (∃ i<N. ∃ j<M. g (m(i,j))
(n(i,j)))
```

```
      unfolding mtx-pointwise-cmpop-def by meson

    have aux: List.product [0..<N] [0..<M] = l1 @ (i, j) # l2 ⟹ i<N ∧ j<M
for l1 i j l2
    proof −
      assume List.product [0..<N] [0..<M] = l1 @ (i, j) # l2
      hence (i,j)∈set (List.product [0..<N] [0..<M]) by simp
      thus ?thesis by simp
    qed

    show ?thesis
      unfolding opr-fold-impl-def
      apply (refine-vcg
        nfoldli-rule[where I=λl1 - s.
            if s=2 then ∃ i<N. ∃ j<M. ¬f (m(i,j)) (n(i,j))
            else (
              (s=0 ∨ s=1) ∧
              (∀ (i,j)∈set l1. f (m(i,j)) (n(i,j))) ∧
              (s=1 ⟷ (∃ (i,j)∈set l1. g (m(i,j)) (n(i,j))))
            )]
      )
      apply (vc-solve dest: aux solve: asm-rl simp: EQ) [6]
      apply (fastforce simp: EQ)
      done
  qed

  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×_r Id → ⟨bool-rel⟩nres-rel
    apply (rule frefI)
    using opr-fold-impl-eq
    by (auto intro: nres-relI)

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 $\to_a$ *bool-assn*
begin

lemma *this-loc*: *mtx-pointwise-cmpop-gen-impl N M f g assn A get-impl fi gi*
by *unfold-locales*

context
notes [[*sepref-register-adhoc f g N M*]]
notes [*intf-of-assn*] = *intf-of-assnI*[**where** *R*=*assn* **and** *'a*=*'a i-mtx*]
notes [*sepref-import-param*] = *IdI*[*of N*] *IdI*[*of M*]
notes [*sepref-fr-rules*] = *get-impl-hnr fi-hnr gi-hnr*
begin
sepref-thm *opr-fold-impl1* **is** *uncurry opr-fold-impl* :: $assn^d *_a assn^k \to_a$ *bool-assn*
unfolding *opr-fold-impl-def*[*abs-def*] *nfoldli-nfoldli-prod-conv*[*symmetric*]
by *sepref*

end

concrete-definition (**in** −) *mtx-pointwise-cmpop-fold-impl1*
uses *mtx-pointwise-cmpop-gen-impl.opr-fold-impl1.refine-raw* **is** (*uncurry ?f*,-)∈-
prepare-code-thms (**in** −) *mtx-pointwise-cmpop-fold-impl1-def*

lemma *op-hnr*[*sepref-fr-rules*]: (*uncurry* (*mtx-pointwise-cmpop-fold-impl1 N M get-impl fi gi*), *uncurry* (*RETURN oo PR-CONST* (*mtx-pointwise-cmpop f g*))) ∈ $assn^d$ $*_a$ $assn^k$ $\to_a$ *bool-assn*
unfolding *PR-CONST-def*
apply (*rule hfref-weaken-pre'*[*OF - mtx-pointwise-cmpop-fold-impl1.refine*[*OF this-loc*,*FCOMP opr-fold-impl-refine*]])
apply (*auto dest*: *assn-range*)
done

end

end

3.14 Matrices by Array (Row-Major)

theory *IICF-Array-Matrix*
imports *../Intf/IICF-Matrix Separation-Logic-Imperative-HOL.Array-Blit*
begin

definition *is-amtx N M c mtx* ≡ ∃$_A$*l*. *mtx* ↦$_a$ *l* ∗ ↑(
length l = *N*∗*M*
∧ (∀ *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*)

apply *rule*
unfolding *is-amtx-def*
apply *clarsimp*
apply *prec-extract-eqs*
apply (*rule ext*)
apply (*rename-tac x*)
apply (*case-tac x*; *simp*)
apply (*rename-tac i j*)
apply (*case-tac i<N*; *case-tac j<M*; *simp*)
done

lemma *is-amtx-bounded*:
shows *rdomp* (*is-amtx N M*) *m* ⟹ *mtx-nonzero m* ⊆ {*0..<N*}×{*0..<M*}
unfolding *rdomp-def*
apply (*clarsimp simp*: *mtx-nonzero-def is-amtx-def*)
by (*meson not-less*)

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*) (λ*i*. *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*
by (*rule mlex-bound*)

lemma *mtx-idx-unique-conv*[*simp*]:
fixes *M* :: *nat*
assumes *j*<*M j'*<*M*
shows (*i* ∗ *M* + *j* = *i'* ∗ *M* + *j'*) ⟷ (*i*=*i'* ∧ *j*=*j'*)
using *assms*
apply *auto*
subgoal

by (*metis add-right-cancel div-if div-mult-self3 linorder-neqE-nat not-less0*)
subgoal
using ‹⟦$j < M$; $j' < M$; $i * M + j = i' * M + j'$⟧ ⟹ $i = i'$› **by** *auto*
done

lemma *mtx-tabulate-rl*[*sep-heap-rules*]:
assumes *NONZ*: *mtx-nonzero c* ⊆ {*0*..<*N*}×{*0*..<*M*}
shows <*emp*> *mtx-tabulate N M c* <*IICF-Array-Matrix.is-amtx N M c*>
proof (*cases M=0*)
case *True* **thus** *?thesis*
unfolding *mtx-tabulate-def*
using *mtx-nonzeroD*[*OF - NONZ*]
by (*sep-auto simp*: *is-amtx-def*)
next
case *False* **hence** *M-POS*: *0*<*M* **by** *auto*
show *?thesis*
unfolding *mtx-tabulate-def*
apply (*sep-auto*
decon:
imp-for′-rule[**where**
I=λ*k* (*i*,*j*,*mi*). ∃$_A$*m*. *mi* ↦$_a$ *m*
∗ ↑(*k*=*i*∗*M*+*j* ∧ *j*<*M* ∧ *k*≤*N*∗*M* ∧ *length m* = *N*∗*M*)
∗ ↑(∀*i′*<*i*. ∀*j*<*M*. *m*!(*i′*∗*M*+*j*) = *c* (*i′*,*j*))
∗ ↑(∀*j′*<*j*. *m*!(*i*∗*M*+*j′*) = *c* (*i*,*j′*))
]
simp: *nth-list-update M-POS dest*: *Suc-lessI*
)
unfolding *is-amtx-def*
using *mtx-nonzeroD*[*OF - NONZ*]
apply *sep-auto*
by (*metis add.right-neutral M-POS mtx-idx-unique-conv*)
qed

lemma *mtx-copy-rl*[*sep-heap-rules*]:
<*is-amtx N M c mtx*> *amtx-copy mtx* <λ*r*. *is-amtx N M c mtx* ∗ *is-amtx N M c r*>
by (*sep-auto simp*: *amtx-copy-def is-amtx-def*)

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*
unfolding *IS-ID-def PRES-ZERO-UNIQUE-def* **by** *auto*

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*
apply (*rule fref-ncI*) **unfolding** *op-amtx-dfltNxM-def*[*abs-def*] *mtx-rel-def*
apply *parametricity*
by (*auto simp add*: *PRES-ZERO-UNIQUE-def*)
end

lemma *mtx-dflt-rl*[*sep-heap-rules*]: <*emp*> *amtx-dflt N M k* <*is-amtx N M* (*op-amtx-dfltNxM N M k*)>
by (*sep-auto simp*: *amtx-dflt-def is-amtx-def*)

lemma *mtx-get-rl*[*sep-heap-rules*]: ⟦*i*<*N*; *j*<*M* ⟧ ⟹ <*is-amtx N M c mtx*> *mtx-get M mtx* (*i*,*j*) <λ*r*. *is-amtx N M c mtx* ∗ ↑(*r* = *c* (*i*,*j*))>
by (*sep-auto simp*: *mtx-get-def is-amtx-def*)

lemma *mtx-set-rl*[*sep-heap-rules*]: ⟦*i*<*N*; *j*<*M* ⟧
⟹ <*is-amtx N M c mtx*> *mtx-set M mtx* (*i*,*j*) *v* <λ*r*. *is-amtx N M* (*c*((*i*,*j*) := *v*)) *r*>
by (*sep-auto simp*: *mtx-set-def is-amtx-def nth-list-update*)

definition *amtx-assn N M A* ≡ *hr-comp* (*is-amtx N M*) (⟨*the-pure A*⟩*mtx-rel*)
lemmas [*fcomp-norm-unfold*] = *amtx-assn-def*[*symmetric*]
lemmas [*safe-constraint-rules*] = *CN-FALSEI*[*of is-pure amtx-assn N M A* **for** *N M A*]

lemma [*intf-of-assn*]: *intf-of-assn A TYPE*('*a*) ⟹ *intf-of-assn* (*amtx-assn N M A*) *TYPE*('*a i-mtx*)
by *simp*

abbreviation *asmtx-assn N A* ≡ *amtx-assn N N A*

lemma *mtx-rel-pres-zero*:
assumes *PRES-ZERO-UNIQUE A*
assumes (*m*,*m*′)∈⟨*A*⟩*mtx-rel*
shows *m ij* = *0* ⟷ *m*′ *ij* = *0*
using *assms*
apply1 (*clarsimp simp*: *IS-PURE-def PRES-ZERO-UNIQUE-def is-pure-conv mtx-rel-def*)
apply (*drule fun-relD*) **applyS** (*rule IdI*[*of ij*]) **applyS** *auto*
done

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*}
apply (*clarsimp simp*: *mtx-nonzero-def amtx-assn-def rdomp-hrcomp-conv*)
apply (*drule is-amtx-bounded*)
using *assms*
by (*fastforce simp*: *IS-PURE-def is-pure-conv mtx-rel-pres-zero*[*symmetric*] *mtx-nonzero-def*)

lemma *mtx-tabulate-aref*:
(*mtx-tabulate N M*, *RETURN o op-mtx-new*)
∈ [λ*c*. *mtx-nonzero c* ⊆ {*0*..<*N*}×{*0*..<*M*}]$_a$ *id-assn*^{k} → *IICF-Array-Matrix.is-amtx N M*
by *sepref-to-hoare sep-auto*

lemma *mtx-copy-aref*:
(*amtx-copy*, *RETURN o op-mtx-copy*) ∈ (*is-amtx N M*)^k →$_a$ *is-amtx N M*
apply *rule* **apply** *rule*
apply (*sep-auto simp*: *pure-def*)
done

lemma *mtx-nonzero-bid-eq*:
assumes *R*⊆*Id*
assumes (*a*, *a'*) ∈ *Id* → *R*
shows *mtx-nonzero a* = *mtx-nonzero a'*
using *assms*
apply (*clarsimp simp*: *mtx-nonzero-def*)
apply (*metis fun-relE2 pair-in-Id-conv subsetCE*)
done

lemma *mtx-nonzero-zu-eq*:
assumes *PRES-ZERO-UNIQUE R*
assumes (*a*, *a'*) ∈ *Id* → *R*
shows *mtx-nonzero a* = *mtx-nonzero a'*
using *assms*
apply (*clarsimp simp*: *mtx-nonzero-def PRES-ZERO-UNIQUE-def*)
by (*metis* (*no-types*, *opaque-lifting*) *IdI Image-singleton-iff converse-iff singletonD tagged-fun-relD-none*)

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*
by (*rule op-mtx-new.fref*)

sepref-decl-impl (*no-register*) *amtx-new-by-tab*: *mtx-tabulate-aref* **uses** *op-mtx-new-fref'*
by (*auto simp*: *mtx-nonzero-zu-eq*)

sepref-decl-impl *amtx-copy*: *mtx-copy-aref* **.**

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*)
by (*auto simp*: *mop-mtx-new-alt*[*abs-def*])

context fixes *N M* :: *nat* **begin**

sepref-register *PR-CONST* (*op-amtx-new N M*) :: (*nat* × *nat* ⇒ *'a*) ⇒ *'a i-mtx*
end

lemma *amtx-new-hnr*[*sepref-fr-rules*]:
fixes *A* :: *'a*::*zero* ⇒ *'b*::{*zero*,*heap*} ⇒ *assn*
shows *CONSTRAINT* (*IS-PURE PRES-ZERO-UNIQUE*) *A* ⟹
(*mtx-tabulate N M*, (*RETURN* ∘ *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*
using *amtx-new-by-tab-hnr*[*of A N M*] **by** *simp*

lemma [*def-pat-rules*]: *op-amtx-new*$*N*$*M* ≡ *UNPROTECT* (*op-amtx-new N M*) **by** *simp*

context fixes *N M* :: *nat* **notes** [*param*] = *IdI*[*of N*] *IdI*[*of M*] **begin**

lemma *mtx-dflt-aref*:
(*amtx-dflt N M*, *RETURN o PR-CONST* (*op-amtx-dfltNxM N M*)) ∈ *id-assn*^{k} →$_a$ *is-amtx N M*
apply *rule* **apply** *rule*
apply (*sep-auto simp*: *pure-def*)
done
sepref-decl-impl *amtx-dflt*: *mtx-dflt-aref* .

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*
apply *rule* **apply** *rule*
apply (*sep-auto simp*: *pure-def*)
done
sepref-decl-impl *amtx-get*: *amtx-get-aref* .

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*
apply *rule* **apply** (*rule hn-refine-preI*) **apply** *rule*
apply (*sep-auto simp*: *pure-def hn-ctxt-def invalid-assn-def*)
done

sepref-decl-impl *amtx-set*: *amtx-set-aref* .

lemma *amtx-get-aref'*:
(*uncurry* (*mtx-get M*), *uncurry* (*RETURN oo op-mtx-get*)) ∈ (*is-amtx N M*)^k ∗$_a$ (*prod-assn* (*pure* (*nbn-rel N*)) (*pure* (*nbn-rel M*)))^k →$_a$ *id-assn*
apply *rule* **apply** *rule*
apply (*sep-auto simp*: *pure-def IS-PURE-def IS-ID-def*)

done

sepref-decl-impl *amtx-get'*: *amtx-get-aref'* **.**

lemma *amtx-set-aref'*: (*uncurry2* (*mtx-set M*), *uncurry2* (*RETURN ooo op-mtx-set*))

$\in$ (*is-amtx N M*)^d $*_a$ (*prod-assn* (*pure* (*nbn-rel N*)) (*pure* (*nbn-rel M*)))^k $*_a$ *id-assn*^{k} $\to_a$ *is-amtx N M*
apply *rule* **apply** (*rule hn-refine-preI*) **apply** *rule*
apply (*sep-auto simp*: *pure-def hn-ctxt-def invalid-assn-def IS-PURE-def IS-ID-def*)
done

sepref-decl-impl *amtx-set'*: *amtx-set-aref'* **.**

end

3.14.1 Pointwise Operations

context
fixes *M N* :: *nat*
begin
sepref-decl-op *amtx-lin-get*: $\lambda f\ i.$ *op-mtx-get f* (*i div M*, *i mod M*) :: $\langle A\rangle$*mtx-rel* $\to$ *nat-rel* $\to$ *A*
unfolding *op-mtx-get-def mtx-rel-def*
by (*rule frefI*) (*parametricity*; *simp*)

sepref-decl-op *amtx-lin-set*: $\lambda f\ i\ x.$ *op-mtx-set f* (*i div M*, *i mod M*) *x* :: $\langle A\rangle$*mtx-rel* $\to$ *nat-rel* $\to$ *A* $\to$ $\langle A\rangle$*mtx-rel*
unfolding *op-mtx-set-def mtx-rel-def*
apply (*rule frefI*) **apply** *parametricity* **by** *simp-all*

lemma *op-amtx-lin-get-aref*: (*uncurry Array.nth*, *uncurry* (*RETURN oo PR-CONST op-amtx-lin-get*)) $\in$ [λ(-,*i*). *i*<*N*$*$*M*]$_a$ (*is-amtx N M*)^k $*_a$ *nat-assn*^{k} $\to$ *id-assn*
apply *sepref-to-hoare*
unfolding *is-amtx-def*
apply *sep-auto*
apply (*metis mult.commute div-eq-0-iff div-mult2-eq div-mult-mod-eq mod-less-divisor mult-is-0 not-less0*)
done

sepref-decl-impl *amtx-lin-get*: *op-amtx-lin-get-aref* **by** *auto*

lemma *op-amtx-lin-set-aref*: (*uncurry2* ($\lambda m\ i\ x.$ *Array.upd i x m*), *uncurry2* (*RETURN ooo PR-CONST op-amtx-lin-set*)) $\in$ [λ((-,*i*),-). *i*<*N*$*$*M*]$_a$ (*is-amtx N M*)^d $*_a$ *nat-assn*^{k} $*_a$ *id-assn*^{k} $\to$ *is-amtx N M*
proof $-$
have [*simp*]: $i < N * M \Longrightarrow \neg(M \le i \bmod M)$ **for** *i*
by (*cases* $N = 0 \vee M = 0$) (*auto simp add*: *not-le*)

have [*simp*]: $i < N * M \Longrightarrow \neg(N \leq i \mathbin{div} M)$ **for** *i*
apply (*cases* $N = 0 \vee M = 0$)
apply (*auto simp add*: *not-le*)
apply (*metis mult.commute div-eq-0-iff div-mult2-eq neq0-conv*)
done
show *?thesis*
apply *sepref-to-hoare*
unfolding *is-amtx-def*
by (*sep-auto simp*: *nth-list-update*)
qed

sepref-decl-impl *amtx-lin-set*: *op-amtx-lin-set-aref* **by** *auto*
end

lemma *amtx-fold-lin-get*: $m\ (i \mathbin{div} M,\ i \mathbin{mod} M) = op\text{-}amtx\text{-}lin\text{-}get\ M\ m\ i$ **by** *simp*
lemma *amtx-fold-lin-set*: $m\ ((i \mathbin{div} M,\ i \mathbin{mod} M) := x) = op\text{-}amtx\text{-}lin\text{-}set\ M\ m\ i\ x$ **by** *simp*

locale *amtx-pointwise-unop-impl* = *mtx-pointwise-unop-loc* +
fixes $A :: 'a \Rightarrow 'ai::\{zero,heap\} \Rightarrow assn$
fixes *fi* :: $nat \times nat \Rightarrow 'ai \Rightarrow 'ai\ Heap$
assumes *fi-hnr*:
$(uncurry\ fi, uncurry\ (RETURN\ oo\ f)) \in (prod\text{-}assn\ nat\text{-}assn\ nat\text{-}assn)^k *_a A^k \rightarrow_a A$
begin

lemma *this-loc*: *amtx-pointwise-unop-impl N M f A fi* **by** *unfold-locales*

context
assumes *PURE*: *CONSTRAINT* (*IS-PURE PRES-ZERO-UNIQUE*) *A*
begin
context
notes [[*sepref-register-adhoc f N M*]]
notes [*sepref-import-param*] = *IdI*[*of N*] *IdI*[*of M*]
notes [*sepref-fr-rules*] = *fi-hnr*
notes [*safe-constraint-rules*] = *PURE*
notes [*simp*] = *algebra-simps*
begin
sepref-thm *opr-fold-impl1* **is** *RETURN o opr-fold-impl* :: $(amtx\text{-}assn\ N\ M\ A)^d \rightarrow_a amtx\text{-}assn\ N\ M\ A$
unfolding *opr-fold-impl-def fold-prod-divmod-conv'*
apply (*rewrite amtx-fold-lin-set*)
apply (*rewrite* **in** *f* - ⌑ *amtx-fold-lin-get*)
by *sepref*
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*
unfolding *PR-CONST-def*
apply (*rule hfref-weaken-pre'*[*OF - amtx-pointwise-unnop-fold-impl1.refine*[*OF this-loc PURE*,*FCOMP opr-fold-impl-refine*]])
by (*simp add*: *amtx-assn-bounded*[*OF PURE*])
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*
by *unfold-locales*

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*
unfolding *opr-fold-impl-def*[*abs-def*] *fold-prod-divmod-conv'*
apply (*rewrite amtx-fold-lin-set*)
apply (*rewrite* **in** *f* ⌑ - *amtx-fold-lin-get*)
apply (*rewrite* **in** *f* - ⌑ *amtx-fold-lin-get*)
by *sepref*

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*

$M\ A)^k \rightarrow_a$ *amtx-assn N M A*
unfolding *PR-CONST-def*
apply (*rule hfref-weaken-pre'*[*OF - amtx-pointwise-binop-fold-impl1.refine*[*OF this-loc PURE,FCOMP opr-fold-impl-refine*]])
apply (*auto dest*: *amtx-assn-bounded*[*OF PURE*])
done

end

locale *amtx-pointwise-cmpop-impl* = *mtx-pointwise-cmpop-loc* +
fixes *A* :: *'a* ⇒ *'ai*::{*zero,heap*} ⇒ *assn*
fixes *fi* :: *'ai* ⇒ *'ai* ⇒ *bool Heap*
fixes *gi* :: *'ai* ⇒ *'ai* ⇒ *bool Heap*
assumes *fi-hnr*:
(*uncurry fi,uncurry* (*RETURN oo f*)) ∈ $A^k *_a A^k \rightarrow_a$ *bool-assn*
assumes *gi-hnr*:
(*uncurry gi,uncurry* (*RETURN oo g*)) ∈ $A^k *_a A^k \rightarrow_a$ *bool-assn*
begin

lemma *this-loc*: *amtx-pointwise-cmpop-impl f g A fi gi*
by *unfold-locales*

context
notes [[*sepref-register-adhoc f g N M*]]
notes [*sepref-import-param*] = *IdI*[*of N*] *IdI*[*of M*]
notes [*sepref-fr-rules*] = *fi-hnr gi-hnr*
assumes *PURE*[*safe-constraint-rules*]: *CONSTRAINT* (*IS-PURE PRES-ZERO-UNIQUE*) *A*
begin
sepref-thm *opr-fold-impl1* **is** *uncurry opr-fold-impl* :: (*amtx-assn N M A*)$^d *_a$(*amtx-assn N M A*)$^k \rightarrow_a$ *bool-assn*
unfolding *opr-fold-impl-def*[*abs-def*] *nfoldli-prod-divmod-conv*
apply (*rewrite* **in** *f* ⌑ - *amtx-fold-lin-get*)
apply (*rewrite* **in** *f* - ⌑ *amtx-fold-lin-get*)
apply (*rewrite* **in** *g* ⌑ - *amtx-fold-lin-get*)
apply (*rewrite* **in** *g* - ⌑ *amtx-fold-lin-get*)
by *sepref*
end

concrete-definition (**in** −) *amtx-pointwise-cmpop-fold-impl1* **for** *N M fi gi*
uses *amtx-pointwise-cmpop-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*)$^k \rightarrow_a$ *bool-assn*
unfolding *PR-CONST-def*

apply (*rule hfref-weaken-pre'*[*OF - amtx-pointwise-cmpop-fold-impl1.refine*[*OF this-loc PURE*,*FCOMP opr-fold-impl-refine*]])
apply (*auto dest*: *amtx-assn-bounded*[*OF PURE*])
done

end

3.14.2 Regression Test and Usage Example

context begin

To work with a matrix, the dimension should be fixed in a context

context
fixes *N M* :: *nat*
— We also register the dimension as an operation, such that we can use it like a constant
notes [[*sepref-register-adhoc N M*]]
notes [*sepref-import-param*] = *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*) (λ*i m*.
nfoldli [*0*..<*M*] (λ-. *True*) (λ*j m*. *do* {
let mij = *m*(*i*,*j*);
RETURN (*m*((*i*,*j*) := *x* ∗ *mij*))
}
) *m*
) *m*

After declaration of an implementation for multiplication, refinement is straightforward. Note that we use the fixed *N* in the refinement assertions.

private lemma *times-param*: ((∗),(∗)::*'a*⇒-) ∈ *Id* → *Id* → *Id* **by** *simp*

context
notes [*sepref-import-param*] = *times-param*
begin
qualified 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*)
unfolding *scmul-def*[*abs-def*]
by *sepref*
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*
unfolding *init-test-def*
by *sepref*

Initialization from function diagonal is more complicated: First, we have to define the function as a new constant

qualified definition *diagonalN k* ≡ λ(*i*,*j*). *if i=j* ∧ *j<N then k else 0*

If it carries implicit parameters, we have to wrap it into a *PR-CONST* tag:

private sepref-register *PR-CONST diagonalN*
private lemma [*def-pat-rules*]: *IICF-Array-Matrix.diagonalN$N* ≡ *UNPROTECT diagonalN* **by** *simp*

Then, we have to implement the constant, where the result assertion must be for a pure function. Note that, due to technical reasons, we need the *the-pure* in the function type, and the refinement rule to be parameterized over an assertion variable (here *A*). Of course, you can constrain *A* further, e.g., *CONSTRAINT* (*IS-PURE IS-ID*) *A*

private lemma *diagonalN-hnr*[*sepref-fr-rules*]:
assumes *CONSTRAINT* (*IS-PURE PRES-ZERO-UNIQUE*) *A*

shows (*return o diagonalN*, *RETURN o* (*PR-CONST diagonalN*)) ∈ *A*^{k} →$_a$ *pure* (*nat-rel* ×$_r$ *nat-rel* → *the-pure A*)
using *assms*
apply *sepref-to-hoare*
apply (*sep-auto simp*: *diagonalN-def is-pure-conv IS-PURE-def PRES-ZERO-UNIQUE-def*)
done

In order to discharge preconditions, we need to prove some auxiliary lemma that non-zero indexes are within range

lemma *diagonal-nonzero-ltN*[*simp*]: (*a*,*b*)∈*mtx-nonzero* (*diagonalN k*) ⟹ *a<N* ∧ *b<N*
by (*auto simp*: *mtx-nonzero-def diagonalN-def split*: *if-split-asm*)

private definition *init-test2* ≡ *do* {
ASSERT (*N>2*); — Ensure that the coordinate (*1*,*2*) is valid
let m = op-mtx-new (*diagonalN* (*1*::*int*));
RETURN (*m*(*1*,*2*))
}
private sepref-definition *init-test2-impl* **is** *uncurry0 init-test2* :: *unit-assn*^{k}→$_a$*int-assn*
unfolding *init-test2-def amtx-fold-custom-new*[*of N N*]
by *sepref*

end

export-code *IICF-Array-Matrix.scmul-impl* **in** *SML-imp*

end

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* :: $('k \rightharpoonup 'v) \Rightarrow 'm \Rightarrow assn$
fixes *contains-key* :: $'k \Rightarrow 'm \Rightarrow bool\ Heap$
assumes *contains-key-rule*[*sep-heap-rules*]:
$<$*is-map m p*$>$ *contains-key k p* $<\lambda r.$ *is-map m p* $* \uparrow(r \longleftrightarrow k \in dom\ m)>_t$

locale *gen-contains-key-by-lookup* = *imp-map-lookup*
begin
definition *contains-key k m* $\equiv$ *do* $\{r \leftarrow$ *lookup k m*; *return* $(\neg$*is-None r*$)\}$

```
  sublocale imp-map-contains-key is-map contains-key
    apply unfold-locales
    unfolding contains-key-def
    apply (sep-auto split: option.splits)
    done

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
  by unfold-locales (sep-auto simp: os-head-def neq-Nil-conv)

interpretation os: imp-list-tail os-list os-tl
  by unfold-locales (sep-auto simp: os-tl-def neq-Nil-conv)



definition cs-is-empty :: 'a::heap cs-list ⇒ bool Heap where
  cs-is-empty p ≡ return (is-None p)
interpretation cs: imp-list-is-empty cs-list cs-is-empty
  by unfold-locales (sep-auto simp: cs-is-empty-def split: option.splits)

definition cs-head :: 'a::heap cs-list ⇒ 'a Heap where
  cs-head p ≡ case p of
    None ⇒ raise STR ''cs-head: Empty list''
  | Some p ⇒ do { n ← !p; return (val n)}
interpretation cs: imp-list-head cs-list cs-head
  by unfold-locales (sep-auto simp: neq-Nil-conv cs-head-def)

definition cs-tail :: 'a::heap cs-list ⇒ 'a cs-list Heap where
  cs-tail p ≡ do { (-,r) ← cs-pop p; return r }
interpretation cs: imp-list-tail cs-list cs-tail
```

by *unfold-locales* (*sep-auto simp*: *cs-tail-def*)

lemma *is-hashmap-finite*[*simp*]: $h \models$ *is-hashmap m mi* $\Longrightarrow$ *finite* (*dom m*)
unfolding *is-hashmap-def is-hashmap'-def*
by *auto*

lemma *is-hashset-finite*[*simp*]: $h \models$ *is-hashset s si* $\Longrightarrow$ *finite s*
unfolding *is-hashset-def*
by (*auto dest*: *is-hashmap-finite*)

definition *ias-is-it s a si* $\equiv \lambda(a',i).$
$\exists_A l.\ a \mapsto_a l * \uparrow(a'=a \wedge s=$*ias-of-list l* $\wedge$ ($i=$*length l* $\wedge$ *si*$=\{\} \vee i<$*length l* $\wedge\ i \in s \wedge$ *si*$=s \cap \{x.\ x \geq i\}$))

context begin
private function *first-memb* **where**
first-memb lmax a i = *do* {
if i<*lmax then do* {
x ← *Array.nth a i*;
if x then return i else first-memb lmax a (*Suc i*)
} *else*
return i
}
by *pat-completeness auto*
termination by (*relation measure* ($\lambda(l,-,i).\ l-i$)) *auto*
declare *first-memb.simps*[*simp del*]

private lemma *first-memb-rl-aux*:
assumes *lmax* $\leq$ *length l i*$\leq$*lmax*
shows
$< a \mapsto_a l >$
first-memb lmax a i
$<\lambda k.\ a \mapsto_a l * \uparrow(k \leq lmax \wedge (\forall j.\ i \leq j \wedge j < k \longrightarrow \neg l!j) \wedge i \leq k \wedge (k=lmax \vee l!k))>$
using *assms*
proof (*induction lmax a i rule*: *first-memb.induct*)
case (*1 lmax a i*)
show *?case*
apply (*subst first-memb.simps*)
using *1.prems*
apply (*sep-auto heap*: *1.IH*; ((*sep-auto*;*fail*) | *metis eq-iff not-less-eq-eq*))
done
qed

private lemma *first-memb-rl*[*sep-heap-rules*]:
assumes *lmax* ≤ *length l i*≤*lmax*
shows < *a* ↦$_a$ *l* >
first-memb lmax a i
<λ*k*. *a*↦$_a$ *l* ∗ ↑(*ias-of-list l* ∩ {*i*..<*k*} = {} ∧ *i*≤*k* ∧ (*k*<*lmax* ∧ *k*∈*ias-of-list l* ∨ *k*=*lmax*)) >
using *assms*
by (*sep-auto simp*: *ias-of-list-def heap*: *first-memb-rl-aux*)

definition *ias-it-init a* = *do* {
l ← *Array.len a*;
i ← *first-memb l a 0*;
return (*a*,*i*)
}

definition *ias-it-has-next* ≡ λ(*a*,*i*). *do* {
l ← *Array.len a*;
return (*i*<*l*)
}

definition *ias-it-next* ≡ λ(*a*,*i*). *do* {
l ← *Array.len a*;
i' ← *first-memb l a* (*Suc i*);
return (*i*,(*a*,*i'*))
}

lemma *ias-of-list-bound*: *ias-of-list l* ⊆ {*0*..<*length l*} **by** (*auto simp*: *ias-of-list-def*)

end

interpretation *ias*: *imp-set-iterate is-ias ias-is-it ias-it-init ias-it-has-next ias-it-next*
apply *unfold-locales*
unfolding *is-ias-def ias-is-it-def*
unfolding *ias-it-init-def* **using** *ias-of-list-bound*
apply (*sep-auto*)
unfolding *ias-it-next-def* **using** *ias-of-list-bound*
apply (*sep-auto*; *fastforce*)
unfolding *ias-it-has-next-def*
apply *sep-auto*
apply *sep-auto*
done

lemma *ias-of-list-finite*[*simp*, *intro*!]: *finite* (*ias-of-list l*)
using *finite-subset*[*OF ias-of-list-bound*] **by** *auto*

lemma *is-ias-finite*[*simp*]: *h* ⊨ *is-ias S x* ⟹ *finite S*
unfolding *is-ias-def* **by** *auto*

lemma *to-list-ga-rec-rule*:
assumes *imp-set-iterate is-set is-it it-init it-has-next it-next*
assumes *imp-list-prepend is-list l-prepend*
assumes *FIN*: *finite it*
assumes *DIS*: *distinct l set l* $\cap$ *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
< $\lambda r.\ \exists_A l'.$ *is-set s si*
$*$ *is-list l' r*
$*$ $\uparrow$(*distinct l'* $\wedge$ *set l'* = *set l* $\cup$ *it*) $>_t$
proof −
interpret *imp-set-iterate is-set is-it it-init it-has-next it-next*
+ *imp-list-prepend is-list l-prepend*
by *fact*+

from *FIN DIS* **show** *?thesis*
proof (*induction arbitrary*: *l li iti rule*: *finite-psubset-induct*)
case (*psubset it*)
show *?case*
apply (*subst to-list-ga-rec.simps*)
using *psubset.prems* **apply** (*sep-auto heap*: *psubset.IH*)
apply (*rule ent-frame-fwd*[*OF quit-iteration*])
apply *frame-inference*
apply *solve-entails*
done
qed
qed
lemma *to-list-ga-rule*:
assumes *IT*: *imp-set-iterate is-set is-it it-init it-has-next it-next*
assumes *EM*: *imp-list-empty is-list l-empty*
assumes *PREP*: *imp-list-prepend is-list l-prepend*
assumes *FIN*: *finite s*
shows
<*is-set s si*>
to-list-ga it-init it-has-next it-next
l-empty l-prepend si
<$\lambda r.\ \exists_A l.$ *is-set s si* $*$ *is-list l r* $*$ *true* $*$ $\uparrow$(*distinct l* $\wedge$ *set l* = *s*)>
proof −
interpret *imp-list-empty is-list l-empty* +
imp-set-iterate is-set is-it it-init it-has-next it-next
by *fact*+

note [*sep-heap-rules*] = *to-list-ga-rec-rule*[*OF IT PREP*]
show *?thesis*
unfolding *to-list-ga-def*
by (*sep-auto simp*: *FIN*)

qed

3.15.1 Binding Locales

method *solve-sepl-binding* = (
unfold-locales;
(*unfold option-assn-pure-conv*)*?*;
sep-auto
intro!: *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*
by *solve-sepl-binding*

sepref-decl-impl (*no-register*) *empty*: *empty-hnr-aux* .
end

locale *bind-map-is-empty* = *imp-map-is-empty* + *bind-map*
begin
lemma *is-empty-hnr-aux*: (*is-empty*,*RETURN o op-map-is-empty*) ∈ *is-map*^{k}
→$_a$ *bool-assn*
by *solve-sepl-binding*

sepref-decl-impl *is-empty*: *is-empty-hnr-aux* .
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*
by *solve-sepl-binding*

sepref-decl-impl *update*: *update-hnr-aux* .
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} $\rightarrow_a$ *is-map*
by *solve-sepl-binding*

sepref-decl-impl *delete*: *delete-hnr-aux* .
end

locale *bind-map-lookup* = *imp-map-lookup* + *bind-map*
begin
lemma *lookup-hnr-aux*: (*uncurry lookup*,*uncurry* (*RETURN oo op-map-lookup*)) ∈ *id-assn*^{k} $*_a$ *is-map*^{k} $\rightarrow_a$ *id-assn*
by *solve-sepl-binding*

sepref-decl-impl *lookup*: *lookup-hnr-aux* .
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} $\rightarrow_a$ *bool-assn*
by *solve-sepl-binding*

sepref-decl-impl *contains-key*: *contains-key-hnr-aux* .
end

Set

locale *bind-set* = *imp-set is-set* **for** *is-set* :: (′*ai set*) ⇒ ′*m* ⇒ *assn* +
fixes *A* :: ′*a* ⇒ ′*ai* ⇒ *assn*
begin
definition *assn* ≡ *hr-comp is-set* (⟨*the-pure A*⟩*set-rel*)
lemmas [*safe-constraint-rules*] = *CN-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* .
lemma *APAlu*: ⟦*PROP Q*; *CONSTRAINT* (*IS-PURE IS-LEFT-UNIQUE*) *A*⟧ ⟹ *PROP Q* .
lemma *APAru*: ⟦*PROP Q*; *CONSTRAINT* (*IS-PURE IS-RIGHT-UNIQUE*) *A*⟧ ⟹ *PROP Q* .
lemma *APAbu*: ⟦*PROP Q*; *CONSTRAINT* (*IS-PURE IS-LEFT-UNIQUE*) *A*;

CONSTRAINT (*IS-PURE IS-RIGHT-UNIQUE*) *A*⟧ ⟹ *PROP Q* .

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*
by *solve-sepl-binding*

interpretation *bind-set-setup* **by** *standard*

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*
by *solve-sepl-binding*

interpretation *bind-set-setup* **by** *standard*
lemmas *hnr-op-is-empty*[*sepref-fr-rules*] = *hnr-is-empty-aux*[*THEN APA*,*FCOMP op-set-is-empty.fref*[**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} →$_a$ *bool-assn*
by *solve-sepl-binding*

interpretation *bind-set-setup* **by** *standard*
lemmas *hnr-op-member*[*sepref-fr-rules*] = *hnr-member-aux*[*THEN APAbu*,*FCOMP op-set-member.fref*[**where** *A*=*the-pure A*]]
lemmas *hnr-mop-member*[*sepref-fr-rules*] = *hnr-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} →$_a$ *is-set*
by *solve-sepl-binding*

interpretation *bind-set-setup* **by** *standard*
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*
by *solve-sepl-binding*

interpretation *bind-set-setup* **by** *standard*
lemmas *hnr-op-delete*[*sepref-fr-rules*] = *hnr-delete-aux*[*THEN APAbu*,*FCOMP op-set-delete.fref*[**where** *A*=*the-pure A*]]
lemmas *hnr-mop-delete*[*sepref-fr-rules*] = *hnr-op-delete*[*FCOMP mk-mop-rl2-np*[*OF mop-set-delete-alt*]]
end

locale *bind-set-size* = *imp-set-size* + *bind-set*
begin
lemma *hnr-size-aux*: (*size*, (*RETURN o op-set-size*))∈ *is-set*^{k} $\rightarrow_a$ *nat-assn*
by *solve-sepl-binding*

interpretation *bind-set-setup* **by** *standard*
lemmas *hnr-op-size*[*sepref-fr-rules*] = *hnr-size-aux*[*THEN APAbu*,*FCOMP op-set-size.fref*[**where** *A*=*the-pure A*]]
lemmas *hnr-mop-size*[*sepref-fr-rules*] = *hnr-op-size*[*FCOMP mk-mop-rl1-np*[*OF mop-set-size-alt*]]
end

primrec *sorted-wrt'* **where**
sorted-wrt' *R* [] ⟷ *True*
| *sorted-wrt'* *R* (*x*#*xs*) ⟷ *list-all* (*R x*) *xs* ∧ *sorted-wrt'* *R xs*

lemma *sorted-wrt'-eq*: *sorted-wrt'* = *sorted-wrt*
proof (*intro ext iffI*)
fix *R* :: *'a* ⇒ *'a* ⇒ *bool* **and** *xs* :: *'a list*
{
assume *sorted-wrt R xs*
thus *sorted-wrt' R xs*
by (*induction xs*)(*auto simp*: *list-all-iff*)
}
{
assume *sorted-wrt' R xs*
thus *sorted-wrt R xs*
by (*induction xs*) (*auto simp*: *list-all-iff*)

```
  }
qed

lemma param-sorted-wrt[param]: (sorted-wrt, sorted-wrt) ∈ (A → A → bool-rel)
→ ⟨A⟩list-rel → bool-rel
  unfolding sorted-wrt'-eq[symmetric] sorted-wrt'-def
  by parametricity

lemma obtain-list-from-setrel:
  assumes SV: single-valued A
  assumes (set l,s) ∈ ⟨A⟩set-rel
  obtains m where s=set m (l,m)∈⟨A⟩list-rel
  using assms(2)
proof (induction l arbitrary: s thesis)
  case Nil
  show ?case
    apply (rule Nil(1)[where m=[]])
    using Nil(2)
    by auto
next
  case (Cons x l)
  obtain s' y where s=insert y s' (x,y)∈A (set l,s')∈⟨A⟩set-rel
  proof −
    from Cons.prems(2) obtain y where X0: y∈s (x,y)∈A
      unfolding set-rel-def by auto
    from Cons.prems(2) have
      X1: ∀ a∈set l. ∃ b∈s. (a,b)∈A and
      X2: ∀ b∈s. ∃ a∈insert x (set l). (a,b)∈A
      unfolding set-rel-def by auto
    show ?thesis proof (cases ∃ a∈set l. (a,y)∈A)
      case True
      show ?thesis
        apply (rule that[of y s])
        subgoal using X0 by auto
        subgoal by fact
        subgoal
          apply (rule set-relI)
          subgoal using X1 by blast
         subgoal by (metis IS-RIGHT-UNIQUED SV True X0(2) X2 insert-iff)

          done
        done
    next
      case False
      show ?thesis
        apply (rule that[of y s−{y}])
        subgoal using X0 by auto
        subgoal by fact
        subgoal
```

```
        apply (rule set-relI)
        subgoal using False X1 by fastforce
        subgoal using IS-RIGHT-UNIQUED SV X0(2) X2 by fastforce
        done
      done
    qed
  qed
  moreover from Cons.IH[OF - ‹(set l,s')∈⟨A⟩set-rel›] obtain m where s'=set
m (l,m)∈⟨A⟩list-rel .
  ultimately show thesis
    apply −
    apply (rule Cons.prems(1)[of y#m])
    by auto
 qed

 lemma param-it-to-sorted-list[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
  unfolding it-to-sorted-list-def[abs-def]
  apply (auto simp: it-to-sorted-list-def pw-nres-rel-iff refine-pw-simps)
  apply (rule obtain-list-from-setrel; assumption?; clarsimp)
  apply (intro exI conjI; assumption?)
  using param-distinct[param-fo] apply blast
  apply simp
  using param-sorted-wrt[param-fo] apply blast
  done



 locale bind-set-iterate = imp-set-iterate + bind-set +
  assumes is-set-finite: h ⊨ is-set S x ⟹ finite S
 begin
  context begin
   private lemma is-imp-set-iterate: imp-set-iterate is-set is-it it-init it-has-next
it-next by unfold-locales

    private lemma is-imp-list-empty: imp-list-empty (list-assn id-assn) (return
[])
      apply unfold-locales
      apply solve-constraint
      apply sep-auto
      done

   private lemma is-imp-list-prepend: imp-list-prepend (list-assn id-assn) (return
oo List.Cons)
      apply unfold-locales
      apply solve-constraint
      apply (sep-auto simp: pure-def)
      done
```

definition *to-list* ≡ *to-list-ga it-init it-has-next it-next* (*return* []) (*return oo List.Cons*)
private lemmas *tl-rl* = *to-list-ga-rule*[*OF is-imp-set-iterate is-imp-list-empty is-imp-list-prepend*, *folded to-list-def*]

private lemma *to-list-sorted1*: (*to-list*,*PR-CONST* (*it-to-sorted-list* (λ- -. *True*))) ∈ *is-set*^{k} →$_a$ *list-assn id-assn*
unfolding *PR-CONST-def*
apply (*intro hfrefI*)
apply (*rule hn-refine-preI*)
apply (*rule hn-refineI*)
unfolding *it-to-sorted-list-def*
apply (*sep-auto intro*: *hfrefI hn-refineI intro*: *is-set-finite heap*: *tl-rl*)
done

private lemma *to-list-sorted2*: ⟦
CONSTRAINT (*IS-PURE IS-LEFT-UNIQUE*) *A*;
CONSTRAINT (*IS-PURE IS-RIGHT-UNIQUE*) *A*⟧ ⟹
(*PR-CONST* (*it-to-sorted-list* (λ- -. *True*)), *PR-CONST* (*it-to-sorted-list* (λ- -. *True*))) ∈ ⟨*the-pure A*⟩*set-rel* → ⟨⟨*the-pure A*⟩*list-rel*⟩*nres-rel*
unfolding *PR-CONST-def CONSTRAINT-def IS-PURE-def*
by *clarify parametricity*

lemmas *to-list-hnr* = *to-list-sorted1*[*FCOMP to-list-sorted2*, *folded assn-def*]

lemmas *to-list-is-to-sorted-list* = *IS-TO-SORTED-LISTI*[*OF to-list-hnr*]
lemma *to-list-gen*[*sepref-gen-algo-rules*]: ⟦*CONSTRAINT* (*IS-PURE IS-LEFT-UNIQUE*) *A*; *CONSTRAINT* (*IS-PURE IS-RIGHT-UNIQUE*) *A*⟧
⟹ *GEN-ALGO to-list* (*IS-TO-SORTED-LIST* (λ- -. *True*) (*bind-set.assn is-set A*) *A*)
by (*simp add*: *GEN-ALGO-def to-list-is-to-sorted-list*)

end
end

locale *bind-set-union* = *imp-set-union* + *bind-set* +
assumes *is-prime-set-finite*: *h* ⊨ *is-set S x* ⟹ *finite S*
begin
lemma *hnr-union-aux*: (*uncurry union*, *uncurry* (*RETURN oo op-set-union*))
∈ *is-set*^{d} **$_a$ *is-set*^{k} →$_a$ *is-set*
apply (*sep-auto intro*!: *is-prime-set-finite*)
unfolding *invalid-assn-def pure-def pure-assn-def hfref-def*
apply (*solve-sepl-binding*) **unfolding** *entails-def*
using *is-prime-set-finite* **subgoal**
using *mod-starD* **by** *blast*
apply *sep-auto*
using *is-prime-set-finite mod-starD*
unfolding *hoare-triple-def ex-assn-def* **apply** *sep-auto*

using *mod-starD union-rule*
by (*metis assn-times-comm mult-1 pure-assn-def pure-true*)

interpretation *bind-set-setup* **by** *standard*
lemmas *hnr-op-union*[*sepref-fr-rules*] = *hnr-union-aux*[*FCOMP op-set-union.fref*[**where** *A*=*the-pure A*]]
lemmas *hnr-mop-union*[*sepref-fr-rules*] = *hnr-op-union*[*FCOMP mk-mop-rl2-np*[*OF mop-set-union-alt*]]

end

List

locale *bind-list* = *imp-list is-list* **for** *is-list* :: (*'ai list*) ⇒ *'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*
apply *rule* **apply** *rule* **apply** (*sep-auto simp*: *pure-def*) **done**

lemmas *hnr*
= *hnr-aux*[*FCOMP op-list-empty.fref*[*of the-pure A*], *folded assn-def*]

lemmas *hnr-mop* = *hnr*[*FCOMP mk-mop-rl0-np*[*OF mop-list-empty-alt*]]
end

locale *bind-list-is-empty* = *imp-list-is-empty* + *bind-list*
begin
lemma *hnr-aux*: (*is-empty*,*RETURN o op-list-is-empty*)∈(*is-list*)^k →$_a$ *pure bool-rel*
apply *rule* **apply** *rule* **apply** (*sep-auto simp*: *pure-def*) **done**

lemmas *hnr*[*sepref-fr-rules*]
= *hnr-aux*[*FCOMP op-list-is-empty.fref*, *of the-pure A*, *folded assn-def*]
lemmas *hnr-mop*[*sepref-fr-rules*] = *hnr*[*FCOMP mk-mop-rl1-np*[*OF mop-list-is-empty-alt*]]
end

locale *bind-list-append* = *imp-list-append* + *bind-list*
begin
lemma *hnr-aux*: (*uncurry* (*swap-args2 append*),*uncurry* (*RETURN oo op-list-append*))

$\in(\textit{is-list})^d *_a (\textit{pure Id})^k \to_a \textit{is-list}$ **by** *solve-sepl-binding*

lemmas *hnr*[*sepref-fr-rules*]
= *hnr-aux*[*FCOMP op-list-append.fref*,*of A*, *folded assn-def*]
lemmas *hnr-mop*[*sepref-fr-rules*] = *hnr*[*FCOMP mk-mop-rl2-np*[*OF mop-list-append-alt*]]
end

locale *bind-list-prepend* = *imp-list-prepend* + *bind-list*
begin
lemma *hnr-aux*: (*uncurry prepend*,*uncurry* (*RETURN oo op-list-prepend*))
$\in(\textit{pure Id})^k *_a (\textit{is-list})^d \to_a \textit{is-list}$ **by** *solve-sepl-binding*

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*)
$\in[\lambda l.\ l\neq[]]_a\ (\textit{is-list})^d \to \textit{pure Id}$ **by** *solve-sepl-binding*

lemmas *hnr*[*sepref-fr-rules*] = *hnr-aux*[*FCOMP op-list-hd.fref*,*of A*, *folded assn-def*]
lemmas *hnr-mop*[*sepref-fr-rules*] = *hnr*[*FCOMP mk-mop-rl1*[*OF mop-list-hd-alt*]]
end

locale *bind-list-tl* = *imp-list-tail* + *bind-list*
begin
lemma *hnr-aux*: (*tail*,*RETURN o op-list-tl*)
$\in[\lambda l.\ l\neq[]]_a\ (\textit{is-list})^d \to \textit{is-list}$
by *solve-sepl-binding*

lemmas *hnr*[*sepref-fr-rules*] = *hnr-aux*[*FCOMP op-list-tl.fref*,*of the-pure A*, *folded assn-def*]
lemmas *hnr-mop*[*sepref-fr-rules*] = *hnr*[*FCOMP mk-mop-rl1*[*OF mop-list-tl-alt*]]
end

locale *bind-list-rotate1* = *imp-list-rotate* + *bind-list*
begin
lemma *hnr-aux*: (*rotate*,*RETURN o op-list-rotate1*)
$\in(\textit{is-list})^d \to_a \textit{is-list}$
by *solve-sepl-binding*

lemmas *hnr*[*sepref-fr-rules*] = *hnr-aux*[*FCOMP op-list-rotate1.fref*,*of the-pure A*, *folded assn-def*]
lemmas *hnr-mop*[*sepref-fr-rules*] = *hnr*[*FCOMP mk-mop-rl1-np*[*OF mop-list-rotate1-alt*]]
end

locale *bind-list-rev* = *imp-list-reverse* + *bind-list*
begin
lemma *hnr-aux*: (*reverse*,*RETURN o op-list-rev*)
∈(*is-list*)^d →$_a$ *is-list*
by *solve-sepl-binding*

lemmas *hnr*[*sepref-fr-rules*] = *hnr-aux*[*FCOMP op-list-rev.fref*,*of the-pure A*, *folded assn-def*]
lemmas *hnr-mop*[*sepref-fr-rules*] = *hnr*[*FCOMP mk-mop-rl1-np*[*OF mop-list-rev-alt*]]
end

3.15.2 Array Map (iam)

definition *op-iam-empty* ≡ *IICF-Map.op-map-empty*
interpretation *iam*: *bind-map-empty is-iam iam-new*
by *unfold-locales*

interpretation *iam*: *map-custom-empty op-iam-empty*
by *unfold-locales* (*simp add*: *op-iam-empty-def*)
lemmas [*sepref-fr-rules*] = *iam.empty-hnr*[*folded op-iam-empty-def*]

definition [*simp*]: *op-iam-empty-sz* (*N*::*nat*) ≡ *IICF-Map.op-map-empty*
lemma [*def-pat-rules*]: *op-iam-empty-sz*$*N* ≡ *UNPROTECT* (*op-iam-empty-sz N*)
by *simp*

interpretation *iam-sz*: *map-custom-empty PR-CONST* (*op-iam-empty-sz N*)
apply *unfold-locales*
apply (*simp*)
done
lemma [*sepref-fr-rules*]: (*uncurry0 iam-new*, *uncurry0* (*RETURN* (*PR-CONST* (*op-iam-empty-sz N*)))) ∈ *unit-assn*^{k} →$_a$ *iam.assn K V*
using *iam.empty-hnr*[*of K V*] **by** *simp*

interpretation *iam*: *bind-map-update is-iam Array-Map-Impl.iam-update*
by *unfold-locales*

interpretation *iam*: *bind-map-delete is-iam Array-Map-Impl.iam-delete*
by *unfold-locales*

interpretation *iam*: *bind-map-lookup is-iam Array-Map-Impl.iam-lookup*
by *unfold-locales*

setup *Locale-Code.open-block*
interpretation *iam*: *gen-contains-key-by-lookup is-iam Array-Map-Impl.iam-lookup*
by *unfold-locales*
setup *Locale-Code.close-block*

interpretation *iam*: *bind-map-contains-key is-iam iam.contains-key*
by *unfold-locales*

3.15.3 Array Set (ias)

definition [*simp*]: *op-ias-empty* ≡ *op-set-empty*
interpretation *ias*: *bind-set-empty is-ias ias-new* **for** *A*
by *unfold-locales*

interpretation *ias*: *set-custom-empty ias-new op-ias-empty*
by *unfold-locales simp*
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*)
by *simp*

interpretation *ias-sz*: *bind-set-empty is-ias ias-new-sz N* **for** *N A*
by *unfold-locales*

interpretation *ias-sz*: *set-custom-empty ias-new-sz N PR-CONST* (*op-ias-empty-sz N*) **for** *A*
by *unfold-locales simp*
lemma [*sepref-fr-rules*]:
(*uncurry0* (*ias-new-sz N*), *uncurry0* (*RETURN* (*PR-CONST* (*op-ias-empty-sz N*)))) ∈ *unit-assn*^{k} →$_a$ *ias.assn A*
using *ias-sz.hnr-op-empty*[*of N A*] **by** *simp*

interpretation *ias*: *bind-set-member is-ias Array-Set-Impl.ias-memb* **for** *A*
by *unfold-locales*

interpretation *ias*: *bind-set-insert is-ias Array-Set-Impl.ias-ins* **for** *A*
by *unfold-locales*

interpretation *ias*: *bind-set-delete is-ias Array-Set-Impl.ias-delete* **for** *A*
by *unfold-locales*

setup *Locale-Code.open-block*
interpretation *ias*: *bind-set-iterate is-ias ias-is-it ias-it-init ias-it-has-next ias-it-next* **for** *A*
by *unfold-locales auto*
setup *Locale-Code.close-block*

3.15.4 Hash Map (hm)

interpretation *hm*: *bind-map-empty is-hashmap hm-new*
by *unfold-locales*

definition *op-hm-empty* ≡ *IICF-Map.op-map-empty*
interpretation *hm*: *map-custom-empty op-hm-empty*
  by *unfold-locales* (*simp add*: *op-hm-empty-def*)
lemmas [*sepref-fr-rules*] = *hm.empty-hnr*[*folded op-hm-empty-def*]

interpretation *hm*: *bind-map-is-empty is-hashmap Hash-Map.hm-isEmpty*
  by *unfold-locales*

interpretation *hm*: *bind-map-update is-hashmap Hash-Map.hm-update*
  by *unfold-locales*

interpretation *hm*: *bind-map-delete is-hashmap Hash-Map.hm-delete*
  by *unfold-locales*

interpretation *hm*: *bind-map-lookup is-hashmap Hash-Map.hm-lookup*
  by *unfold-locales*

setup *Locale-Code.open-block*
interpretation *hm*: *gen-contains-key-by-lookup is-hashmap Hash-Map.hm-lookup*
  by *unfold-locales*
setup *Locale-Code.close-block*

interpretation *hm*: *bind-map-contains-key is-hashmap hm.contains-key*
  by *unfold-locales*

3.15.5 Hash Set (hs)

interpretation *hs*: *bind-set-empty is-hashset hs-new* **for** *A*
  by *unfold-locales*

definition *op-hs-empty* ≡ *IICF-Set.op-set-empty*
interpretation *hs*: *set-custom-empty hs-new op-hs-empty* **for** *A*
  by *unfold-locales* (*simp add*: *op-hs-empty-def*)
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*
  by *unfold-locales*

interpretation *hs*: *bind-set-member is-hashset Hash-Set-Impl.hs-memb* **for** *A*
  by *unfold-locales*

interpretation *hs*: *bind-set-insert is-hashset Hash-Set-Impl.hs-ins* **for** *A*
  by *unfold-locales*

interpretation *hs*: *bind-set-delete is-hashset Hash-Set-Impl.hs-delete* **for** *A*
  by *unfold-locales*

interpretation *hs*: *bind-set-size is-hashset hs-size* **for** *A*
  by *unfold-locales*

setup *Locale-Code.open-block*
interpretation *hs*: *bind-set-iterate is-hashset hs-is-it hs-it-init hs-it-has-next hs-it-next* **for** *A*
by *unfold-locales simp*
setup *Locale-Code.close-block*

interpretation *hs*: *bind-set-union is-hashset hs-is-it hs-it-init hs-it-has-next hs-it-next hs-union*
for *A*
by *unfold-locales simp*

3.15.6 Open Singly Linked List (osll)

interpretation *osll*: *bind-list os-list* **for** *A* **by** *unfold-locales*
interpretation *osll-empty*: *bind-list-empty os-list os-empty* **for** *A*
by *unfold-locales*

definition *osll-empty* ≡ *op-list-empty*
interpretation *osll*: *list-custom-empty osll.assn A os-empty osll-empty*
apply *unfold-locales*
apply (*rule osll-empty.hnr*)
by (*simp add*: *osll-empty-def*)

interpretation *osll-is-empty*: *bind-list-is-empty os-list os-is-empty* **for** *A*
by *unfold-locales*

interpretation *osll-prepend*: *bind-list-prepend os-list os-prepend* **for** *A*
by *unfold-locales*

interpretation *osll-hd*: *bind-list-hd os-list os-head* **for** *A*
by *unfold-locales*

interpretation *osll-tl*: *bind-list-tl os-list os-tl* **for** *A*
by *unfold-locales*

interpretation *osll-rev*: *bind-list-rev os-list os-reverse* **for** *A*
by *unfold-locales*

3.15.7 Circular Singly Linked List (csll)

interpretation *csll*: *bind-list cs-list* **for** *A* **by** *unfold-locales*

interpretation *csll-empty*: *bind-list-empty cs-list cs-empty* **for** *A*
by *unfold-locales*

definition *csll-empty* ≡ *op-list-empty*
interpretation *csll*: *list-custom-empty csll.assn A cs-empty csll-empty*
apply *unfold-locales*
apply (*rule csll-empty.hnr*)

by (*simp add*: *csll-empty-def*)

interpretation *csll-is-empty*: *bind-list-is-empty cs-list cs-is-empty* **for** *A*
by *unfold-locales*

interpretation *csll-prepend*: *bind-list-prepend cs-list cs-prepend* **for** *A*
by *unfold-locales*

interpretation *csll-append*: *bind-list-append cs-list cs-append* **for** *A*
by *unfold-locales*

interpretation *csll-hd*: *bind-list-hd cs-list cs-head* **for** *A*
by *unfold-locales*

interpretation *csll-tl*: *bind-list-tl cs-list cs-tail* **for** *A*
by *unfold-locales*

interpretation *csll-rotate1*: *bind-list-rotate1 cs-list cs-rotate* **for** *A*
by *unfold-locales*

schematic-goal *hn-refine* (*emp*) (*?c*::*?'c Heap*) *?Γ' ?R* (*do* {
x ← *mop-list-empty*;
RETURN (*1* ∈ *dom* [*1*::*nat* ↦ *True*, *2*↦*False*], {*1*,*2*::*nat*}, *1*#(*2*::*nat*)#*x*)
})
apply (*subst iam-sz.fold-custom-empty*[**where** *N*=*10*])
apply (*subst hs.fold-custom-empty*)
apply (*subst osll.fold-custom-empty*)
by *sepref*

end

3.16 The Imperative Isabelle Collection Framework

theory *IICF*
imports

Intf/IICF-Set
Impl/IICF-List-SetO

Intf/IICF-Multiset
Intf/IICF-Prio-Bag

Impl/IICF-List-Mset
Impl/IICF-List-MsetO

Impl/Heaps/IICF-Impl-Heap

Intf/IICF-Map
Intf/IICF-Prio-Map

Impl/Heaps/IICF-Impl-Heapmap

Intf/IICF-List

Impl/IICF-Array
Impl/IICF-HOL-List
Impl/IICF-Array-List
Impl/IICF-Indexed-Array-List
Impl/IICF-MS-Array-List

Intf/IICF-Matrix

Impl/IICF-Array-Matrix

Impl/IICF-Sepl-Binding

begin
thy-deps
end

Chapter 4

User Guides

This chapter contains the available user guides.

4.1 Quickstart Guide

theory *Sepref-Guide-Quickstart*
imports *../IICF/IICF*
begin

4.1.1 Introduction

Sepref is an Isabelle/HOL tool to semi-automatically synthesize imperative code from abstract specifications.

The synthesis works by replacing operations on abstract data by operations on concrete data, leaving the structure of the program (mostly) unchanged. 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:

apply (*clarsimp intro*!: *nres-relI*)

The - *nres* type defines the refinement ordering, which is a lifted subset ordering, with *FAIL* being the greatest element. This means, that we can assume a non-empty list during the refinement proof (otherwise, the RHS will be *FAIL*, and the statement becomes trivial)

The Isabelle Refinement Framework provides various techniques to extract verification conditions from given goals, we use the standard VCG here:

unfolding *min-of-list-def min-of-list1-def*
apply (*refine-vcg*)

The VCG leaves us with a standard HOL goal, which is easily provable

by (*auto simp*: *neq-Nil-conv Min.set-eq-fold*[*symmetric*])

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*
unfolding *min-of-list-def*[*abs-def*] *min-of-list1-def*[*abs-def*]
apply (*refine-vcg*)
by (*auto simp*: *neq-Nil-conv Min.set-eq-fold*[*symmetric*])

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*
unfolding *min-of-list2-def*[*abs-def*] *min-of-list1-def*[*abs-def*]
apply *refine-vcg*
apply *clarsimp-all*
apply (*rewrite* **in** -=¤ *fold-idx-conv*)
by (*auto simp*: *nth-tl hd-conv-nth*)

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*
unfolding *min-of-list2-def*[*abs-def*]
by *sepref*

The generated constant represents an Imperative/HOL program, and is executable:

thm *min-of-list3-def*
export-code *min-of-list3* **checking** *SML-imp*

Also note that the Sepref tool applied a deforestation optimization: It recognizes a fold over $[0..<n]$, and implements it by the tail-recursive function *imp-for'*, which uses a counter instead of an intermediate list.

There are a couple of optimizations, which come in the form of two sets of simplifier rules, which are applied one after the other:

thm *sepref-opt-simps*
thm *sepref-opt-simps2*

They are just named theorem collections, e.g., *sepref-opt-simps add*/*del* can be used to modify them.

Moreover, a refinement theorem is generated, which states the correspondence between *min-of-list3* and *min-of-list2*:

thm *min-of-list3.refine*

It states the relations between the parameter and the result of the concrete and abstract function. The parameter is related by *array-assn id-assn*. Here, *array-assn A* relates arrays with lists, such that the elements are related A — in our case by *nat-assn*, which relates natural numbers to themselves. We also say that we *implement* lists of nats by arrays of nats. The result is also implemented by natural numbers.

Moreover, the parameters may be stored on the heap, and we have to indicate whether the function keeps them intact or not. Here, we use the annotation -^k (for *keep*) to indicate that the parameter is kept intact, and -^d (for *destroy*) to indicate that it is destroyed.

Overall Correctness Statement

Finally, we can use transitivity of refinement to link our implementation to the specification. The *FCOMP* attribute is able to compose refinement theorems:

theorem *min-of-list3-correct*: (*min-of-list3*,*min-of-list*) ∈ (*array-assn nat-assn*)^k $\rightarrow_a$ *nat-assn*
using *min-of-list3.refine*[*FCOMP min-of-list2-refine*, *FCOMP min-of-list1-refine*]
.

While the above statement is suited to re-use the algorithm within the sepref-framework, a more low-level correctness theorem can be stated using separation logic. This has the advantage that understanding the statement depends on less definitional overhead:

lemma $l{\neq}[] \Longrightarrow <\mathit{array\text{-}assn\ nat\text{-}assn\ l\ a}> \mathit{min\text{-}of\text{-}list3\ a} <\lambda x.\ \mathit{array\text{-}assn\ nat\text{-}assn\ l\ a} * \uparrow(\forall\, y{\in}\mathit{set\ l}.\ x{\leq}y)>_t$

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:

by (*sep-auto*
  simp: *min-of-list-def pure-def pw-le-iff refine-pw-simps*
  heap: *min-of-list3-correct*[*THEN hfrefD*, *of l a*, *THEN hn-refineD*, *simplified*])

If the automatic method does not work, here is a more explicit proof, that can be adapted for proving similar statements:

lemma $l{\neq}[] \Longrightarrow <\mathit{array\text{-}assn\ nat\text{-}assn\ l\ a}> \mathit{min\text{-}of\text{-}list3\ a} <\lambda x.\ \mathit{array\text{-}assn\ nat\text{-}assn\ l\ a} * \uparrow(\forall\, y{\in}\mathit{set\ l}.\ x{\leq}y)>_t$
proof −

We inlined the definition of *min-of-list*. This will yield two proof obligations later, which we discharge as auxiliary lemmas here

assume [*simp*]: $l{\neq}[]$
have [*simp*]: *nofail* (*min-of-list l*)
  by (*auto simp*: *min-of-list-def refine-pw-simps*)
have *1*: $\bigwedge x.\ \mathit{RETURN}\ x \leq \mathit{min\text{-}of\text{-}list}\ l \Longrightarrow \forall\, y{\in}\mathit{set\ l}.\ x{\leq}y$
  by (*auto simp*: *min-of-list-def pw-le-iff refine-pw-simps*)

note *rl* = *min-of-list3-correct*[*THEN hfrefD*, *of l a*, *THEN hn-refineD*, *simplified*]

This should yield a Hoare-triple for *min-of-list3 a*, which can now be used to prove the desired statement via a consequence rule

show *?thesis*
  apply (*rule cons-rule*[*OF - - rl*])

The preconditions should match, however, *sep-auto* is also able to discharge more complicated implications here. Be sure to simplify with *pure-def*, if you have parameters that are not stored on the heap (in our case, we don't, but include the simplification anyway.)

  apply (*sep-auto simp*: *pure-def*)

The heap-parts of the postcondition should also match. The pure parts require the auxiliary statements that we proved above.

  apply (*sep-auto simp*: *pure-def dest*!: *1*)
  done
qed

Using the Algorithm

As an example, we now want to use our algorithm to compute the minimum value of some concrete list. In order to use an algorithm, we have to declare both, 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* **by** *auto*

Here, we use a feature of Sepref to import parametricity theorems. Note that the parametricity theorem we provide here is trivial, as *nat-rel* is identity, and *list-rel* as well as (→) preserve identity. However, we have to specify a parametricity theorem that reflects the structure of the involved types.

Finally, we can invoke Sepref

sepref-definition *min-of-rand-list1* **is** *min-of-rand-list* :: $nat\text{-}assn^k \rightarrow_a nat\text{-}assn$
unfolding *min-of-rand-list-def*[*abs-def*]

We construct a plain list, however, the implementation of *min-of-list* expects an array. We have to insert a conversion, which is conveniently done with the *rewrite* method:

apply (*rewrite* **in** *min-of-list* ¤ *array-fold-custom-of-list*)
by *sepref*

In the generated code, we see that the pure *rand-list* function is invoked, its result is converted to an array, which is then passed to *min-of-list3*.

Note that **sepref-definition** prints the generated theorems to the output on the end of the proof. Use the output panel, or hover the mouse over the by-command to see this output.

The generated algorithm can be exported

export-code *min-of-rand-list1* **checking** *SML OCaml? Haskell? Scala*

and executed

ML-val ‹@{*code min-of-rand-list1*} (@{*code nat-of-integer*} *100*) ()›

Note that Imperative/HOL for ML generates a function from unit, and applying this function triggers execution.

4.1.3 Binary Search Example

As second example, we consider a simple binary search algorithm. We specify the abstract problem, i.e., finding an element in a sorted list.

definition *in-sorted-list x xs* ≡ *ASSERT* (*sorted xs*) ≫ *RETURN* (*x*∈*set xs*)

And give a standard iterative implementation:

```
definition in-sorted-list1-invar x xs ≡ λ(l,u,found).
    (l≤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 $(?\Phi \Longrightarrow ?M \leq ?M') \Longrightarrow ?M \leq \mathit{ASSERT}\ ?\Phi \gg\!= (\lambda\text{-}.\ ?M')$).

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)))
by simp

private lemma neq-nlt-is-gt:
  fixes a b :: 'a::linorder
  shows a≠b ⟹ ¬(a<b) ⟹ a>b by simp

private lemma isl1-aux1:
  assumes sorted xs
  assumes i<length xs
  assumes xs!i < x
  shows x∉set (take i xs)
  using assms
  by (auto simp: take-set leD sorted-nth-mono)

private lemma isl1-aux2:
  assumes x ∉ set (take n xs)
  shows x∉set (drop n xs) ⟷ x∉set xs
  apply (rewrite in - = ¤ append-take-drop-id[of n,symmetric])
  using assms
  by (auto simp del: append-take-drop-id)

lemma in-sorted-list1-refine: (in-sorted-list1, in-sorted-list)∈Id → Id → ⟨Id⟩nres-rel
  unfolding in-sorted-list1-def[abs-def] in-sorted-list-def[abs-def]
  apply (refine-vcg isl1-measure)
  apply (vc-solve simp: in-sorted-list1-invar-def isl1-aux1 isl1-aux2 solve: asm-rl)
  apply (auto simp: take-set set-drop-conv leD sorted-nth-mono) []
  apply (auto simp: take-set leD sorted-nth-mono dest: neq-nlt-is-gt) []
  done
```

end

First, let's synthesize an implementation where the list elements are natural numbers. We will discuss later how to generalize the implementation for arbitrary types.

For technical reasons, the Sepref tool works with uncurried functions. That is, every function has exactly one argument. You can use the *uncurry* function, and we also provide abbreviations *uncurry2* up to $\lambda f.$ *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 $\rightarrow_a$ *bool-assn*
  unfolding *in-sorted-list1-def*[*abs-def*]
  by *sepref*

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'* :: $nat\text{-}assn^k *_a (array\text{-}assn\ nat\text{-}assn)^k \rightarrow_a bool\text{-}assn$
unfolding *in-sorted-list1'-def*[*abs-def*]

— If *sepref* fails, you can use *sepref-dbg-keep* to get some more information.
apply *sepref-dbg-keep*
— This prints a trace of the different phases of sepref, and stops when the first phase fails. It then returns the internal proof state of the tool, which can be inspected further.

Here, the translation phase fails. The translation phase translates the control structures and operations of the abstract program to their concrete counterparts. To inspect the actual problem, we let translation run until the operation where it fails:

supply [[*goals-limit=1*]] — There will be many subgoals during translation, and printing them takes very long with Isabelle :(
apply *sepref-dbg-trans-keep*
— Things get stuck at a goal with predicate *hn-refine*. This is the internal refinement predicate, *hn-refine* $\Gamma\ c\ \Gamma'\ R\ a$ means, that, for operands whose refinement is described by Γ, the concrete program c refines the abstract program a, such that, afterwards, the operands are described by Γ', and the results are refined by R.

Inspecting the first subgoal reveals that we got stuck on refining the abstract operation *RETURN* \$ (*op-list-get* \$ *b* \$ *xf*). Note that the (\$) is just a constant for function application, which is used to tame Isabelle's higher-order unification algorithms. You may use *unfolding APP-def*, or even *simp* to get a clearer picture of the failed goal.

If a translation step fails, it may be helpful to execute as much of the translation step as possible:

apply *sepref-dbg-trans-step-keep*
— The translation step gets stuck at proving *pre-list-get* (*b*, *xf*), which is the precondition for list indexing.
apply (*sepref-dbg-side-keep*) — If you think the side-condition should be provable, this command returns the left-over subgoals after some preprocessing and applying auto

oops

Internals of Sepref

Internally, *sepref* consists of multiple phases that are executed one after the other. Each phase comes with its own debugging method, which only executes that phase. We illustrate this by repeating the refinement of *min-of-list2*. This time, we use **sepref-thm**, which only generates a refinement theorem, but defines no constants:

sepref-thm *min-of-list3'* **is** *min-of-list2* :: $(array\text{-}assn\ nat\text{-}assn)^k \rightarrow_a nat\text{-}assn$

— The *sepref-thm* or *sepref-definition* command assembles a schematic goal statement.

unfolding *min-of-list2-def*[*abs-def*]

— The preprocessing phase converts the goal into the *hn-refine*-form. Moreover, it adds interface type annotations for the parameters. (for now, the interface type is just the HOL type of the parameter, in our case, *nat list*)

apply *sepref-dbg-preproc*

— The next phase applies a consequence rule for the postcondition and result. This is mainly for technical reasons.

apply *sepref-dbg-cons-init*

— The next phase tries to identify the abstract operations, and inserts tag-constants for function application and abstraction. These tags are for technical reasons, working around Isabelle/HOL's unifier idiosyncrasies.

Operation identification assigns a single constant to each abstract operation, which is required for technical reasons. Note that there are terms in HOL, which qualify as a single operation, but consists of multiple constants, for example, $\{x\}$, which is just syntactic sugar for *insert* x $\{\}$. In our case, the operation identification phase rewrites the assertion operations followed by a bind to a single operation *op-ASSERT-bind*, and renames some operations to more canonical names.

apply *sepref-dbg-id*

— Now that it is clear which operations to execute, we have to specify an execution order. Note that HOL has no notion of execution at all. However, if we want to translate to operations that depend on a heap, we need a notion of execution order. We use the *nres*-monad's bind operation as sequencing operator, and flatten all nested operations, using left-to-right evaluation order.

apply *sepref-dbg-monadify*

— The next step just prepares the optimization phase, which will be executed on the translated program. It just applies the rule $[\![$*hn-refine ?Γ ?c ?Γ′ ?R ?a*; *CNV ?c ?c*$]\!] \Longrightarrow$ *hn-refine ?Γ ?c′ ?Γ′ ?R ?a*.

apply *sepref-dbg-opt-init*

— The translation phase does the main job of translating the abstract program to the concrete one. It has rules how to translate abstract operations to concrete ones. For technical reasons, it differentiates between operations, which have only first-order arguments (e.g., *length*) and combinators, which have also higher-order arguments (e.g., *fold*).

The basic idea of translation is to repeatedly apply the translation rule for the top-most combinator/operator, and thus recursively translate the whole program. The rules may produce various types of side-conditions, which are resolved by the tool.

apply *sepref-dbg-trans*

— The next phase applies some simplification rules to optimize the translated program. It essentially simplifies first with the rules *sepref-opt-simps*, and then with *sepref-opt-simps2*.

apply *sepref-dbg-opt*

— The next two phases resolve the consequence rules introduced by the *cons-init* phase.

apply *sepref-dbg-cons-solve*

apply *sepref-dbg-cons-solve*

— The translation phase and the consequence rule solvers may postpone some side conditions on yet-unknown refinement assertions. These are solved in the last phase.

apply *sepref-dbg-constraints*
done

In the next sections, we will explain, by example, how to troubleshoot the various phases of the tool. We will focus on the phases that are most likely to fail.

Initialization

A common mistake is to forget the keep/destroy markers for the refinement assertion, or specify a refinement assertion with a non-matching type. This results in a type-error on the command

sepref-thm *test-add-2* **is** $\lambda x.\ RETURN\ (2{+}x) :: nat\text{-}assn^k \rightarrow_a nat\text{-}assn$
by *sepref*

Translation Phase

In most cases, the translation phase will fail. Let's try the following refinement:

sepref-thm *test* **is** $\lambda l.\ RETURN\ (l!1 + 2) :: (array\text{-}assn\ nat\text{-}assn)^k \rightarrow_a nat\text{-}assn$

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.

apply *sepref-dbg-keep*
— In the trace, we see that the translation phase failed. We are presented the tool's internal goal state just before translation. If a phase fails, the usual procedure is to start the phase in debug mode, and see how far it gets. The debug mode of the translation phase stops at the first operation or combinator it cannot translate. Note, it is a good idea to limit the visible goals, as printing goals in Isabelle can be very, very slow :(

supply [[*goals-limit = 1*]]
apply *sepref-dbg-trans-keep*
— Here, we see that translation gets stuck at *op-list-get*. This may have two reasons: Either there is no rule for this operation, or a side condition cannot be resolved. We apply a single translation step in debug mode, i.e., the translation step is applied as far as possible, leaving unsolved side conditions:

apply *sepref-dbg-trans-step-keep*
— This method reports that the "Apply rule" phase produced a wrong number of subgoals. This phase is expected to solve the goal, but left some unsolved side condition, which we are presented in the goal state. We can either guess what *pre-list-get* means and why it cannot be solved, or try to partially solve the side condition:

apply *sepref-dbg-side-keep*
— From the remaining subgoal, one can guess that there might be a problem with too short lists, where index *1* does not exist.

oops

Inserting an assertion into the abstract program solves the problem:

sepref-thm *test* **is** $\lambda l.\ \mathit{ASSERT}\ (\mathit{length}\ l > 1) \gg \mathit{RETURN}\ (l!1 + 2) :: (\mathit{array\text{-}assn}\ \mathit{nat\text{-}assn})^k \to_a \mathit{nat\text{-}assn}$
by *sepref*

Here is an example for an unimplemented operation:

sepref-thm *test* **is** $\lambda l.\ \mathit{RETURN}\ (\mathit{Min}\ (\mathit{set}\ l)) :: (\mathit{array\text{-}assn}\ \mathit{nat\text{-}assn})^k \to_a \mathit{nat\text{-}assn}$
supply [[*goals-limit* = *1*]]
apply *sepref-dbg-keep*
apply *sepref-dbg-trans-keep*
— Translation stops at the *set* operation
apply *sepref-dbg-trans-step-keep*
— This tactic reports that the "Apply rule" phase failed, which means that there is no applicable rule for the *set* operation on arrays.
oops

4.1.5 The Isabelle Imperative Collection Framework (IICF)

The IICF provides a library of imperative data structures, and some 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 (λit (m,i). dom m = s−it ∧ ran m = {0..<i} ∧ i=card (s
− it))
    s (λx (m,i). RETURN (m(x↦i),i+1)) (Map.empty,0);
  RETURN m
}
```

lemma *nat-seg-map1-refine*: (*nat-seg-map1*, *nat-seg-map*) ∈ *Id* → ⟨*Id*⟩*nres-rel*
  apply (*intro fun-relI*)
  unfolding *nat-seg-map1-def*[*abs-def*] *nat-seg-map-def*[*abs-def*]
  apply (*refine-vcg*)
  apply (*vc-solve simp*: *it-step-insert-iff solve*: *asm-rl dest*: *domD*)
  done

We use hashsets *hs.assn* and hashmaps (*hm.assn*).

sepref-definition *nat-seg-map2* **is** *nat-seg-map1* :: (*hs.assn id-assn*)^k →$_a$ *hm.assn id-assn nat-assn*
  unfolding *nat-seg-map1-def*[*abs-def*]
  apply *sepref-dbg-keep*
  apply *sepref-dbg-trans-keep*
  — We got stuck at *op-map-empty*. This is because Sepref is very conservative when it comes to guessing implementations. Actually, no constructor operation will be assigned a default operation, with some obvious exceptions for numbers and Booleans.
  oops

Assignment of implementations to constructor operations is done by 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*
  unfolding *nat-seg-map1-def*[*abs-def*]
  — We can use the *rewrite* method for position-precise rewriting:
  apply (*rewrite* **in** *FOREACHi - - -* ⌑ *hm.fold-custom-empty*)
  by *sepref*

export-code *nat-seg-map2* **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 (*intro fun-relI*; *simp*)
  fix *l* :: *'a list*
  { fix *n m*
    assume *n*≤*m* **and** *length l* = *m*
    hence *fold* (λ*i l*. *l*[*i*:=*1*+*l*!*i*]) [*n*..<*m*] *l* = *take n l* @ *map* (((+))*1*) (*drop n l*)
      apply (*induction arbitrary*: *l rule*: *inc-induct*)
      apply *simp*
      apply (*clarsimp simp*: *upt-conv-Cons take-Suc-conv-app-nth*)
      apply (*auto simp add*: *list-eq-iff-nth-eq nth-Cons split*: *nat.split*)
      done
  }
  from *this*[*of 0 length l*] **show** *incr-list1 l* = *incr-list l*
    unfolding *incr-list-def incr-list1-def*
    by *simp*
qed

Trying to refine this reveals a problem:

sepref-thm *incr-list2* **is** *RETURN o incr-list1* :: (*array-assn nat-assn*)^d →$_a$ *array-assn nat-assn*
  unfolding *incr-list1-def*[*abs-def*]
  apply *sepref-dbg-keep*
  apply *sepref-dbg-trans-keep*
  apply *sepref-dbg-trans-step-keep*
  apply *sepref-dbg-side-keep*
  — We get stuck at the precondition of *op-list-get*. Indeed, we cannot prove the generated precondition, as the translation process dropped any information from which we could conclude that the index is in range.
  oops

Of course, the fold loop has the invariant that the length of the list does not change, and thus, indexing is in range. We only cannot prove it during the automatic synthesis.

Here, the only solution is to do a manual refinement into the nres-monad, and adding an assertion that indexing is always in range.

We use the *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 (*intro nres-relI fun-relI*; *simp*)
  fix *l* :: *'a list*
  show *incr-list2 l* ≤ *RETURN* (*incr-list l*)
    unfolding *incr-list2-def incr-list-def*
    — *nfoldli* comes with an invariant proof rule. In order to use it, we have to specify the invariant manually:
   apply (*refine-vcg nfoldli-rule*[**where** *I*=λ*l1 l2 s*. *s* = *map* (((+))*1*) (*take* (*length l1*) *l*) @ *drop* (*length l1*) *l*])
    apply (*vc-solve*
      simp: *upt-eq-append-conv upt-eq-Cons-conv*
     simp: *nth-append list-update-append upd-conv-take-nth-drop take-Suc-conv-app-nth*
      solve: *asm-rl*
)
    done
qed

sepref-definition *incr-list3* **is** *incr-list2* :: (*array-assn nat-assn*)^d →$_a$ *array-assn nat-assn*
  unfolding *incr-list2-def*[*abs-def*]
  by *sepref*

lemmas *incr-list3-correct* = *incr-list3.refine*[*FCOMP incr-list2-refine*]

4.1.7 Linearity and Copying

Consider the following implementation of an operation to swap to list indexes. While it is perfectly valid in a functional setting, an imperative implementation has a problem here: Once the update a index *i* is done, the old value cannot be read from index *i* any more. We try to implement the list with an array:

sepref-thm *swap-nonlinear* **is** *uncurry2* (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*
  supply [[*goals-limit* = *1*]]
  apply *sepref-dbg-keep*
  apply *sepref-dbg-trans-keep* — (1) We get stuck at an *op-list-get* operation
  apply *sepref-dbg-trans-step-keep* — (2) Further inspection reveals that the "recover pure" phase fails, and we are left with a subgoal of the form *CON-*

STRAINT is-pure (*array-assn id-assn*). Constraint side conditions are deferrable side conditions: They are produced as side-conditions, and if they cannot be solved immediately, they are deferred and processed later, latest at the end of the synthesis. However, definitely unsolvable constraints are not deferred, but halt the translation phase immediately, and this is what happened here: At (1) we can see that the refinement for the array we want to access is *hn-invalid* (*array-assn id-assn*). This means, the data structure was destroyed by some preceding operation. The *hn-invalid* only keeps a record of this fact. When translating an operation that uses an invalidated parameter, the tool tries to restore the invalidated parameter: This only works if the data structure was purely functional, i.e., not stored on the heap. This is where the *is-pure* constraint comes from. However, arrays are always stored on the heap, so this constraint is definitely unsolvable, and thus immediately rejected instead of being deferred.

Note: There are scenarios where a constraint gets deferred *before* it becomes definitely unsolvable. In these cases, you only see the problem after the translation phase, and it may be somewhat tricky to figure out the reason.

oops

The fix for our swap function is quite obvious. Using a temporary storage for the intermediate value, we write:

sepref-thm *swap-with-tmp* **is** *uncurry2* ($\lambda l\ i\ j.$ *do* {
  ASSERT (i<*length* $l \wedge j$<*length* l);
  let tmp = $l!i$;
  RETURN ($l[i:=l!j, j:=tmp]$)
}) :: $(\textit{array-assn id-assn})^d *_a \textit{nat-assn}^k *_a \textit{nat-assn}^k \rightarrow_a \textit{array-assn id-assn}$
  by *sepref*

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* ($\lambda l\ i\ j.$ *do* {
  ASSERT (i<*length* $l \wedge j$<*length* l);
  let tmp = $l!i$;
  RETURN ($l[i:=l!j, j:=tmp]$)
}) :: $(\textit{array-assn id-assn})^k *_a \textit{nat-assn}^k *_a \textit{nat-assn}^k \rightarrow_a \textit{array-assn id-assn}$
  apply *sepref-dbg-keep* — We get stuck at a frame, which would require restoring an invalidated array
  apply *sepref-dbg-cons-solve-keep* — Which would only work if arrays were pure
  oops

If copying is really required, you have to insert it manually. Reconsider the example *incr-list* from above. This time, we want to preserve the original data (note the $()^k$ annotation):

sepref-thm *incr-list3-preserve* **is** *incr-list2* :: $(\textit{array-assn nat-assn})^k \rightarrow_a \textit{array-assn nat-assn}$
  unfolding *incr-list2-def*[*abs-def*]
  — We explicitly insert a copy-operation on the list, before it is passed to the fold operation

apply (*rewrite* **in** *nfoldli - - - ⌑ op-list-copy-def*[*symmetric*])
by *sepref*

4.1.8 Nesting of Data Structures

Sepref and the IICF support nesting of data structures with some limitations:

- Only the container or its elements can be visible at the same time. For example, if you have a product of two arrays, you can either see the two arrays, or the product. An operation like *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*
by *sepref*

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*
apply *sepref-dbg-keep*
apply *sepref-dbg-trans-keep*
apply *sepref-dbg-trans-step-keep*
oops

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*
by *sepref-dbg-keep*

Trying to create a list of arrays, first attempt:

sepref-thm *set-of-arrays-ex* **is** *uncurry0* (*RETURN* (*op-list-append* [] *op-array-empty*))
:: $unit\text{-}assn^k \rightarrow_a$ *arl-assn* (*array-assn nat-assn*)
unfolding *arl.fold-custom-empty*
apply *sepref-dbg-keep*
apply *sepref-dbg-trans-keep*
apply *sepref-dbg-trans-step-keep*
supply [[*goals-limit* = *1*, *unify-trace-failure*]]

— Many IICF data structures, in particular the array based ones, requires the element types to be of *default*. If this is not the case, Sepref will simply find no refinement for the operations. Be aware that type-class related mistakes are hard to debug in Isabelle/HOL, above we sketched how to apply the refinement rule that is supposed to match with unifier tracing switched on. The *to-hnr* attribute is required to convert the rule from the relational form to the internal *hn-refine* form. Note that some rules are already in *hn-refine* form, and need not be converted, e.g., *hn-refine* (*hn-ctxt ?P1.0 ?x1.0 ?x1'* ∗ *hn-ctxt ?P2.0 ?x2.0 ?x2'*) (*return* (*?x1'*, *?x2'*)) (*hn-invalid ?P1.0 ?x1.0 ?x1'* ∗ *hn-invalid ?P2.0 ?x2.0 ?x2'*) (*?P1.0* $\times_a$ *?P2.0*) (*RETURN* $ (*Pair* $ *?x1.0* $ *?x2.0*)).
oops

So lets choose a circular singly linked list (csll), which does not require its elements to be of default type class

sepref-thm *set-of-arrays-ex* **is** *uncurry0* (*RETURN* (*op-list-append* [] *op-array-empty*))
:: $unit\text{-}assn^k \rightarrow_a$ *csll.assn* (*array-assn nat-assn*)
unfolding *csll.fold-custom-empty*
apply *sepref-dbg-keep*
apply *sepref-dbg-trans-keep*
apply *sepref-dbg-trans-step-keep*
— We end up with an unprovable purity-constraint: As many IICF types, csll only supports pure member types. We expect this restriction to be lifted in some future version.
oops

Finally, there are a few data structures that already support nested element types, for example, functional lists:

sepref-thm *set-of-arrays-ex* **is** *uncurry0* (*RETURN* (*op-list-append* [] *op-array-empty*))
:: $unit\text{-}assn^k \rightarrow_a$ *list-assn* (*array-assn nat-assn*)
unfolding *HOL-list.fold-custom-empty*
by *sepref*

4.1.9 Fixed-Size Data Structures

For many algorithms, the required size of a data structure is already known, such that it is not necessary to use data structures with dynamic resizing.

The Sepref-tool supports such data structures, however, with some limitations.

Running Example

Assume we want to read a sequence of natural numbers in the range $\{0..<N\}$, 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* :: $(\textit{list-assn nat-assn})^k \to_a \textit{arl-assn nat-assn}$
  unfolding *remdup-def*[*abs-def*]
  — Lets use a bit-vector for the set
  apply (*rewrite* **in** *nfoldli* - - - ⌑ *ias.fold-custom-empty*)
  — And an array-list for the list
  apply (*rewrite* **in** *nfoldli* - - - ⌑ *arl.fold-custom-empty*)
  by *sepref*

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* :: $(\textit{list-assn nat-assn})^k \to_a \textit{arl-assn nat-assn}$
  unfolding *remdup-def*[*abs-def*]
  — Many of the dynamic array-based data structures in the IICF can be pre-initialized to a certain size. THis initialization is only a hint, and has no abstract consequences. The list data structure will still be resized if it grows larger than the initialization size.
  apply (*rewrite* **in** *nfoldli* - - - ⌑ *ias-sz.fold-custom-empty*[*of N*])
  apply (*rewrite* **in** *nfoldli* - - - ⌑ *arl-sz.fold-custom-empty*[*of N*])
  by *sepref*

end

To get a usable function, we may add the fixed N as a parameter, effectively converting the initialization hint to a parameter, which, however, has no abstract meaning

definition *remdup-initsz* (N::*nat*) ≡ *remdup*
lemma *remdup-init-hnr*:
($uncurry\ remdup1\text{-}initsz$, $uncurry\ remdup\text{-}initsz$) ∈ $nat\text{-}assn^k *_a (list\text{-}assn\ nat\text{-}assn)^k$ $\rightarrow_a$ *arl-assn nat-assn*
using *remdup1-initsz.refine* **unfolding** *remdup-initsz-def*[*abs-def*]
unfolding *hfref-def hn-refine-def*
by (*auto simp*: *pure-def*)

Static Data Structures

We use a locale to hide local declarations. Note: This locale will never be interpreted, otherwise all the local setup, that does not make sense outside the locale, would become visible. TODO: This is probably some abuse of locales to emulate complex private setup, including declaration of constants and lemmas.

locale *my-remdup-impl-loc* =
fixes N :: *nat*
assumes $N>0$ — This assumption is not necessary, but used to illustrate the general case, where the locale may have such assumptions
begin

For locale hierarchies, the following seems not to be available directly in Isabelle, however, it is useful when transferring stuff between the global theory and the locale

lemma *my-remdup-impl-loc-this*: *my-remdup-impl-loc N* **by** *unfold-locales*

Note that this will often require to use N as a usual constant, which is refined. For pure refinements, we can use the *sepref-import-param* attribute, which will convert a parametricity theorem to a rule for Sepref:

sepref-register *N*
lemma *N-hnr*[*sepref-import-param*]: (N,N)∈*nat-rel* **by** *simp*
thm *N-hnr*

Alternatively, we could directly prove the following rule, which, however, is more cumbersome:

lemma *N-hnr′*: ($uncurry0\ (return\ N)$, $uncurry0\ (RETURN\ N)$)∈$unit\text{-}assn^k \rightarrow_a$ *nat-assn*
by *sepref-to-hoare sep-auto*

Next, we use an array-list with a fixed maximum capacity. Note that the capacity is part of the refinement assertion now.

sepref-definition *remdup1-fixed* **is** *remdup* :: $(\textit{list-assn nat-assn})^k \rightarrow_a \textit{marl-assn N nat-assn}$
unfolding *remdup-def*[*abs-def*]
apply (*rewrite* **in** *nfoldli* - - - ⌑ *ias-sz.fold-custom-empty*[*of N*])
apply (*rewrite* **in** *nfoldli* - - - ⌑ *marl-fold-custom-empty-sz*[*of N*])
supply [[*goals-limit* = *1*]]
apply *sepref-dbg-keep*
apply *sepref-dbg-trans-keep*
apply *sepref-dbg-trans-step-keep*
— In order to append to the array list, we have to show that the size is not yet exceeded. This may require to add some assertions on the abstract level. We already have added some assertions in the definition of *remdup*.
oops

Moreover, we add a precondition on the list

sepref-definition *remdup1-fixed* **is** *remdup* :: $[\lambda l.\ \textit{set}\ l \subseteq \{0..<N\}]_a$ $(\textit{list-assn nat-assn})^k \rightarrow \textit{marl-assn N nat-assn}$
unfolding *remdup-def*[*abs-def*]
apply (*rewrite* **in** *nfoldli* - - - ⌑ *ias-sz.fold-custom-empty*[*of N*])
apply (*rewrite* **in** *nfoldli* - - - ⌑ *marl-fold-custom-empty-sz*[*of N*])
supply [[*goals-limit* = *1*]]
apply *sepref-dbg-keep*
apply *sepref-dbg-trans-keep*
apply *sepref-dbg-trans-step-keep*
apply *sepref-dbg-side-keep*
— We can start from this subgoal to find missing lemmas
oops

We can prove the remaining subgoal, e.g., by *auto* with the following lemma declared as introduction rule:

lemma *aux1*[*intro*]: $[\![\ \textit{set}\ l \subset \{0..<N\};\ \textit{distinct}\ l\ ]\!] \Longrightarrow \textit{length}\ l < N$
apply (*simp add*: *distinct-card*[*symmetric*])
apply (*drule psubset-card-mono*[*rotated*])
apply *auto*
done

We use some standard boilerplate to define the constant globally, although being inside the locale. This is required for code-generation.

sepref-thm *remdup1-fixed* **is** *remdup* :: $[\lambda l.\ \textit{set}\ l \subseteq \{0..<N\}]_a$ $(\textit{list-assn nat-assn})^k \rightarrow \textit{marl-assn N nat-assn}$
unfolding *remdup-def*[*abs-def*]
apply (*rewrite* **in** *nfoldli* - - - ⌑ *ias-sz.fold-custom-empty*[*of N*])
apply (*rewrite* **in** *nfoldli* - - - ⌑ *marl-fold-custom-empty-sz*[*of N*])
by *sepref*

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*
unfolding *test-remdup-def PR-CONST-def*
by *sepref*
concrete-definition (**in** −) *test-remdup1* **uses** *my-remdup-impl-loc.test-remdup1.refine-raw*
is (*uncurry0 ?f*,-)∈-
prepare-code-thms (**in** −) *test-remdup1-def*
lemmas *test-remdup1-refine*[*sepref-fr-rules*] = *test-remdup1.refine*[*of N*]

end

Outside the locale, a refinement of *my-remdup-impl-loc.test-remdup* also makes sense, however, with an extra argument *N*.

thm *test-remdup1.refine*
lemma *test-remdup1-refine-aux*: (*test-remdup1*, *my-remdup-impl-loc.test-remdup*)
∈ [*my-remdup-impl-loc*]$_a$ *nat-assn*^{k} → *nat-assn*
using *test-remdup1.refine*
unfolding *hfref-def hn-refine-def*
by (*auto simp*: *pure-def*)

We can also write a more direct precondition, as long as it implies the locale

lemma *test-remdup1-refine*: (*test-remdup1*, *my-remdup-impl-loc.test-remdup*) ∈ [λ*N*. *N*>*0*]$_a$ *nat-assn*^{k} → *nat-assn*
apply (*rule hfref-cons*[*OF test-remdup1-refine-aux - entt-refl entt-refl entt-refl*])
by *unfold-locales*

export-code *test-remdup1* **checking** *SML*

We can also register the abstract constant and the refinement, to use it in further refinements

sepref-register *my-remdup-impl-loc.test-remdup*
lemmas [*sepref-fr-rules*] = *test-remdup1-refine*

Static Data Structures with Custom Element Relations

In the previous section, we have presented a refinement using an array-list without dynamic resizing. However, the argument that we actually could append to this array was quite complicated.

Another possibility is to use bounded refinement relations, i.e., a refinement relation intersected with a condition for the abstract object. In our case, *b-assn id-assn* (λ*x*. *x* < *N*) relates natural numbers less than *N* to themselves.

We will repeat the above development, using the bounded relation approach:

definition *bremdup l* ≡ *do* {
(*s*,*r*) ← *nfoldli l* (λ-. *True*)
(λ*x* (*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* **by** *unfold-locales*

sepref-register *N*
lemma *N-hnr*[*sepref-import-param*]: (*N*,*N*)∈*nat-rel* **by** *simp*

Conceptually, what we insert in our list are elements, and these are less than *N*.

abbreviation *elem-assn* ≡ *nbn-assn N*

lemma *aux1*[*intro*]: ⟦ *set l* ⊂ {*0*..<*N*}; *distinct l* ⟧ ⟹ *length l* < *N*
apply (*simp add*: *distinct-card*[*symmetric*])
apply (*drule psubset-card-mono*[*rotated*])
apply *auto*
done

sepref-thm *remdup1-fixed* **is** *remdup* :: [λ*l*. *set l* ⊆ {*0*..<*N*}]$_a$ (*list-assn elem-assn*)^k → *marl-assn N elem-assn*
unfolding *remdup-def*[*abs-def*]
apply (*rewrite* **in** *nfoldli* - - - ⌑ *ias-sz.fold-custom-empty*[*of N*])
apply (*rewrite* **in** *nfoldli* - - - ⌑ *marl-fold-custom-empty-sz*[*of N*])
by *sepref*

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* ≡ *do* {*l* ← *remdup* [*0*..<*N*]; *RETURN* (*length l*) }
sepref-register *test-remdup*

This refinement depends on the (somewhat experimental) subtyping feature to convert from *id-assn* to *b-assn id-assn* (λ*x*. *x* < *N*), based on context information

sepref-thm *test-remdup1* **is**
uncurry0 (*PR-CONST test-remdup*) :: *unit-assn*^{k} →$_a$ *nat-assn*
unfolding *test-remdup-def PR-CONST-def*
by *sepref*

concrete-definition (**in** −) *test-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} → *nat-assn*
using *test-bremdup1.refine*
unfolding *hfref-def hn-refine-def*
by (*auto simp*: *pure-def*)

lemma *test-bremdup1-refine*: (*test-bremdup1*, *my-bremdup-impl-loc.test-remdup*) ∈ [λ*N*. *N*>*0*]$_a$ *nat-assn*^{k} → *nat-assn*
apply (*rule hfref-cons*[*OF test-bremdup1-refine-aux - entt-refl entt-refl entt-refl*])
by *unfold-locales*

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*
  apply *sepref-dbg-keep*
  apply *sepref-dbg-trans-keep*
  apply *sepref-dbg-trans-step-keep*
  supply [[*unify-trace-failure*, *goals-limit=1*]]

  — The problem manifests itself in trying to carry an abstract variable (the argument to *op-arl-empty-sz*) to the concrete program (the second argument of *hn-refine*). However, the concrete program can only depend on the concrete variables, so unification fails.
  oops

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*.

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} →$_a$ *id-assn*
unfolding *mtx-dup-diag-f-def*
by *sepref*

end

Then, we instantiate the corresponding locale, to get an implementation for array matrices. Note that we restrict ourselves to square matrices here:

interpretation *dup-diag*: *amtx-pointwise-unop-impl N N mtx-dup-diag-f id-assn mtx-dup-diag-f1*
apply *standard*
applyS (*simp add*: *mtx-dup-diag-f-def*) []
applyS (*rule mtx-dup-diag-f1.refine*)
done

We introduce an abbreviation for the abstract operation. Note: We do not have to register it (this is done once and for all for *mtx-pointwise-unop*), nor do we have to declare a refinement rule (done by *amtx-pointwise-unop-impl*-locale)

abbreviation *mtx-dup-diag* ≡ *mtx-pointwise-unop mtx-dup-diag-f*

The operation is usable now:

sepref-thm *mtx-dup-test* **is** λ*m*. *RETURN* (*mtx-dup-diag* (*mtx-dup-diag m*)) :: (*asmtx-assn N int-assn*)^d →$_a$ *asmtx-assn N int-assn*
by *sepref*

Similarly, there are operations to combine to matrices, and to compare two matrices:

interpretation *pw-add*: *amtx-pointwise-binop-impl N M* (((+))::(-::*monoid-add*) ⇒ -) *id-assn return oo* ((+))
for *N M*
apply *standard*
apply *simp*
apply (*sepref-to-hoare*) **apply** *sep-auto* — Alternative to synthesize concrete operation, for simple ad-hoc refinements
done
abbreviation *mtx-add* ≡ *mtx-pointwise-binop* ((+))

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 $\rightarrow_a$ *amtx-assn N M int-assn*
by *sepref*

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 $\rightarrow_a$ *amtx-assn N M int-assn*
apply *sepref-dbg-keep*
apply *sepref-dbg-trans-keep*
— We get stuck at a *COPY* goal, indicating that a matrix has to be copied.
apply *sepref-dbg-trans-step-keep*
— Which only works for pure refinements
oops

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 $\rightarrow_a$ *amtx-assn N M int-assn*
by *sepref*

A compare operation checks that all pairs of entries fulfill some property *f*, and at least one entry fullfills a property *g*.

interpretation *pw-lt*: *amtx-pointwise-cmpop-impl N M* (($\leq$)::(-::*order*) $\Rightarrow$ -) (($\neq$)::(-::*order*) $\Rightarrow$ -) *id-assn return oo* ($\leq$) *return oo* ($\neq$)
for *N M*
apply *standard*
apply *simp*
apply *simp*
apply (*sepref-to-hoare*) **apply** *sep-auto*
apply (*sepref-to-hoare*) **apply** *sep-auto*
done
abbreviation *mtx-lt* $\equiv$ *mtx-pointwise-cmpop* ($\leq$) ($\neq$)

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 $\rightarrow_a$ *bool-assn*
by *sepref* — Note: Better fold over single matrix (currently no locale for that), instead of creating a new matrix.

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*
unfolding *co-set-gen-def*
apply (*rewrite hs.fold-custom-empty*)
apply *sepref-dbg-keep*
apply *sepref-dbg-trans-keep*
— We run into the problem that the Sepref tool uses *nat-assn* to refine natural numbers, and only later tries to convert it to *nbn-assn*. However, at this point, the information is already lost.
oops

We can use a feature of Sepref, to annotate the desired assertion directly into the abstract program. For this, we use *annotate-assn*, which inserts the (special) constant *ASSN-ANNOT*, which is just identity, but enforces refinement with the given assertion.

sepref-definition *co-set-gen1* **is** *uncurry0* (*PR-CONST co-set-gen*) :: *unit-assn*^{k} →$_a$ *hs.assn co-assn*
unfolding *co-set-gen-def PR-CONST-def*
apply (*rewrite hs.fold-custom-empty*)
apply (*rewrite* **in** *insert* ¤ - *annotate-assn*[**where** *A*=*co-assn*])
— Annotate the pair as coordinate before insertion
by *sepref*
lemmas [*sepref-fr-rules*] = *co-set-gen1.refine*

sepref-register *co-set-gen*

Now we can use the entries from the set as coordinates, without any worries about them being out of range

sepref-thm *co-set-use* **is** (λ*m*. *do* {
co ← *co-set-gen*;
FOREACH co (λ(*i*,*j*) *m*. *RETURN* (*m*((*i*,*j*) := *1*))) *m*
}) :: *mtx-assn*^{d} →$_a$ *mtx-assn*

by *sepref*

end

4.1.10 Type Classes

TBD

4.1.11 Higher-Order

TBD

4.1.12 A-Posteriori Optimizations

The theorem collection *sepref-opt-simps* and *sepref-opt-simps2* contain simplifier lemmas that are applied, in two stages, to the generated Imperative/HOL program.

This is the place where some optimizations, such as deforestation, and simplifying monad-expressions using the monad laws, take place.

thm *sepref-opt-simps*
thm *sepref-opt-simps2*

4.1.13 Short-Circuit Evaluation

Consider

sepref-thm *test-sc-eval* **is** $RETURN\ o\ (\lambda l.\ length\ l > 0 \wedge hd\ l) :: (list\text{-}assn\ bool\text{-}assn)^k \rightarrow_a bool\text{-}assn$
  apply *sepref-dbg-keep*
  apply *sepref-dbg-trans-keep*
  apply *sepref-dbg-trans-step-keep*
  — Got stuck, as the operands of $\wedge$ are evaluated before applying the operator, i.e., *hd* is also applied to empty lists
  oops

sepref-thm *test-sc-eval* **is** $RETURN\ o\ (\lambda l.\ length\ l > 0 \wedge hd\ l) :: (list\text{-}assn\ bool\text{-}assn)^k \rightarrow_a bool\text{-}assn$
  unfolding *short-circuit-conv* — Enables short-circuit evaluation by rewriting $\wedge$, $\vee$, and $\longrightarrow$ to *if*-expressions
  by *sepref*

end

4.2 Reference Guide

theory *Sepref-Guide-Reference*

imports *../IICF/IICF*
begin

This guide contains a short reference of the most important Sepref commands, methods, and attributes, as well as a short description of the internal working, and troubleshooting information with examples.

Note: To get an impression how to actually use the Sepref-tool, read the quickstart guide first!

4.2.1 The Sepref Method

The *sepref* method is the central method of the tool. Given a schematic goal of the form *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*) })
  by *sepref*

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*)}))
  ∈ $nat\text{-}assn^k$ $*_a$ (*array-assn bool-assn*)^k $\to_a$ *bool-assn*
  by *sepref*

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*
    apply *sepref-dbg-keep*

apply *sepref-dbg-id-keep*
— Forgot to register n
oops

Solution: Register n, be careful not to export meaningless registrations from context!

```
context
  notes [[sepref-register-adhoc N]]
begin
   sepref-thm N-plus-2-example is uncurry0 (RETURN (N+2)) :: unit-assn^k
→_a nat-assn by sepref
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
  apply sepref-dbg-keep
  apply sepref-dbg-trans-keep
  — Stuck at refinement for function update on map
  oops
```

Solution: Register with correct interface type

```
sepref-register my-map :: ('k,'v) i-map
sepref-thm my-map-example is uncurry0 (RETURN (my-map(False↦1))) :: unit-assn^k
→_a hm.assn bool-assn nat-assn
  by sepref
```

Monadify Phase

The monadify phase rewrites the program such that every operation becomes visible on the monad level, that is, nested HOL-expressions are flattened. Also combinators (e.g. if, fold, case) may get flattened, if special rules are registered for that.

Moreover, the monadify phase fixes the number of operands applied to an operation, using eta-expansion to add missing operands.

Finally, the monadify phase handles duplicate parameters to an operation, by inserting a *COPY* tag. This is necessary as our tool expects the parameters of a function to be separate, even for read-only parameters[1].

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*

[1] Using fractional permissions or some other more fine grained ownership model might lift this restriction in the future.

stops at a failing sub-phase and presents the internal goal state before the failing sub-phase.

Monadify: Arity

In the first sub-phase, the rules from *sepref-monadify-arity* are used to standardize the number of operands applied to a constant. The rules work by rewriting each constant to a lambda-expression with the desired number of arguments, and the using beta-reduction to account for already existing arguments. Also higher-order arguments can be enforced, for example, the rule for fold enforces three arguments, the function itself having two arguments ($fold \equiv \lambda x.\ (\#\lambda xa.\ (\#\lambda xb.\ (\#SP\ fold\ \$\ (\lambda xa.\ (\#\lambda xb.\ (\#x\ \$\ xa\ \$\ xb\#)\#))\ \$\ xa\ \$\ xb\#)\#)\#)$).

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-option* $ *?fn* $ (λ*x*. (#*?fs x*#)) $ *?ov*) ≡ (≫=) $ (*EVAL* $ *?ov*) $ (λ*x*. (#*case-option* $ (*EVAL* $ *?fn*) $ (λ*x*. (#*EVAL* $ *?fs x*#)) $ *x*#))
EVAL $ (*Let* $ *?v* $ (λ*x*. (#*?f x*#))) ≡ (≫=) $ (*EVAL* $ *?v*) $ (λ*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*
  apply *sepref-dbg-keep*
  apply *sepref-dbg-monadify-keep*
  — An *EVAL*-tag with an abstraction remains. This is b/c the default heuristics tries to interpret the function inside the fold as a plain value argument.
  oops

Solution: Register appropriate arity and combinator-rules

lemma *my-fold-arity*[*sepref-monadify-arity*]: *my-fold* ≡ λ$_2$*f l s*. *SP my-fold*$(λ$_2$*x s*. *f*$*x*$*s*)$*l*$*s* **by** *auto*

The combinator-rule rewrites to the already existing and set up combinator *nfoldli*:

lemma *monadify-plain-my-fold*[*sepref-monadify-comb*]:
  EVAL$(*my-fold*$(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*))
  by (*simp add*: *fold-eq-nfoldli my-fold-def*)

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*
  by *sepref*

Monadify: Dup

The last three phases, *mark-params*, *dup*, *remove-pass* are to detect duplicate parameters, and insert *COPY* tags. The first phase, *mark-params*, adds *PASS* tags around all parameters. Parameters are bound variables and terms that have a refinement in the precondition.

The second phase detects duplicate parameters and inserts *COPY* tags to remove them. Finally, the last phase removes the *PASS* tags again.

The methods *sepref-dbg-monadify-mark-params*, *sepref-dbg-monadify-dup*, and *sepref-dbg-monadify-remove-pass* gives you access to these phases.

Monadify: Step-Through Example

We give an annotated example of the monadify phase. Note that the program utilizes a few features of monadify:

- The fold function is higher-order, and gets flattened
- The first argument to fold is eta-contracted. The missing argument is added.
- The multiplication uses the same argument twice. A copy-tag is inserted.

sepref-thm *monadify-step-thru-test* **is** $\lambda l.$ *do* {
  let i = *length* l;
  RETURN (*fold* ($\lambda x.$ (+) (x∗x)) l i)
} :: (*list-assn nat-assn*)^k →$_a$ *nat-assn*
apply *sepref-dbg-preproc*
apply *sepref-dbg-cons-init*
apply *sepref-dbg-id*

apply *sepref-dbg-monadify-arity* — Second operand of fold-function is added
apply *sepref-dbg-monadify-comb* — Flattened. *fold* rewritten to *monadic-nfoldli*.

apply *sepref-dbg-monadify-check-EVAL* — No *EVAL* tags left
apply *sepref-dbg-monadify-mark-params* — Parameters marked by *PASS*. Note the multiplication x∗x.

apply *sepref-dbg-monadify-dup* — *COPY* tag inserted.

apply *sepref-dbg-monadify-remove-pass* — *PASS* tag removed again

apply *sepref-dbg-opt-init*
apply *sepref-dbg-trans*
apply *sepref-dbg-opt*

apply *sepref-dbg-cons-solve*
apply *sepref-dbg-cons-solve*
apply *sepref-dbg-constraints*
done

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 ... 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* $*$ *hn-ctxt ?A ?x ?y* $\Longrightarrow_t$ *?F'* $*$ *hn-ctxt ?A ?x ?y*

?F $\Longrightarrow_t$ *?F'* $\Longrightarrow$ *?F* $*$ *hn-ctxt ?A ?x ?y* $\Longrightarrow_t$ *?F'*

?P $\Longrightarrow_t$ *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'*

hn-ctxt ?R ?x ?y $\Longrightarrow_t$ *hn-invalid ?R ?x ?y*

hn-ctxt ?R ?x ?y $\Longrightarrow_t$ *hn-ctxt* (λ- -. *true*) *?x ?y*

CONSTRAINT is-pure ?R $\Longrightarrow$ *hn-invalid ?R ?x ?y* $\Longrightarrow_t$ *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*

$[\![$*hn-ctxt ?R1.0 ?x ?x'* $\vee_A$ *hn-ctxt ?R2.0 ?x ?x'* $\Longrightarrow_t$ *hn-ctxt ?R ?x ?x'*; *?Fl* $\vee_A$ *?Fr* $\Longrightarrow_t$ *?F*$]\!] \Longrightarrow$ *?Fl* $*$ *hn-ctxt ?R1.0 ?x ?x'* $\vee_A$ *?Fr* $*$ *hn-ctxt ?R2.0 ?x ?x'* $\Longrightarrow_t$ *?F* $*$ *hn-ctxt ?R ?x ?x'*

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'* 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: ($\bigwedge x\ x'$. $[\![x \in$ *set ?l*; $x' \in$ *set ?l*$]\!] \Longrightarrow$ *hn-ctxt ?A x x'* $\Longrightarrow_t$ *hn-ctxt ?A' x x'*$) \Longrightarrow$ *hn-ctxt* (*list-assn ?A*) *?l ?l'* $\Longrightarrow_t$ *hn-ctxt* (*list-assn ?A'*) *?l ?l'*. An example for semantic information is the bounded assertion, which intersects any given assertion with a predicate on the abstract domain. The frame matcher is set up such that it can convert between bounded assertions, generating semantic side conditions to discharge implications between bounds ($[\![$*hn-ctxt* (*b-assn ?A ?P*) *?x*

?y $\Longrightarrow_t$ *hn-ctxt ?A′ ?x ?y*; ⟦*vassn-tag* (*hn-ctxt ?A ?x ?y*); *vassn-tag* (*hn-ctxt ?A′ ?x ?y*); *?P ?x*⟧ $\Longrightarrow$ *?P′ ?x*⟧ $\Longrightarrow$ *hn-ctxt* (*b-assn ?A ?P*) *?x ?y* $\Longrightarrow_t$ *hn-ctxt* (*b-assn ?A′ ?P′*) *?x ?y*).

This is essentially a subtyping mechanism on the level of refinement assertions, which is quite useful for maintaining natural side conditions on operands. A standard example is to maintain a list of array indices: The refinement assertion for array indices is *id-assn* restricted to indices that are in range: *b-assn id-assn* ($\lambda x.\ x < N$). When inserting natural numbers into this list, one has to prove that they are actually in range (conversion from *id-assn* to $\lambda n.$ *b-assn id-assn* ($\lambda x.\ x < n$)). Elements of the list can be used as natural numbers (conversion from $\lambda n.$ *b-assn id-assn* ($\lambda x.\ x < n$) to *id-assn*). Additionally, the side condition solver can derive that the predicate holds on the abstract variable (via the *vassn-tag* inserted by the operator steps).

Translation: Annotated Example

context
  fixes *N*::*nat*
  notes [[*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*)

  apply *sepref-dbg-preproc*
  apply *sepref-dbg-cons-init*
  apply *sepref-dbg-id*

apply *sepref-dbg-monadify*

apply *sepref-dbg-opt-init*

apply *sepref-dbg-trans-step* — Combinator rule for bind, generating two *hn-refine* goals, and a frame rule to separate the bound variable from the rest.
apply *sepref-dbg-trans-step* — Rule for empty hashset, solves goal
apply *sepref-dbg-trans-step* — Combinator rule for nfoldli ($[\![$*INDEP ?Rk*; *INDEP ?Rσ*; *?P* $\Longrightarrow_t$ *?Γ* $*$ *hn-ctxt* (*list-assn ?Rk*) *?s′ ?s* $*$ *hn-ctxt ?Rσ ?σ′ ?σ*; $\bigwedge\sigma$ σ'. *hn-refine* (*?Γ* $*$ *hn-ctxt ?Rσ σ′ σ*) (*?c σ*) (*?Γc σ′ σ*) *id-assn* (*?c′ σ′*); $\bigwedge\sigma'$ σ. *TERM monadic-nfoldli* $\Longrightarrow$ *?Γc σ′ σ* $\Longrightarrow_t$ *?Γ* $*$ *hn-ctxt ?Rσ σ′ σ*; $\bigwedge x'$ x σ' σ. $x' \in$ *set ?s′* $\Longrightarrow$ *hn-refine* (*?Γ* $*$ *hn-ctxt ?Rk x′ x* $*$ *hn-ctxt ?Rσ σ′ σ*) (*?f x σ*) (*?Γf x′ x σ′ σ*) *?Rσ* (*?f′ x′ σ′*); $\bigwedge x'$ x σ' σ. *TERM monadic-nfoldli* $\Longrightarrow$ *?Γf x′ x σ′ σ* $\Longrightarrow_t$ *?Γ* $*$ *hn-ctxt ?Rk′ x′ x* $*$ *hn-ctxt ?Pfσ σ′ σ*; $\bigwedge x$ x'. *hn-ctxt ?Rk′ x′ x* $\vee_A$ *hn-ctxt ?Rk x′ x* $\Longrightarrow_t$ *hn-ctxt ?Rk″ x′ x*$]\!]$ $\Longrightarrow$ *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′* $ (λx. (#*?c′ x*#)) $ (λx. (#λxa. (#*?f′ x xa*#)#)) $ *?σ′*))
  apply *sepref-dbg-trans-step* — INDEP
  apply *sepref-dbg-trans-step* — INDEP
  apply *sepref-dbg-trans-step* — Frame to get list and initial state
  apply *sepref-dbg-trans-step* — Refinement of continuation condition
  apply *sepref-dbg-trans-step* — Frame to recover state after continuation condition

  — Loop body
  apply *sepref-dbg-trans-step*
  apply *sepref-dbg-trans-step*
  apply *sepref-dbg-trans-step*
  apply *sepref-dbg-trans-step*
  apply *sepref-dbg-trans-step*
  apply *sepref-dbg-trans-step*
  apply *sepref-dbg-trans-step*
  apply *sepref-dbg-trans-step*
  — At this point, we arrived at the *nbn-rel* annotation. There is enough information to show $x'a < N$
  apply *sepref-dbg-trans-step*
  apply *sepref-dbg-trans-step*
  apply *sepref-dbg-trans-step*
  apply *sepref-dbg-trans-step*
  — At this point, we have to merge the postconditions from the two if branches. *nat-rel* gets merged with *invalid-assn* (*nbn-assn n*), yielding *invalid-assn nat-assn*
  apply *sepref-dbg-trans-step*
  apply *sepref-dbg-trans-step* — Frame rule separating bound variable from rest

  apply *sepref-dbg-trans-step* — Frame rule separating fold-state from rest
 apply *sepref-dbg-trans-step* — Merging elements of list before body with elements of list after body, to get refinement for resulting list

apply *sepref-dbg-trans-step* — Frame rule from initial bind, separating bound variable from the rest

apply *sepref-dbg-opt*
apply *sepref-dbg-cons-solve* — Frame rule, recovering the invalidated list or pure elements, propagating recovery over the list structure
apply *sepref-dbg-cons-solve* — Trivial frame rule
apply *sepref-dbg-constraints*
done

end

Optimization Phase

The optimization phase simplifies the generated program, first with *sepref-opt-simps*, and then with *sepref-opt-simps2*. For simplification, the tag *CNV* is used, which is discharged with *CNV ?x ?x* after simplification.

Method *sepref-dbg-opt* gives direct access to this phase. The simplification is used to beautify the generated code. The most important simplifications collapse code that does not depend on the heap to plain expressions (using the monad laws), and apply certain deforestation optimizations.

Consider the following example:

sepref-thm *opt-example* **is** $\lambda n.$ *do* { *let r* = *fold* (+) [*1*..<*n*] *0*; *RETURN* (*n*∗*n*+*2*) }
:: $nat\text{-}assn^k \rightarrow_a nat\text{-}assn$
apply *sepref-dbg-preproc*
apply *sepref-dbg-cons-init*
apply *sepref-dbg-id*
apply *sepref-dbg-monadify*
apply *sepref-dbg-opt-init*
apply *sepref-dbg-trans*
— The generated program contains many superfluous binds, moreover, it actually generates a list and then folds over it
supply [[*show-main-goal*]]
apply *sepref-dbg-opt*
— The superfluous binds have been collapsed, and the fold over the list has been replaced by *imp-for′*, which uses a counter.
apply *sepref-dbg-cons-solve*
apply *sepref-dbg-cons-solve*
apply *sepref-dbg-constraints*
done

Cons-Solve Phases

These two phases, accessible via *sepref-dbg-cons-solve*, applies the frame inference tool to solve the two implications generated by the consequence rule phase.

Constraints Phase

This phase, accessible via *sepref-dbg-constraints*, solve the deferred constraints that are left, and then removes the *CONSTRAINT-SLOT* subgoal.

4.2.2 Refinement Rules

There are two forms of specifying refinement between an Imperative/HOL program and an abstract program in the *nres*-monad. The *hn-refine* form (also hnr-form) is the more low-level form. The term $P \Longrightarrow$ *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 $PC \Longrightarrow (uncurry_x\ f,\ uncurry_x\ g) \in [P]_a\ {A_1}^{k1}\ *_a$ ... $*_a\ {A_n}^{kn} \to R$, 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: $Args \rightarrow_a R \equiv Args \rightarrow_a R$.

For example, consider *arl-swap-hnr* [*unfolded pre-list-swap-def*]. It reads $CONSTRAINT\ is\text{-}pure\ A \Longrightarrow (uncurry2\ arl\text{-}swap,\ uncurry2\ (RETURN \circ\circ\circ\ op\text{-}list\text{-}swap)) \in [\lambda((l,\ i),\ j).\ i < length\ l \wedge j < length\ l]_a\ (arl\text{-}assn\ A)^d *_a id\text{-}assn^k *_a id\text{-}assn^k \rightarrow arl\text{-}assn\ A$

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. $((@), (@)) \in \langle A\rangle list\text{-}rel \rightarrow \langle A\rangle list\text{-}rel \rightarrow \langle A\rangle list\text{-}rel$), 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-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*↦*n*+*1*)
definition *map-op2 m n* ≡ *m*(*n*↦*n*+*2*)

definition *map-op3 m n* ≡ *m*(*n*↦*n*+*3*)
definition *map-op-to-map* (*m*::*'a*⇀*'b*) ≡ *m*

context
  notes [*map-type-eqs*] = *map-type-eqI*[*of TYPE*(*'a*⇀*'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*⇀*'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

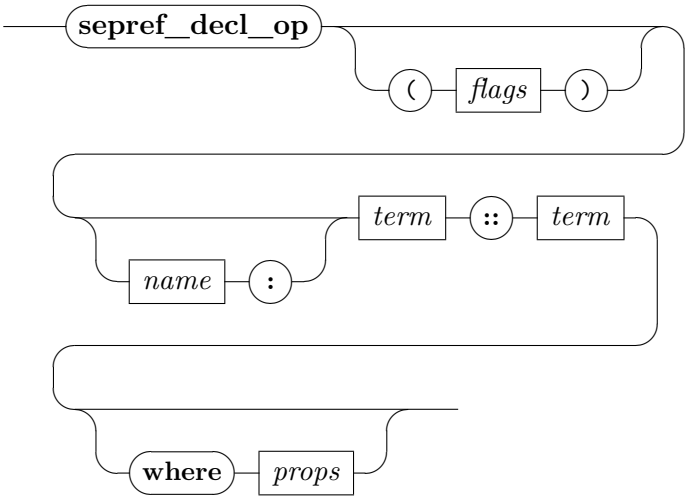
The following command fails, as the specified interface type does not correspond to the HOL type of the term: **sepref-register** *hd* :: (*nat*,*nat*) *i-map*

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.

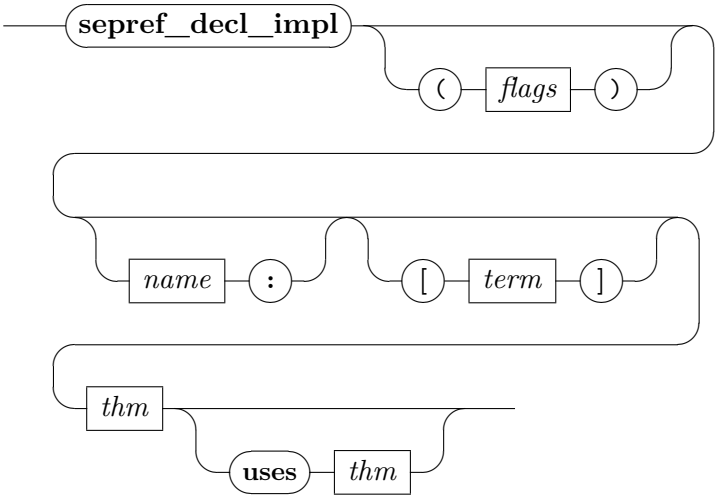
A list of properties (separated by space and/or *and*) can be specified, which get constraint-preconditions of the relation.
Finally, the following flags can be specified. Each flag can be prefixed by *no-* to invert its meaning:

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
have *card* $\{x.\ 3<x \wedge x<(7::nat)\} = card\ \{x.\ 4\leq x \wedge x\leq(6::nat)\}$

apply (*fo-rule arg-cong*)
apply *auto*
done

— While the first goal could also have been solved with *rule arg-cong*[*where f*=*card*], things would be much more verbose for the following goal. (Such goals actually occur in practice!)

fix f :: *nat set* ⇒ *nat set* ⇒ *bool*
have $\bigwedge a.\ f\ \{x.\ x*2 + a + 3 < 10\}\ \{x.\ 3<x \wedge x<(7::nat)\} = f\ \{x.\ x*2 + a \leq 6\}\ \{x.\ 4\leq x \wedge x\leq(6::nat)\}$
apply (*fo-rule arg-cong fun-cong cong*)+
apply *auto*
done

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* **by** *simp*

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
unfolding *op-graph-succ-def*[*abs-def*] *graph-rel-def*
by *parametricity*

context begin
private definition *graph-α1 l* ≡ { (*i*,*j*). *i*<*length l* ∧ *j*∈*l*!*i* }

private definition *graph-rel1* ≡ *br graph-α1* (λ-. *True*)

private definition *succ1 l i* ≡ *if i*<*length l then l*!*i else* {}

private lemma *succ1-refine*: (*succ1*,*op-graph-succ*) ∈ *graph-rel1* → *Id* → ⟨*Id*⟩*set-rel*
by (*auto simp*: *graph-rel1-def graph-α1-def br-def succ1-def split*: *if-split-asm intro*!: *ext*)

private definition *assn2* ≡ *array-assn* (*pure* (⟨*Id*⟩*list-set-rel*))

definition *adjg-assn A* ≡ *hr-comp* (*hr-comp assn2 graph-rel1*) (⟨*the-pure A*⟩*graph-rel*)

context
notes [*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} →$_a$ *pure* (⟨*Id*⟩*list-set-rel*))
unfolding *succ1-def*[*abs-def*] *assn2-def*
by *sepref*

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} →$_a$ *pure* (⟨*the-pure A*⟩*list-set-rel*)
using *succ2.refine*[*FCOMP succ1-refine*, *FCOMP param-op-graph-succ*, *simplified*, *of A*]
by (*simp add*: *IS-PURE-def list-set-rel-compp*)

end

end

lemma [*intf-of-assn*]:
intf-of-assn A (*i*::*'I itself*) ⟹ *intf-of-assn* (*adjg-assn A*) *TYPE*(*'I i-graph*) **by** *simp*

definition *cr-graph*
:: *nat* ⇒ (*nat* × *nat*) *list* ⇒ *nat list Heap.array Heap*
where
cr-graph numV Es ≡ *do* {
a ← *Array.new numV* [];
a ← *imp-nfoldli Es* (λ-. *return True*) (λ(*u*,*v*) *a*. *do* {
l ← *Array.nth a u*;
let l = *v*#*l*;

```
    a ← 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 ⇒ bool
begin
  definition dfs :: ('v set × bool) nres where
    dfs ≡ do {
      (V,r) ← RECT (λdfs (V,v).
        if v∈V then RETURN (V,False)
        else do {
          let V=insert v V;
          if tgt v then
            RETURN (V,True)
          else
            FOREACH_C (E``{v}) (λ(-,b). ¬b) (λv' (V,-). dfs (V,v')) (V,False)
        }
      ) ({},v0);
      RETURN (V,r)
    }

  definition reachable ≡ {v. (v0,v)∈E*}
```

definition *dfs-spec* ≡ *SPEC* (λ(*V*,*r*). (*r* ⟷ *reachable*∩*Collect tgt*≠{}) ∧ (¬*r* ⟶ *V*=*reachable*))

lemma *dfs-correct*:
assumes *fr*: *finite reachable*
shows *dfs* ≤ *dfs-spec*
proof −
have *F*: ⋀*v*. *v*∈*reachable* ⟹ *finite* (*E*‘‘{*v*})
using *fr*
apply (*auto simp*: *reachable-def*)
by (*metis* (*mono-tags*) *Image-singleton Image-singleton-iff*
finite-subset rtrancl.rtrancl-into-rtrancl subsetI)

define *rpre* **where** *rpre* = (λ*S* (*V*,*v*).
v∈*reachable*
∧ *V*⊆*reachable*
∧ *S*⊆*V*
∧ (*V* ∩ *Collect tgt* = {})
∧ *E*‘‘(*V*−*S*) ⊆ *V*)

define *rpost* **where** *rpost* = (λ*S* (*V*,*v*) (*V*′,*r*).
(*r*⟷*V*′∩*Collect tgt* ≠ {})
∧ *V*⊆*V*′
∧ *v*∈*V*′
∧ *V*′⊆*reachable*
∧ (¬*r* ⟶ (*E*‘‘(*V*′−*S*) ⊆ *V*′)))

define *fe-inv* **where** *fe-inv* = (λ*S V v it* (*V*′,*r*).
(*r*⟷*V*′∩*Collect tgt* ≠ {})
∧ *insert v V*⊆*V*′
∧ *E*‘‘{*v*} − *it* ⊆ *V*′
∧ *V*′⊆*reachable*
∧ *S*⊆*insert v V*
∧ (¬*r* ⟶ *E*‘‘(*V*′−*S*) ⊆ *V*′ ∪ *it* ∧ *E*‘‘(*V*′−*insert v S*) ⊆ *V*′))

have *vc-pre-initial*: *rpre* {} ({}, *v0*)
by (*auto simp*: *rpre-def reachable-def*)

{

fix *S V v*
assume *rpre S* (*V*,*v*)
and *v*∈*V*
hence *rpost S* (*V*,*v*) (*V*,*False*)
unfolding *rpre-def rpost-def*
by *auto*
} note *vc-node-visited* = *this*

```
{

 fix S V v
 assume tgt v
 and rpre S (V,v)
 hence rpost S (V,v) (insert v V, True)
   unfolding rpre-def rpost-def
   by auto
} note vc-node-found = this

{
 fix S V v
 assume rpre S (V, v)
 hence finite (E‘‘{v})
   unfolding rpre-def using F by (auto)
} note vc-foreach-finite = this

{

 fix S V v
 assume A: v ∉ V ¬tgt v
   and PRE: rpre S (V, v)
 hence fe-inv S V v (E‘‘{v}) (insert v V, False)
   unfolding fe-inv-def rpre-def by (auto)
} note vc-enter-foreach = this

{

 fix S V v v' it V'
 assume A: v ∉ V ¬tgt v v' ∈ it it ⊆ E‘‘{v}
   and FEI: fe-inv S V v it (V', False)
   and PRE: rpre S (V, v)

 from A have v' ∈ E‘‘{v} by auto
 moreover from PRE have v ∈ reachable by (auto simp: rpre-def)
 hence E‘‘{v} ⊆ reachable by (auto simp: reachable-def)
 ultimately have [simp]: v'∈reachable by blast

 have rpre (insert v S) (V', v')
   unfolding rpre-def
   using FEI PRE by (auto simp: fe-inv-def rpre-def) []
} note vc-rec-pre = this

{

 fix S V V' v v' it Vr''
 assume fe-inv S V v it (V', False)
   and rpost (insert v S) (V', v') Vr''
```

hence *fe-inv S V v* (*it* − {*v'*}) *Vr''*
unfolding *rpre-def rpost-def fe-inv-def*
by *clarsimp blast*
} **note** *vc-iterate-foreach* = *this*

{

fix *S V v V'*
assume *PRE*: *rpre S* (*V*, *v*)
assume *A*: *v* ∉ *V* ¬*tgt v*
assume *FEI*: *fe-inv S V v* {} (*V'*, *False*)
have *rpost S* (*V*, *v*) (*V'*, *False*)
unfolding *rpost-def*
using *FEI* **by** (*auto simp*: *fe-inv-def*) []
} **note** *vc-foreach-completed-imp-post* = *this*

{

fix *S V v V' it*
assume *PRE*: *rpre S* (*V*, *v*)
and *A*: *v* ∉ *V* ¬*tgt v it* ⊆ *E*``{*v*}
and *FEI*: *fe-inv S V v it* (*V'*, *True*)
hence *rpost S* (*V*, *v*) (*V'*, *True*)
by (*auto simp add*: *rpre-def rpost-def fe-inv-def*) []
} **note** *vc-foreach-interrupted-imp-post* = *this*

{
fix *V r*
assume *rpost* {} ({}, *v0*) (*V*, *r*)
hence (*r* ⟷ *reachable* ∩ *Collect tgt* ≠ {}) ∧ (¬*r* ⟶ *V*=*reachable*)
by (*auto*
simp: *rpost-def reachable-def*
dest: *Image-closed-trancl*
intro: *rev-ImageI*)
} **note** *vc-rpost-imp-spec* = *this*

show *?thesis*
unfolding *dfs-def dfs-spec-def*
apply (*refine-rcg refine-vcg*)
apply (*rule order-trans*)

apply(*rule RECT-rule-arb*[**where**
pre=*rpre*
and *M*=λ*a x*. *SPEC* (*rpost a x*)
and *V*=*finite-psupset reachable* <∗*lex*∗> {}
])
apply *refine-mono*
apply (*blast intro*: *fr*)
apply (*rule vc-pre-initial*)

```
    apply (refine-rcg refine-vcg
      FOREACHc-rule'[where I=fe-inv S v s for S v s]
      )
    apply (simp-all add: vc-node-visited vc-node-found)

    apply (simp add: vc-foreach-finite)

    apply (auto intro: vc-enter-foreach) []

    apply (rule order-trans)
    apply (rprems)
    apply (erule (5) vc-rec-pre)
      apply (auto simp add: fe-inv-def finite-psupset-def) []
      apply (refine-rcg refine-vcg)
      apply (simp add: vc-iterate-foreach)

    apply (auto simp add: vc-foreach-completed-imp-post) []

    apply (auto simp add: vc-foreach-interrupted-imp-post) []

    apply (auto dest: vc-rpost-imp-spec) []
    done
  qed

end

lemma dfs-correct': (uncurry2 dfs, uncurry2 dfs-spec)
  ∈ [λ((E,s),t). finite (reachable E s)]_f ((Id×_rId)×_rId) → ⟨Id⟩nres-rel
  apply (intro frefI nres-relI; clarsimp)
  by (rule dfs-correct)
```

5.2.2 Refinement to Imperative/HOL

We set up a schematic proof goal, and use the sepref-tool to synthesize the implementation.

```
sepref-definition dfs-impl is
   uncurry2 dfs :: (adjg-assn nat-assn)^k *_a nat-assn^k *_a (pure (nat-rel → bool-rel))^k
→_a prod-assn (ias.assn nat-assn) bool-assn
  unfolding dfs-def[abs-def] — Unfold definition of DFS
  using [[goals-limit = 1]]
  apply (rewrite in RECT - (¤,-) ias.fold-custom-empty) — Select impls
  apply (rewrite in if ¤ then RETURN (-,True) else - fold-pho-apply)
  apply sepref — Invoke sepref-tool
  done
export-code dfs-impl checking SML-imp
  — Generate SML code with Imperative/HOL

export-code dfs-impl in Haskell module-name DFS
```

Finally, correctness is shown by combining the generated refinement theorem with the abstract correctness theorem.

lemmas *dfs-impl-correct'* = *dfs-impl.refine*[*FCOMP dfs-correct'*]

corollary *dfs-impl-correct*:
finite (*reachable E s*) ⟹
<*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*)) >$_t$
using *dfs-impl-correct'*[*THEN hfrefD*, *THEN hn-refineD*, *of* ((*E*,*s*),*tgt*) ((*Ei*,*s*),*tgt*), *simplified*]
apply (*rule cons-rule*[*rotated* −*1*])
apply (*sep-auto intro*!: *ent-ex-preI simp*: *dfs-spec-def pure-def*)+
done

end

5.3 Imperative Implementation of Dijkstra's Shortest Paths Algorithm

theory *Sepref-Dijkstra*
imports
../IICF/IICF
../Sepref-ICF-Bindings
Dijkstra-Shortest-Path.Dijkstra
Dijkstra-Shortest-Path.Test
HOL−Library.Code-Target-Numeral

Sepref-WGraph
begin

instantiation *infty* :: (*heap*) *heap*
begin
instance
apply *standard*
apply (*rule-tac x*=λ*Infty* ⇒ *0* | *Num a* ⇒ *to-nat a* + *1* **in** *exI*)
apply (*rule injI*)
apply (*auto split*: *infty.splits*)
done
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*)
apply (*intro ext*)
subgoal for *x y* **by** (*cases x*; *cases y*; *simp add*: *pure-def*)
done

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*)
by (*auto simp*: *is-pure-conv infty-assn-pure-conv*)

lemma *infty-assn-id*[*simp*]: *infty-assn id-assn* = *id-assn*
by (*simp add*: *infty-assn-pure-conv*)

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

by (*auto simp*: *infty-rel-def IS-BELOW-ID-def*)

sepref-register *Num Infty*

lemma *Num-hnr*[*sepref-fr-rules*]: (*return o Num*,*RETURN o Num*)∈A^d →$_a$ *infty-assn A*
by *sepref-to-hoare sep-auto*

lemma *Infty-hnr*[*sepref-fr-rules*]: (*uncurry0* (*return Infty*),*uncurry0* (*RETURN Infty*))∈*unit-assn*^{k} →$_a$ *infty-assn A*
by *sepref-to-hoare sep-auto*

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*
by *simp*
lemma [*sepref-monadify-comb*]: *case-infty*$*f1*$*f2*$*x* ≡ (≫=)$(*EVAL*$*x*)$(λ$_2$*x. SP case-infty*$*f1*$*f2*$*x*) **by** *simp*
lemma [*sepref-monadify-comb*]: *EVAL*$(*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*)
apply (*rule eq-reflection*)
by (*simp split*: *infty.splits*)

lemma *infty-assn-ctxt*: *infty-assn A x y* = *z* ⟹ *hn-ctxt* (*infty-assn A*) *x y* = *z*
by (*simp add*: *hn-ctxt-def*)

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*)
apply (*rule hn-refine-cons-pre*[*OF FR*])
apply1 *extract-hnr-invalids*
apply (*cases e*; *cases e′*; *simp add*: *infty-assn.simps*[*THEN infty-assn-ctxt*])
subgoal
apply (*rule hn-refine-cons*[*OF - Infty - entt-refl*]; *assumption?*)
applyS (*simp add*: *hn-ctxt-def*)
apply (*subst mult.commute*, *rule entt-fr-drop*)
apply (*rule entt-trans*[*OF - MERGE2*])
apply (*simp add*:)
done
subgoal
apply (*rule hn-refine-cons*[*OF - Num - entt-refl*]; *assumption?*)
applyS (*simp add*: *hn-ctxt-def*)
apply (*rule entt-star-mono*)
apply1 (*rule entt-fr-drop*)
applyS (*simp add*: *hn-ctxt-def*)
apply1 (*rule entt-trans*[*OF - MERGE2*])
applyS (*simp add*:)
done
done

lemma *hnr-val*[*sepref-fr-rules*]: (*return o Weight.val*,*RETURN o Weight.val*) ∈ [λ*x*. *x*≠*Infty*]$_a$ (*infty-assn A*)^d → *A*
apply *sepref-to-hoare*
subgoal for *x y* **by** (*cases x*; *cases y*; *sep-auto*)
done

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*)
unfolding *infty-plus-eq-plus*[*symmetric*] *infty-plus-def*[*abs-def*]
by *sepref*
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*
unfolding *infty-less-def*[*abs-def*]
by *parametricity*

lemma *infty-less-eq-less*: *infty-less* (<) = (<)
unfolding *infty-less-def*[*abs-def*]
apply (*clarsimp intro*!: *ext*)
subgoal for *a b* **by** (*cases a*; *cases b*; *auto*)
done

context
fixes *A* :: *'a*::*weight* ⇒ *'b* ⇒ *assn*
fixes *lessi*
assumes *GA*[*unfolded GEN-ALGO-def*, *sepref-fr-rules*]: *GEN-ALGO lessi* (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*)
unfolding *infty-less-eq-less*[*symmetric*] *infty-less-def*[*abs-def*]
by *sepref*
end
concrete-definition *infty-less-impl* **uses** *infty-less-impl.refine-raw* **is** (*uncurry ?impl*,-)∈-
lemmas [*sepref-fr-rules*] = *infty-less-impl.refine*

lemma *param-mpath'*: (*mpath'*,*mpath'*)
∈ ⟨⟨*A*×*r B* ×*r A*⟩*list-rel* ×*r B*⟩*option-rel* → ⟨⟨*A*×*r B* ×*r A*⟩*list-rel*⟩*option-rel*
proof −
have *1*: *mpath'* = *map-option fst*
apply (*intro ext*, *rename-tac x*)
apply (*case-tac x*)
apply *simp*
apply (*rename-tac a*)
apply (*case-tac a*)
apply *simp*
done
show *?thesis*
unfolding *1*
by *parametricity*
qed
lemmas (**in** −) [*sepref-import-param*] = *param-mpath'*

lemma *param-mpath-weight'*:
(*mpath-weight'*, *mpath-weight'*) ∈ ⟨⟨*A*×$_r$*B*×$_r$*A*⟩*list-rel* ×$_r$ *B*⟩*option-rel* → ⟨*B*⟩*infty-rel*
by (*auto elim*!: *option-relE simp*: *infty-rel-def top-infty-def*)

lemmas [*sepref-import-param*] = *param-mpath-weight'*

context *Dijkstra* **begin**
lemmas *impl-aux* = *mdijkstra-def*[*unfolded mdinit-def mpop-min-def mupdate-def*]

lemma *mdijkstra-correct*:
(*mdijkstra*, *SPEC* (*is-shortest-path-map v0*)) ∈ ⟨*br* α*r res-invarm*⟩*nres-rel*
proof −
note *mdijkstra-refines*
also note *dijkstra'-refines*
also note *dijkstra-correct*
finally show *?thesis*
by (*rule nres-relI*)
qed

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* ⇒ -

lemma *w-plus-param*: ((+), (+)::*'W*⇒-) ∈ *Id* → *Id* → *Id* **by** *simp*
lemma *w-less-param*: ((<), (<)::*'W*⇒-) ∈ *Id* → *Id* → *Id* **by** *simp*
lemmas [*sepref-import-param*] = *w-plus-param w-less-param*
lemma [*sepref-gen-algo-rules*]:
GEN-ALGO (*return oo* (+)) (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*)
by (*sep-auto simp*: *GEN-ALGO-def pure-def intro*!: *hfrefI hn-refineI*)+

lemma *conv-prio-pop-min*: *prio-pop-min m* = *do* {
ASSERT (*dom m* ≠ {});
((*k*,*v*),*m*) ← *mop-pm-pop-min id m*;
RETURN (*k*,*v*,*m*)
}
unfolding *prio-pop-min-def mop-pm-pop-min-def*
by (*auto simp*: *pw-eq-iff refine-pw-simps ran-def*)
end

context fixes N :: *nat* **and** *w-dummy*::$'W$::{*heap*,*weight*} **begin**

interpretation *Dijkstra-Impl w-dummy* .

definition *drmap-assn2* ≡ *IICF-Sepl-Binding.iam.assn*
(*pure* (*node-rel N*))
(*prod-assn*
(*list-assn* (*prod-assn* (*pure* (*node-rel N*)) (*prod-assn weight-assn* (*pure* (*node-rel N*)))))
weight-assn)

concrete-definition *mdijkstra′* **uses** *Dijkstra.impl-aux*

sepref-definition *dijkstra-imp* **is** *uncurry mdijkstra′*
:: (*is-graph N* (*Id*::($'W$×$'W$) *set*))^k ∗$_a$ (*pure* (*node-rel N*))^k →$_a$ *drmap-assn2*
unfolding *mdijkstra′-def*
apply (*subst conv-prio-pop-min*)
apply (*rewrite* **in** *RETURN* (-,¤) *iam.fold-custom-empty*)
apply (*rewrite hm-fold-custom-empty-sz*[*of N*])
apply (*rewrite* **in** -(- ↦ (¤,*0*)) *HOL-list.fold-custom-empty*)
unfolding *drmap-assn2-def*
using [[*id-debug*, *goals-limit* = *1*]]
by *sepref*

end

export-code *dijkstra-imp* **checking** *SML-imp*

The main correctness theorem

thm *Dijkstra.mdijkstra-correct*

lemma *mdijkstra′-aref*: (*uncurry mdijkstra′*,*uncurry* (*SPEC oo weighted-graph.is-shortest-path-map*))
∈ [λ(*G*,*v0*). *Dijkstra G v0*]$_f$ *Id*×$_r$*Id* → ⟨*br Dijkstra.*α*r Dijkstra.res-invarm*⟩*nres-rel*
using *Dijkstra.mdijkstra-correct*
by (*fastforce intro*!: *frefI simp*: *mdijkstra′.refine*[*symmetric*])

definition *drmap-assn N* ≡ *hr-comp* (*drmap-assn2 N*) (*br Dijkstra.*α*r 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*
∈ [λ(*G*, *v0*). *v0* ∈ *nodes G* ∧ (∀(*v*, *w*, *v′*) ∈ *edges G*. *0* ≤ *w*)]$_a$ (*is-graph N Id*)^k
∗$_a$ (*node-assn N*)^k → *drmap-assn N*
apply (*rule hfref-weaken-pre′*[*OF* - *dijkstra-imp.refine*[*FCOMP mdijkstra′-aref*]])
proof *clarsimp*
fix *G* :: (*nat*,$'w$::{*weight*,*heap*}) *graph* **and** *v0*
assume *v0-is-node*: *v0* ∈ *nodes G*

and *nonneg-weights*: $\forall (v, w, v') \in edges\ G.\ 0 \leq w$
and *v0<N*
and *RDOM*: *rdomp* (*is-graph N Id*) *G*

from *RDOM* **interpret** *valid-graph G* **unfolding** *is-graph-def rdomp-def* **by** *auto*

from *RDOM* **have** [*simp*]: *finite V* **unfolding** *is-graph-def rdomp-def* **by** *auto*

from *RDOM* **have** $\forall v{\in}V.\ \{(w, v').\ (v, w, v') \in E\} \in$
Range (⟨*Id* $\times_r$ *node-rel N*⟩*list-set-rel*)
by (*auto simp*: *succ-def is-graph-def rdomp-def*)
hence $\forall v{\in}V.$ *finite* $\{(w, v').\ (v, w, v') \in E\}$
unfolding *list-set-rel-range* **by** *simp*
hence *finite* (*Sigma V* ($\lambda v.\ \{(w, v').\ (v, w, v') \in E\}$))
by *auto*
also have $E \subseteq$ (*Sigma V* ($\lambda v.\ \{(w, v').\ (v, w, v') \in E\}$))
using *E-valid*
by *auto*
finally (*finite-subset*[*rotated*]) **have** [*simp*]: *finite E* **.**

show *Dijkstra G v0*
apply (*unfold-locales*)
unfolding *is-graph-def* **using** *v0-is-node nonneg-weights*
by *auto*
qed

end

corollary *dijkstra-imp-rule*:
<*is-graph n Id G Gi* $*$ $\uparrow$($v0 \in nodes\ G \wedge (\forall (v, w, v') \in edges\ G.\ 0 \leq w)$)>
dijkstra-imp n Gi v0
<λ*mi*. (*is-graph n Id*) *G Gi*
$*$ ($\exists_A m.$ *drmap-assn n m mi* $*$ $\uparrow$(*weighted-graph.is-shortest-path-map G v0 m*))
$>_t$
using *dijkstra-imp-correct*[*to-hnr*, *of v0 G n v0 Gi*]
unfolding *hn-refine-def*
apply (*clarsimp*)
apply (*erule cons-rule*[*rotated* −*1*])
apply (*sep-auto simp*: *hn-ctxt-def pure-def is-graph-def*)
apply (*sep-auto simp*: *hn-ctxt-def*)
done

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*)
by *simp-all*

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*
unfolding *red-witness-rel-def-internal*[*abs-def*] **by** (*simp add*: *relAPP-def*)

lemma *red-witness-rel-sv*[*constraint-rules*]:
single-valued R ⟹ *single-valued* (⟨*R*⟩*red-witness-rel*)
unfolding *red-witness-rel-def*
by *tagged-solver*

lemma [*sepref-fr-rules*]: *hn-refine*
(*hn-val R u u'* * *hn-val R v v'*)
(*return* (*red-init-witness u' v'*))
(*hn-val R u u'* * *hn-val R v v'*)
(*option-assn* (*pure* (⟨*R*⟩*red-witness-rel*)))
(*RETURN*$(*red-init-witness*$*u*$*v*))
apply *simp*
unfolding *red-init-witness-def*
apply *rule*
apply (*sep-auto simp*: *hn-ctxt-def pure-def red-witness-rel-def*)
done

lemma [*sepref-fr-rules*]: *hn-refine*
(*hn-val R u u'* * *hn-ctxt* (*option-assn* (*pure* (⟨*R*⟩*red-witness-rel*))) *w w'*)
(*return* (*prep-wit-red u' w'*))
(*hn-val R u u'* * *hn-ctxt* (*option-assn* (*pure* (⟨*R*⟩*red-witness-rel*))) *w w'*)
(*option-assn* (*pure* (⟨*R*⟩*red-witness-rel*)))
(*RETURN*$(*prep-wit-red*$*u*$*w*))
apply *rule*
apply (*cases w*)

apply (*sep-auto simp*: *hn-ctxt-def pure-def red-witness-rel-def*)
apply (*cases w'*)
apply (*sep-auto simp*: *hn-ctxt-def pure-def red-witness-rel-def*)
apply (*sep-auto simp*: *hn-ctxt-def pure-def red-witness-rel-def*)
done

term *red-dfs*

sepref-definition *red-dfs-impl* **is**
(*uncurry2* (*uncurry red-dfs*))
:: $(adjg\text{-}assn\ nat\text{-}assn)^k *_a (ias.assn\ nat\text{-}assn)^k *_a (ias.assn\ nat\text{-}assn)^d *_a nat\text{-}assn^k$
$\rightarrow_a$ *UNSPEC*
unfolding *red-dfs-def*[*abs-def*]
using [[*goals-limit* = *1*]]
by *sepref*
export-code *red-dfs-impl* **checking** *SML-imp*

declare *red-dfs-impl.refine*[*sepref-fr-rules*]

sepref-register *red-dfs* :: *'a i-graph* ⇒ *'a set* ⇒ *'a set* ⇒ *'a*
⇒ (*'a set* ∗ *'a i-red-witness option*) *nres*

lemma *id-init-wit-blue*[*id-rules*]:
init-wit-blue ::$_i$ *TYPE*(*'a* ⇒ *'a i-red-witness option* ⇒ *'a blue-witness*)
by *simp*

lemma *hn-blue-wit*[*sepref-import-param*]:
(*NO-CYC*,*NO-CYC*)∈*blue-wit-rel*
(*prep-wit-blue*,*prep-wit-blue*)∈*nat-rel*→*blue-wit-rel*→*blue-wit-rel*→*blue-wit-rel*
((=),(=))∈*blue-wit-rel*→*blue-wit-rel*→*bool-rel*
by *simp-all*

lemma *hn-init-wit-blue*[*sepref-fr-rules*]: *hn-refine*
(*hn-val nat-rel v v'* ∗ *hn-ctxt* (*option-assn* (*pure* (⟨*nat-rel*⟩*red-witness-rel*))) *w w'*)
(*return* (*init-wit-blue v' w'*))
(*hn-val nat-rel v v'* ∗ *hn-ctxt* (*option-assn* (*pure* (⟨*nat-rel*⟩*red-witness-rel*))) *w w'*)
(*pure blue-wit-rel*)
(*RETURN*$(*init-wit-blue*$*v*$*w*))
apply *rule*
apply (*sep-auto simp*: *hn-ctxt-def pure-def*)
apply (*case-tac w*, *sep-auto*)
apply (*case-tac w'*, *sep-auto*, *sep-auto simp*: *red-witness-rel-def*)
done

lemma *hn-extract-res*[*sepref-import-param*]:
(*extract-res*, *extract-res*) ∈ *blue-wit-rel* → *Id*
by *simp*

thm *red-dfs-impl.refine*

sepref-definition *blue-dfs-impl* **is** *uncurry2 blue-dfs* :: $((adjg\text{-}assn\ nat\text{-}assn)^k *_a (ias.assn\ nat\text{-}assn)^k *_a nat\text{-}assn^k \rightarrow_a id\text{-}assn)$
unfolding *blue-dfs-def*[*abs-def*]
apply (*rewrite* **in** *RECT* - ⌑ *ias.fold-custom-empty*)+
using [[*goals-limit* = *1*]]
by *sepref*
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* $(E^* ``\{v0\})]_f$ $((Id\times_r Id)\times_r Id) \rightarrow \langle Id\rangle nres\text{-}rel$
apply (*intro frefI nres-relI*)
unfolding *blue-dfs-spec-def* **apply** *clarsimp*
apply (*refine-vcg blue-dfs-correct*)
done

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*
)>$_t$
using *blue-dfs-impl-correct'*[*THEN hfrefD*, *THEN hn-refineD*, *of* ((*E*,*A*),*v0*) ((*succ-impl*,*A-impl*),*v0*), *simplified*]
apply (*rule cons-rule*[*rotated* −*1*])
using *assms*
by (*sep-auto simp*: *blue-dfs-spec-def pure-def*)+

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*)
by *simp*

sepref-definition *blue-dfs-impl-sz* **is** *uncurry2 blue-dfs* :: $((adjg\text{-}assn\ nat\text{-}assn)^k *_a (ias.assn\ nat\text{-}assn)^k *_a nat\text{-}assn^k \rightarrow_a id\text{-}assn)$
unfolding *blue-dfs-def*[*abs-def*]
apply (*rewrite* **in** *RECT* - ⌑ *testsuite-blue-dfs-modify*)
using [[*goals-limit* = *1*]]
by *sepref*

end

export-code *blue-dfs-impl-sz* **checking** *SML-imp*

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*
)>$_t$
using *blue-dfs-impl-sz-correct'*[*THEN hfrefD*, *THEN hn-refineD*, *of* ((*E*,*A*),*v0*) ((*succ-impl*,*A-impl*),*v0*), *simplified*]
apply (*rule cons-rule*[*rotated* −*1*])
using *assms*
by (*sep-auto simp*: *blue-dfs-spec-def pure-def*)+

end

5.5 Generic Worklist Algorithm with Subsumption

theory *Worklist-Subsumption*
imports *../Sepref*
begin

5.5.1 Utilities

definition *take-from-set* **where**

take-from-set s = ASSERT (s ≠ {}) ≫ 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})*
using *assms* **unfolding** *take-from-set-def* **by** *simp*

lemmas [*refine-vcg*] = *take-from-set-correct*[*THEN order.trans*]

definition *take-from-mset* **where**
take-from-mset s = 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#})*
using *assms* **unfolding** *take-from-mset-def* **by** *simp*

lemmas [*refine-vcg*] = *take-from-mset-correct*[*THEN order.trans*]

lemma *set-mset-mp*: *set-mset m ⊆ s ⟹ n < count m x ⟹ x∈s*
by (*meson count-greater-zero-iff le-less-trans subsetCE zero-le*)

lemma *pred-not-lt-is-zero*: (¬ *n − Suc 0 < n*) ⟷ *n=0* **by** *auto*

5.5.2 Search Spaces

A search space consists of a step relation, a start state, a final state predicate, and a subsumption preorder.

locale *Search-Space-Defs* =
fixes *E* :: *'a ⇒ 'a ⇒ bool* — Step relation
and *a₀* :: *'a* — Start state
and *F* :: *'a ⇒ bool* — Final states
and *subsumes* :: *'a ⇒ 'a ⇒ bool* (**infix** ‹≼› *50*) — Subsumption preorder
begin
definition *reachable* **where**
*reachable = E** a₀*

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*]: $a \preceq a$
and *trans*[*trans*]: $a \preceq b \Longrightarrow b \preceq c \Longrightarrow a \preceq c$

assumes *mono*: $a \preceq b \Longrightarrow E\ a\ a' \Longrightarrow$ *reachable a* $\Longrightarrow$ *reachable b* $\Longrightarrow \exists\ b'.\ E\ b\ b' \wedge a' \preceq b'$
and *F-mono*: $a \preceq a' \Longrightarrow F\ a \Longrightarrow F\ a'$
begin

lemma *start-reachable*[*intro*!, *simp*]:
reachable a_0
unfolding *reachable-def* **by** *simp*

lemma *step-reachable*:
assumes *reachable a E a* a'
shows *reachable* a'
using *assms* **unfolding** *reachable-def* **by** *simp*

lemma *finitely-branching*:
assumes *reachable a*
shows *finite* (*Collect* (*E a*))
by (*metis assms finite-reachable finite-subset mem-Collect-eq step-reachable subsetI*)

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* $a' \longrightarrow$ (∃ $b' \in$ *passed* ∪ *set-mset wait*. $a' \preceq b'$))

definition *start-subsumed passed wait* = (∃ *a* ∈ *passed* ∪ *set-mset wait*. $a_0 \preceq a$)

definition *worklist-inv* ≡ λ (*passed*, *wait*, *brk*).
passed ⊆ *Collect reachable* ∧
(*brk* $\longrightarrow$ (∃ *f*. *reachable f* ∧ *F f*)) ∧
(¬ *brk* $\longrightarrow$
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
  unfolding worklist-var-def by (auto simp: finite-reachable)

  context
```

begin

private lemma *aux1*:
assumes $\forall x{\in}passed.\ \neg\ a \preceq x$
and $passed \subseteq Collect\ reachable$
and $reachable\ a$
shows
$((insert\ a\ passed,\ wait',\ brk'),$
$passed,\ wait,\ brk)$
$\in worklist\text{-}var$
proof −
from *assms* **have** $a \notin passed$ **by** *auto*
with *assms*(*2*,*3*) **show** *?thesis*
by (*auto simp*: *worklist-inv-def worklist-var-def finite-psupset-def*)
qed

private lemma *aux2*:
assumes
$a' \in passed$
$a \preceq a'$
$a \in\# wait$
worklist-inv-frontier passed wait
shows *worklist-inv-frontier passed* $(wait - \{\#a\#\})$
using *assms* **unfolding** *worklist-inv-frontier-def*
using *trans*
apply *clarsimp*
by (*metis* (*no-types*, *lifting*) *Un-iff count-eq-zero-iff count-single mset-contains-eq mset-un-cases*)

private lemma *aux5*:
assumes
$a' \in passed$
$a \preceq a'$
$a \in\# wait$
start-subsumed passed wait
shows *start-subsumed passed* $(wait - \{\#a\#\})$
using *assms* **unfolding** *start-subsumed-def* **apply** *clarsimp*
by (*metis Un-iff insert-DiffM2 local.trans mset-right-cancel-elem*)

private lemma *aux3*:
assumes
$set\text{-}mset\ wait \subseteq Collect\ reachable$
$a \in\# wait$
$set\text{-}mset\ wait' = set\text{-}mset\ (wait - \{\#a\#\}) \cup Collect\ (E\ a)$
worklist-inv-frontier passed wait
shows *worklist-inv-frontier* $(insert\ a\ passed)\ wait'$
proof −
from *assms*(*1*,*2*) **have** *reachable a*
by (*simp add*: *subset-iff*)

with *finitely-branching* **have** [*simp*, *intro*!]: *finite* (*Collect* (*E a*)) **.**

from *assms*(*2*,*3*,*4*) **show** *?thesis* **unfolding** *worklist-inv-frontier-def*
by (*metis Un-iff insert-DiffM insert-iff local.refl mem-Collect-eq set-mset-add-mset-insert*)
qed

private lemma *aux6*:
assumes
a ∈# *wait*
start-subsumed passed wait
set-mset wait′ = *set-mset* (*wait* − {#*a*#}) ∪ *Collect* (*E a*)
shows *start-subsumed* (*insert a passed*) *wait′*
using *assms* **unfolding** *start-subsumed-def*
by (*metis Un-iff insert-DiffM insert-iff set-mset-add-mset-insert*)

lemma *aux4*:
assumes *worklist-inv-frontier passed* {#} *reachable x start-subsumed passed* {#}
passed ⊆ *Collect reachable*
shows ∃ *x′* ∈ *passed*. *x* ⪯ *x′*
proof −
from ‹*reachable x*› **have** *E*** *a*$_0$ *x* **by** (*simp add*: *reachable-def*)
from *assms*(*3*) **obtain** *b* **where** *a*$_0$ ⪯ *b b* ∈ *passed* **unfolding** *start-subsumed-def* **by** *auto*
have ∃ *x′*. ∃ *x″*. *E*** *b x′* ∧ *x* ⪯ *x′* ∧ *x′* ⪯ *x″* ∧ *x″* ∈ *passed* **if**
*E*** *a x a* ⪯ *b* *b* ⪯ *b′* *b′* ∈ *passed*
reachable a reachable b **for** *a b b′*
using *that* **proof** (*induction arbitrary*: *b b′ rule*: *converse-rtranclp-induct*)
case *base*
then show *?case* **by** *auto*
next
case (*step a a1 b b′*)
from ‹*E a a1*› ‹*a* ⪯ *b*› ‹*reachable a*› ‹*reachable b*› **obtain** *b1* **where**
E b b1 a1 ⪯ *b1*
using *mono* **by** *blast*
then obtain *b1′* **where** *E b′ b1′ b1* ⪯ *b1′* **using** *assms*(*4*) *mono step.prems* **by** *blast*
with ‹*b′* ∈ *passed*› *assms*(*1*) **obtain** *b1″* **where** *b1″* ∈ *passed b1′* ⪯ *b1″*
unfolding *worklist-inv-frontier-def* **by** *auto*
with ‹*b1* ⪯ -› **have** *b1* ⪯ *b1″* **using** *trans* **by** *blast*
with *step.IH*[*OF* ‹*a1* ⪯ *b1*› *this* ‹*b1″* ∈ *passed*›] ‹*reachable a*› ‹*E a a1*› ‹*reachable b*› ‹*E b b1*›
obtain *x′ x″* **where**
*E*** *b1 x′ x* ⪯ *x′ x′* ⪯ *x″ x″* ∈ *passed*
by (*auto intro*: *step-reachable*)
moreover from ‹*E b b1*› ‹*E*** *b1 x′*› **have** *E*** *b x′* **by** *auto*
ultimately show *?case* **by** *auto*
qed
from *this*[*OF* ‹*E*** *a*$_0$ *x*› ‹*a*$_0$ ⪯ *b*› *refl* ‹*b* ∈ -›] *assms*(*4*) ‹*b* ∈ *passed*› **show**

```
?thesis
    by (auto intro: trans)
  qed

  theorem worklist-algo-correct:
    worklist-algo ≤ SPEC (λ brk. brk ⟷ F-reachable)
  proof −
    note [simp] = size-Diff-submset pred-not-lt-is-zero
    note [dest] = set-mset-mp
    show ?thesis
    unfolding worklist-algo-def add-succ-spec-def F-reachable-def
      apply (refine-vcg wf-worklist-var)

      apply (auto; fail) []

     apply (auto simp: worklist-inv-def worklist-inv-frontier-def start-subsumed-def;
fail)

      apply (simp; fail)


        apply (auto simp: worklist-inv-def; fail)

        apply (auto simp: worklist-inv-def aux2 aux5
              dest: in-diffD
              split: if-split-asm; fail) []

        apply (auto simp: worklist-inv-def worklist-var-def intro: finite-subset[OF -
finite-reachable]; fail)



        apply (clarsimp split: if-split-asm)

          apply (clarsimp simp: worklist-inv-def; blast intro: step-reachable; fail)

      apply (auto
        simp: worklist-inv-def step-reachable aux3 aux6 finitely-branching
        dest: in-diffD; fail)[]

        apply (auto simp: worklist-inv-def aux1; fail)

      using F-mono apply (fastforce simp: worklist-inv-def dest!: aux4)
      done
  qed

  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'*)
    apply *simp*
    unfolding *add-succ-spec-def add-succ1-def*
    apply (*refine-vcg nfoldli-rule*[**where** *I* = λ*l1* - (*wait'*,*brk*). *if brk then* $\exists$ *a'*. *E a a'* $\wedge$ *F a' else set-mset wait'* = *set-mset wait* $\cup$ *set l1* $\wedge$ *set l1* $\cap$ *Collect F* = {}])
    apply (*auto*; *fail*)
    using *succs-correct*[*of a*] **apply** (*auto*; *fail*)
    using *succs-correct*[*of a*] **apply** (*auto*; *fail*)
    apply (*auto*; *fail*)
    using *succs-correct*[*of a*] **apply** (*auto*; *fail*)
    done

  definition *worklist-algo1* **where**
    worklist-algo1 = *do*
      {
        if F a_0 *then RETURN True*
        else do {
          let passed = {};
          let wait = {#a_0#};
          (*passed*, *wait*, *brk*) $\leftarrow$ *WHILEIT worklist-inv* (λ (*passed*, *wait*, *brk*). $\neg$ *brk* $\wedge$ *wait* $\neq$ {#})
            (λ (*passed*, *wait*, *brk*). *do*
              {
                (*a*, *wait*) $\leftarrow$ *take-from-mset wait*;
                if ($\exists$ *a'* $\in$ *passed*. *a* $\preceq$ *a'*) *then RETURN* (*passed*, *wait*, *brk*) *else do*
                  {
                    (*wait*,*brk*) $\leftarrow$ *add-succ1 wait a*;
                    let passed = *insert a passed*;
                    RETURN (*passed*, *wait*, *brk*)
                  }
              }

```
        )
        (passed, wait, False);
        RETURN brk
      }
    }
```

lemma *worklist-algo1-ref*[*refine*]: *worklist-algo1* $\leq \Downarrow Id$ *worklist-algo*
unfolding *worklist-algo1-def worklist-algo-def*
apply (*refine-rcg*)
apply *refine-dref-type*
unfolding *worklist-inv-def*
apply *auto*
done

end

end — Theory
theory *Worklist-Subsumption-Impl*
imports *../IICF/IICF Worklist-Subsumption*
begin

locale *Worklist2-Defs* = *Worklist1-Defs* +
fixes $A :: 'a \Rightarrow 'ai \Rightarrow assn$
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*
by *simp-all*

lemma *take-from-mset-as-mop-mset-pick*: *take-from-mset* = *mop-mset-pick*

apply (*intro ext*)
unfolding *take-from-mset-def*[*abs-def*]
by (*auto simp*: *pw-eq-iff refine-pw-simps*)

lemma [*safe-constraint-rules*]: *CN-FALSE is-pure A* $\Longrightarrow$ *is-pure A* **by** *simp*

sepref-thm *worklist-algo2* **is** *uncurry0 worklist-algo1* :: $(unit\text{-}assn^k \rightarrow_a bool\text{-}assn)$
unfolding *worklist-algo1-def add-succ1-def*
supply [[*goals-limit* = *1*]]
apply (*rewrite* **in** *Let* ⌑ - *lso-fold-custom-empty*)
apply (*rewrite* **in** {#a_0#} *lmso-fold-custom-empty*)
unfolding *take-from-mset-as-mop-mset-pick fold-lso-bex*
by *sepref*

end

concrete-definition *worklist-algo2*
for *Lei* a_0 *i Fi succsi*
uses *Worklist2.worklist-algo2.refine-raw* **is** (*uncurry0 ?f*,-)∈-
thm *worklist-algo2-def*

context *Worklist2* **begin**
lemma *Worklist2-this*: *Worklist2 E* a_0 *F* ($\preceq$) *succs A succsi* a_0 *i Fi Lei*
by *unfold-locales*

lemma *hnr-F-reachable*: (*uncurry0* (*worklist-algo2 Lei* a_0 *i Fi succsi*), *uncurry0* (*RETURN F-reachable*))
∈ $unit\text{-}assn^k \rightarrow_a bool\text{-}assn$
using *worklist-algo2.refine*[*OF Worklist2-this*,
FCOMP worklist-algo1-ref[*THEN nres-relI*],
FCOMP worklist-algo-correct[*THEN Id-SPEC-refine*, *THEN nres-relI*]]
by (*simp add*: *RETURN-def*)

end

context *Worklist1* **begin**
sepref-decl-op *F-reachable* :: *bool-rel* .
lemma [*def-pat-rules*]: *F-reachable* ≡ *op-F-reachable* **by** *simp*

lemma *hnr-op-F-reachable*:
assumes *GEN-ALGO* a_0 *i* (λa_0 *i*. (*uncurry0* a_0 *i*, *uncurry0* (*RETURN* a_0)) ∈ $unit\text{-}assn^k \rightarrow_a A$)
assumes *GEN-ALGO Fi* (λ*Fi*. (*Fi*,*RETURN o F*) ∈ $A^k \rightarrow_a bool\text{-}assn$)
assumes *GEN-ALGO Lei* (λ*Lei*. (*uncurry Lei*,*uncurry* (*RETURN oo* ($\preceq$))) ∈ $A^k *_a A^k \rightarrow_a bool\text{-}assn$)
assumes *GEN-ALGO succsi* (λ*succsi*. (*succsi*,*RETURN o succs*) ∈ $A^k \rightarrow_a$ *list-assn A*)
shows (*uncurry0* (*worklist-algo2 Lei* a_0 *i Fi succsi*), *uncurry0* (*RETURN*

(*PR-CONST op-F-reachable*)))
    ∈ *unit-assn*^{k} →$_a$ *bool-assn*
  proof −
    from *assms* **interpret** *Worklist2 E* a_0 *F* (≼) *succs A succsi* a_0 *i Fi Lei*
      by (*unfold-locales*; *simp add*: *GEN-ALGO-def*)

    from *hnr-F-reachable* **show** *?thesis* **by** *simp*
  qed

  sepref-decl-impl *hnr-op-F-reachable* .
end

end

5.6 Non-Recursive Algebraic Datatype

theory *Sepref-Snip-Datatype*
imports *../../IICF/IICF*
begin

We define a non-recursive datatype

datatype *'a enum* = *E1 'a* | *E2 'a* | *E3* | *E4 'a '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* ⟹ *is-pure* (*enum-assn A*)
  apply (*auto simp*: *is-pure-iff-pure-assn*)
  apply (*rename-tac x x'*)
  apply (*case-tac x*; *case-tac x'*; *simp add*: *pure-def*)
  done

An identitiy rule is required to easily prove trivial refinement theorems

lemma *enum-assn-id*[*simp*]: *enum-assn id-assn* = *id-assn*
  apply (*intro ext*)
  subgoal for *x y* **by** (*cases x*; *cases y*; *simp add*: *pure-def*)

done

Structural rules.

Without congruence condition

lemma *enum-match-nocong*: $[\![\bigwedge x\ y.\ \textit{hn-ctxt}\ A\ x\ y \Longrightarrow_t \textit{hn-ctxt}\ A'\ x\ y]\!] \Longrightarrow \textit{hn-ctxt}\ (\textit{enum-assn}\ A)\ e\ e' \Longrightarrow_t \textit{hn-ctxt}\ (\textit{enum-assn}\ A')\ e\ e'$
by (*cases e*; *cases e'*; *simp add*: *hn-ctxt-def entt-star-mono*)

lemma *enum-merge-nocong*:
assumes $\bigwedge x\ y.\ \textit{hn-ctxt}\ A\ x\ y \vee_A \textit{hn-ctxt}\ A'\ x\ y \Longrightarrow_A \textit{hn-ctxt}\ Am\ x\ y$
shows $\textit{hn-ctxt}\ (\textit{enum-assn}\ A)\ e\ e' \vee_A \textit{hn-ctxt}\ (\textit{enum-assn}\ A')\ e\ e' \Longrightarrow_A \textit{hn-ctxt}\ (\textit{enum-assn}\ Am)\ e\ e'$
using *assms*
by (*cases e*; *cases e'*; *simp add*: *hn-ctxt-def ent-disj-star-mono*)

With congruence condition

lemma *enum-match-cong*[*sepref-frame-match-rules*]:
$[\![\bigwedge x\ y.\ [\![x \in \textit{set-enum}\ e;\ y \in \textit{set-enum}\ e']\!] \Longrightarrow \textit{hn-ctxt}\ A\ x\ y \Longrightarrow_t \textit{hn-ctxt}\ A'\ x\ y]\!] \Longrightarrow \textit{hn-ctxt}\ (\textit{enum-assn}\ A)\ e\ e' \Longrightarrow_t \textit{hn-ctxt}\ (\textit{enum-assn}\ A')\ e\ e'$
by (*cases e*; *cases e'*; *simp add*: *hn-ctxt-def entt-star-mono*)

lemma *enum-merge-cong*[*sepref-frame-merge-rules*]:
assumes $\bigwedge x\ y.\ [\![x \in \textit{set-enum}\ e;\ y \in \textit{set-enum}\ e']\!] \Longrightarrow \textit{hn-ctxt}\ A\ x\ y \vee_A \textit{hn-ctxt}\ A'\ x\ y \Longrightarrow_t \textit{hn-ctxt}\ Am\ x\ y$
shows $\textit{hn-ctxt}\ (\textit{enum-assn}\ A)\ e\ e' \vee_A \textit{hn-ctxt}\ (\textit{enum-assn}\ A')\ e\ e' \Longrightarrow_t \textit{hn-ctxt}\ (\textit{enum-assn}\ Am)\ e\ e'$
apply (*blast intro*: *entt-disjE enum-match-cong entt-disjD1*[*OF assms*] *entt-disjD2*[*OF assms*])
done

Propagating invalid

lemma *entt-invalid-enum*: $\textit{hn-invalid}\ (\textit{enum-assn}\ A)\ e\ e' \Longrightarrow_t \textit{hn-ctxt}\ (\textit{enum-assn}\ (\textit{invalid-assn}\ A))\ e\ e'$
apply (*simp add*: *hn-ctxt-def invalid-assn-def*[*abs-def*])
apply (*rule enttI*)
apply *clarsimp*
apply (*cases e*; *cases e'*; *auto simp*: *mod-star-conv pure-def*)
done

lemmas *invalid-enum-merge*[*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*
by *sepref-to-hoare sep-auto*
lemma [*sepref-fr-rules*]: (*return o E2*,*RETURN o E2*) ∈ $A^d \to_a$ *enum-assn A*
by *sepref-to-hoare sep-auto*
lemma [*sepref-fr-rules*]: (*uncurry0* (*return E3*),*uncurry0* (*RETURN E3*)) ∈ *unit-assn*$^k \to_a$ *enum-assn A*
by *sepref-to-hoare sep-auto*
lemma [*sepref-fr-rules*]: (*uncurry* (*return oo E4*),*uncurry* (*RETURN oo E4*)) ∈ $A^d *_a A^d \to_a$ *enum-assn A*
by *sepref-to-hoare sep-auto*
lemma [*sepref-fr-rules*]: (*uncurry* (*return oo E5*),*uncurry* (*RETURN oo E5*)) ∈ *bool-assn*$^k *_a A^d \to_a$ *enum-assn A*
by *sepref-to-hoare* (*sep-auto simp*: *pure-def*)

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*
by *simp*

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*) **by** *simp*

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*$)$
apply (*rule eq-reflection*)
by (*simp split*: *enum.splits*)

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*
by (*simp add*: *hn-ctxt-def*)

The cases lemma first extracts the refinement for the datatype from the precondition. Next, it generate proof obligations to refine the functions for every case. Finally the postconditions of the refinement are merged.

Note that we handle the destructed values separately, to allow reconstruction of the original datatype after the case-expression.

Moreover, we provide (invalidated) versions of the original compound value to the cases, which allows access to pure compound values from inside the case.

lemma *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*)
apply (*rule hn-refine-cons-pre*[*OF FR*])
apply1 *extract-hnr-invalids*
apply (*cases e*; *cases e′*; *simp add*: *enum-assn.simps*[*THEN enum-assn-ctxt*])
subgoal
apply (*rule hn-refine-cons*[*OF - E1 - entt-refl*]; *assumption?*)
applyS (*simp add*: *hn-ctxt-def*) — Match precondition for case, get *enum-assn* from assumption generated by *extract-hnr-invalids*
apply (*rule entt-star-mono*) — Split postcondition into pairs for compounds and frame, drop *hn-ctxt XX*

apply1 (*rule entt-fr-drop*)
apply1 (*rule entt-trans*[*OF - MERGE1*])
applyS (*simp add*: *hn-ctxt-def entt-disjI1′ entt-disjI2′*)
apply1 (*rule entt-trans*[*OF - MERGE2*])
applyS (*simp add*: *entt-disjI1′ entt-disjI2′*)
done
subgoal
apply (*rule hn-refine-cons*[*OF - E2 - entt-refl*]; *assumption?*)
applyS (*simp add*: *hn-ctxt-def*)
apply (*rule entt-star-mono*)
apply1 (*rule entt-fr-drop*)
apply1 (*rule entt-trans*[*OF - MERGE1*])
applyS (*simp add*: *hn-ctxt-def entt-disjI1′ entt-disjI2′*)
apply1 (*rule entt-trans*[*OF - MERGE2*])
applyS (*simp add*: *entt-disjI1′ entt-disjI2′*)
done
subgoal
apply (*rule hn-refine-cons*[*OF - E3 - entt-refl*]; *assumption?*)
applyS (*simp add*: *hn-ctxt-def*)
apply (*subst mult.commute*, *rule entt-fr-drop*)
apply (*rule entt-trans*[*OF - MERGE2*])
apply (*simp add*: *entt-disjI1′ entt-disjI2′*)
done
subgoal
apply (*rule hn-refine-cons*[*OF - E4 - entt-refl*]; *assumption?*)
applyS (*simp add*: *hn-ctxt-def*)
apply (*rule entt-star-mono*)
apply1 (*rule entt-fr-drop*)
apply (*rule entt-star-mono*)

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′*)
done
subgoal
apply (*rule hn-refine-cons*[*OF - E5 - entt-refl*]; *assumption?*)
applyS (*simp add*: *hn-ctxt-def*)
apply (*rule entt-star-mono*)
apply1 (*rule entt-fr-drop*)
apply (*rule entt-star-mono*)

apply1 (*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')
  done
done
```

After some more preprocessing (adding extra frame-rules for non-atomic postconditions, and splitting the merge-terms into binary merges), this rule can be registered

```
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
  unfolding test1-def[abs-def] by sepref
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* :: $unit\text{-}assn^k \rightarrow_a nat\text{-}assn$
unfolding *test-def*
supply [[*goals-limit=1*]]
by *sepref*

end

5.7 Snippet to Define Custom Combinators

theory *Sepref-Snip-Combinator*
imports *../../IICF/IICF*
begin

5.7.1 Definition of the Combinator

Currently, when defining new combinators, you are largely on your own. If you can show your combinator equivalent to some other, already existing, combinator, you should apply this equivalence in the monadify phase.
In this example, we show the development of a map combinator from scratch.

We set ourselves in to a context where we fix the abstract and concrete arguments of the monadic map combinator, as well as the refinement assertions, and a frame, that represents the remaining heap content, and may be read by the map-function.

context
fixes *f* :: *'a ⇒ 'b nres*
fixes *l* :: *'a list*

fixes *fi* :: *'ai ⇒ 'bi Heap*
fixes *li* :: *'ai list*

fixes *A A'* :: *'a ⇒ 'ai ⇒ assn* — Refinement for list elements before and after map-function. Different, as map function may invalidate list elements!

fixes $B :: 'b \Rightarrow 'bi \Rightarrow assn$

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-ctxt A x xi* $*$ *F*) (*fi xi*) (*hn-ctxt A'* *x xi* $*$ *F*) *B* (*f*$*x*)
— Refinement for *f*

begin

We implement our combinator using the monadic refinement framework.

definition *mmap* $\equiv$ *RECT* (λ*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*) $\equiv$ *F*
lemma *F-unf*: *hn-ctxt F-assn x y* = *F*
by (*auto simp*: *F-assn-def hn-ctxt-def*)

We build a combinator rule to refine *f*. We need a combinator rule here, because *f* does not only depend on its formal arguments, but also on the frame (represented as dummy argument).

lemma *f-rl'*: *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*)
unfolding *F-unf* **by** (*rule f-rl*)

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*
unfolding *mmap-def HOL-list.fold-custom-empty*
apply *sepref-dbg-keep*
done

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*]: $mmap \equiv \lambda_2 f\ l.\ SP\ mmap\$(\lambda_2 x.\ f\$x)\$l$ **by** *simp*
lemma *mmap-mcomb*[*sepref-monadify-comb*]: $mmap\$f\$x \equiv (\gg\!=)\$(EVAL\$x)\$(\lambda_2 x.\ SP\ mmap\$f\$x)$ **by** *simp*

We can massage the refinement theorem ($\bigwedge x\ xi.$ *hn-refine* (*hn-ctxt ?A x xi* $*$ *?F*) (*?fi xi*) (*hn-ctxt ?A′ x xi* $*$ *?F*) *?B* (*?f* $ *x*)) $\Longrightarrow$ *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 $\bigwedge x\ xi.$ *hn-refine* (*hn-ctxt A x xi* $*$ *F*) (*fi xi*) (*Q x xi*) *B* (*f x*)
— Refinement of map-function
and $\bigwedge x\ xi.$ *Q x xi* $\Longrightarrow_t$ *hn-ctxt A′ x xi* $*$ *F*
— Recover refinement for list-element and original frame from what map-function produced
shows *hn-refine P* (*mmap-impl fi li*) (*hn-ctxt* (*list-assn A′*) *l li* $*$ *F*) (*list-assn B*) ($mmap\$(\lambda_2 x.\ f\ x)\$l$)
unfolding *APP-def PROTECT2-def*
using *hn-refine-cons-pre*[*OF - mmap-impl.refine*, *sepref-prep-comb-rule*, *of P A l li F fi Q B f A′*]
using *assms*
by *simp*

5.7.4 Example

Finally, we can test our combinator. Note how the map-function accesses the array on the heap, which is not among its arguments. This is only possible as we passed around a frame.

sepref-thm *test-mmap*
is $\lambda l.$ *do* { *let a* = *op-array-of-list* [*True*,*True*,*False*]; *mmap* ($\lambda x.$ *do* { *mop-list-get a* (*x mod 3*) }) *l* }

:: $(\textit{list-assn nat-assn})^k \rightarrow_a \textit{list-assn bool-assn}$
unfolding *HOL-list.fold-custom-empty*
by *sepref*

5.7.5 Limitations

Currently, the major limitation is that combinator rules are fixed to specific data types. In our example, we did an implementation for HOL lists. We cannot come up with an alternative implementation, for, e.g., array-lists, but have to use a different abstract combinator.

One workaround is to use some generic operations, as is done for foreach-loops, which require a generic to-list operation. However, in this case, we produce unwanted intermediate lists, and would have to add complicated a-posteriori deforestation optimizations.

end

Chapter 6

Benchmarks

Contains the benchmarks of the IRF/IICF. See the README file in the benchmark folder for more information on how to run the benchmarks.

```
theory Heapmap-Bench
imports
  ../../../IICF/Impl/Heaps/IICF-Impl-Heapmap
  ../../../Sepref-ICF-Bindings
begin

context
  includes bit-operations-syntax
begin

definition rrand :: uint32 ⇒ uint32
  where rrand s ≡ (s * 1103515245 + 12345) AND 0x7FFFFFFF

end

definition rand :: uint32 ⇒ nat ⇒ (uint32 * nat) where
  rand s m ≡ let
    s = rrand s;
    r = nat-of-uint32 s;
    r = (r * m) div 0x80000000
  in (s,r)

partial-function (heap) rep where rep i N f s = (
  if i<N then do {
    s ← f s i;
    rep (i+1) N f s
  } else return s
)

declare rep.simps[code]

term hm-insert-op-impl
```

```
definition testsuite N ≡ do {
  let s=0;
  let N2=efficient-nat-div2 N;
  hm ← hm-empty-op-impl N;

  (hm,s) ← rep 0 N (λ(hm,s) i. do {
    let (s,v) = rand s N2;
    hm ← hm-insert-op-impl N id i v hm;
    return (hm,s)
  }) (hm,s);


  (hm,s) ← rep 0 N (λ(hm,s) i. do {
    let (s,v) = rand s N2;
    hm ← hm-change-key-op-impl id i v hm;
    return (hm,s)
  }) (hm,s);


  hm ← rep 0 N (λhm i. do {
    (-,hm) ← hm-pop-min-op-impl id hm;
    return hm
  }) hm;


  return ()
}

export-code rep in SML-imp

partial-function (tailrec) drep where drep i N f s = (
  if i<N then drep (i+1) N f (f s i)
  else s
)

declare drep.simps[code]


term aluprioi.insert
term aluprioi.empty
term aluprioi.pop

definition ftestsuite N ≡ do {
  let s=0;
  let N2=efficient-nat-div2 N;
  let hm= aluprioi.empty ();

  let (hm,s) = drep 0 N (λ(hm,s) i. do {
```

```
    let (s,v) = rand s N2;
    let hm = aluprioi.insert hm i v;
    (hm,s)
  }) (hm,s);

  let (hm,s) = drep 0 N (λ(hm,s) i. do {
    let (s,v) = rand s N2;
    let hm = aluprioi.insert hm i v;
    (hm,s)
  }) (hm,s);

  let hm = drep 0 N (λhm i. do {
    let (-,-,hm) = aluprioi.pop hm;
    hm
  }) hm;

  ()
}


export-code
  testsuite ftestsuite
  nat-of-integer integer-of-nat
  in SML-imp module-name Heapmap
  file ‹heapmap-export.sml›

end
theory Dijkstra-Benchmark
imports ../../../Examples/Sepref-Dijkstra
  Dijkstra-Shortest-Path.Test
begin

definition nat-cr-graph-imp
  :: nat ⇒ (nat × nat × nat) list ⇒ nat graph-impl Heap
  where nat-cr-graph-imp ≡ cr-graph

concrete-definition nat-dijkstra-imp uses dijkstra-imp-def[where 'W=nat]
prepare-code-thms nat-dijkstra-imp-def

lemma nat-dijkstra-imp-eq: nat-dijkstra-imp = dijkstra-imp
  unfolding dijkstra-imp-def[abs-def] nat-dijkstra-imp-def[abs-def]
  by simp



definition nat-cr-graph-fun nn es ≡ hlg-from-list-nat ([0..<nn], es)

export-code
```

integer-of-nat nat-of-integer

ran-graph

nat-cr-graph-fun nat-dijkstra

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*
unfolding *succ-of-list-def*[*abs-def*]
apply (*autoref* (*keep-goal*))
done

concrete-definition *succ-of-list-impl* **uses** *succ-of-list-impl*

schematic-goal *acc-of-list-impl*:
notes [*autoref-tyrel*] =
ty-REL[**where** *'a*=*nat set* **and** *R*=⟨*nat-rel*⟩*dflt-rs-rel* **for** *R*]

shows (*?f*::*?'c*,*acc-of-list*) ∈ *?R*
unfolding *acc-of-list-def*[*abs-def*]
apply (*autoref* (*keep-goal*))
done

concrete-definition *acc-of-list-impl* **uses** *acc-of-list-impl*

schematic-goal *red-dfs-impl-refine-aux*:

fixes u'::*nat* **and** V'::*nat set*
notes [*autoref-tyrel*] =
ty-REL[**where** *'a*=*nat set* **and** R=⟨*nat-rel*⟩*dflt-rs-rel*]
assumes [*autoref-rules*]:
(u,u')∈*nat-rel*
(V,V')∈⟨*nat-rel*⟩*dflt-rs-rel*
(*onstack*,*onstack'*)∈⟨*nat-rel*⟩*dflt-rs-rel*
(E,E')∈⟨*nat-rel*⟩*slg-rel*
shows (*RETURN* (*?f*::*?'c*), *red-dfs E' onstack' V' u'*) ∈ *?R*
apply −
unfolding *red-dfs-def*
apply (*autoref-monadic*)
done

concrete-definition *red-dfs-impl* **uses** *red-dfs-impl-refine-aux*
prepare-code-thms *red-dfs-impl-def*
declare *red-dfs-impl.refine*[*autoref-higher-order-rule*, *autoref-rules*]

schematic-goal *ndfs-impl-refine-aux*:
fixes *s*::*nat* **and** *succi*
notes [*autoref-tyrel*] =
ty-REL[**where** *'a*=*nat set* **and** R=⟨*nat-rel*⟩*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*
unfolding *blue-dfs-def*
apply (*autoref-monadic* (*trace*))
done

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* **.**

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*
unfolding *succ-of-list-def*[*abs-def*]
apply (*autoref* (*keep-goal*))
done

concrete-definition *succ-of-list-impl* **uses** *succ-of-list-impl*

schematic-goal *acc-of-list-impl*:
notes [*autoref-tyrel*] =
ty-REL[**where** *'a=nat set* **and** *R=⟨nat-rel⟩iam-set-rel* **for** *R*]

shows (*?f::?'c,acc-of-list*) ∈ *?R*
unfolding *acc-of-list-def*[*abs-def*]
apply (*autoref* (*keep-goal*))
done

concrete-definition *acc-of-list-impl* **uses** *acc-of-list-impl*

schematic-goal *red-dfs-impl-refine-aux*:

fixes *u'*::*nat* **and** *V'*::*nat set*
notes [*autoref-tyrel*] =
ty-REL[**where** *'a=nat set* **and** *R=⟨nat-rel⟩iam-set-rel*]
assumes [*autoref-rules*]:
(*u,u'*)∈*nat-rel*
(*V,V'*)∈⟨*nat-rel*⟩*iam-set-rel*
(*onstack,onstack'*)∈⟨*nat-rel*⟩*iam-set-rel*
(*E,E'*)∈⟨*nat-rel*⟩*slg-rel*
shows (*RETURN* (*?f::?'c*), *red-dfs E' onstack' V' u'*) ∈ *?R*
apply −
unfolding *red-dfs-def*
apply (*autoref-monadic*)
done

concrete-definition *red-dfs-impl* **uses** *red-dfs-impl-refine-aux*
prepare-code-thms *red-dfs-impl-def*
declare *red-dfs-impl.refine*[*autoref-higher-order-rule*, *autoref-rules*]

schematic-goal *ndfs-impl-refine-aux*:
fixes *s*::*nat* **and** *succi*
notes [*autoref-tyrel*] =
ty-REL[**where** *'a=nat set* **and** *R=⟨nat-rel⟩iam-set-rel*]
assumes [*autoref-rules*]:
(*succi,E*)∈⟨*nat-rel*⟩*slg-rel*
(*Ai,A*)∈⟨*nat-rel*⟩*iam-set-rel*

notes [*autoref-rules*] = *IdI*[*of s*]
shows (*RETURN* (*?f*::*?'c*), *blue-dfs E A s*) ∈ ⟨*?R*⟩*nres-rel*
unfolding *blue-dfs-def*
apply (*autoref-monadic* (*trace*))
done

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* **.**

definition *imp-ndfs-impl* ≡ *blue-dfs-impl*
definition *imp-ndfs-sz-impl* ≡ *blue-dfs-impl-sz*
definition *imp-acc-of-list l* ≡ *From-List-GA.ias-from-list* (*map nat-of-integer l*)
definition *imp-graph-of-list n l* ≡ *cr-graph* (*nat-of-integer n*) (*map* (*pairself nat-of-integer*) *l*)

export-code
nat-of-integer integer-of-nat
fun.fun-ndfs-impl fun.fun-succ-of-list fun.fun-acc-of-list
funs.funs-ndfs-impl funs.funs-succ-of-list funs.funs-acc-of-list
imp-ndfs-impl imp-ndfs-sz-impl imp-acc-of-list imp-graph-of-list
in *SML-imp* **module-name** *NDFS-Benchmark* **file** ‹*NDFS-Benchmark-export.sml*›

ML-val ‹*open Time*›
end