

A Separation Logic Framework for Imperative HOL

Peter Lammich and Rene Meis

May 26, 2024

Abstract

We provide a framework for separation-logic based correctness proofs of Imperative HOL programs. Our framework comes with a set of proof methods to automate canonical tasks such as verification condition generation and frame inference. Moreover, we provide a set of examples that show the applicability of our framework. The examples include algorithms on lists, hash-tables, and union-find trees. We also provide abstract interfaces for lists, maps, and sets, that allow to develop generic imperative algorithms and use data-refinement techniques.

As we target Imperative HOL, our programs can be translated to efficiently executable code in various target languages, including ML, OCaml, Haskell, and Scala.

Contents

1	Introduction	4
2	Exception-Aware Relational Framework	5
	2.0.1 Link with <i>effect</i> and <i>success</i>	6
	2.0.2 Elimination Rules for Basic Combinators	7
2.1	Array Commands	8
2.2	Reference Commands	10
3	Assertions	10
3.1	Partial Heaps	10
3.2	Assertions	12
	3.2.1 Empty Partial Heap	13
3.3	Connectives	14
	3.3.1 Empty Heap and Separation Conjunction	14
	3.3.2 Magic Wand	15
	3.3.3 Boolean Algebra on Assertions	16
	3.3.4 Existential Quantification	17
	3.3.5 Pure Assertions	17
	3.3.6 Pointers	19

3.4	Properties of the Models-Predicate	19
3.5	Entailment	21
3.5.1	Properties	21
3.5.2	Weak Entails	23
3.6	Precision	24
4	Hoare-Triples	25
4.1	Definition	26
4.2	Rules	27
4.2.1	Basic Rules	27
4.2.2	Rules for Atomic Commands	28
4.2.3	Rules for Composed Commands	29
5	Automation	31
5.1	Normalization of Assertions	31
5.1.1	Simplifier Setup Fine-Tuning	32
5.2	Normalization of Entailments	33
5.3	Frame Matcher	33
5.4	Frame Inference	35
5.5	Entailment Solver	35
5.6	Verification Condition Generator	35
5.7	ML-setup	36
5.8	Semi-Automatic Reasoning	36
5.8.1	Manual Frame Inference	38
5.9	Quick Overview of Proof Methods	38
6	Separation Logic Framework Entrypoint	40
7	Interface for Lists	40
8	Singly Linked List Segments	41
8.1	Nodes	42
8.2	List Segment Assertion	42
8.3	Lemmas	43
8.3.1	Concatenation	43
8.3.2	Splitting	43
8.3.3	Precision	43
9	Open Singly Linked Lists	44
9.1	Definitions	44
9.2	Precision	44
9.3	Operations	44
9.3.1	Allocate Empty List	44
9.3.2	Emptiness check	44
9.3.3	Prepend	45

9.3.4	Pop	45
9.3.5	Reverse	45
9.3.6	Remove	46
9.3.7	Iterator	47
9.3.8	List-Sum	47
10	Circular Singly Linked Lists	48
10.1	Datatype Definition	48
10.2	Precision	49
10.3	Operations	49
10.3.1	Allocate Empty List	49
10.3.2	Prepend Element	49
10.3.3	Append Element	50
10.3.4	Pop First Element	50
10.3.5	Rotate	51
10.4	Test	51
11	Interface for Maps	52
12	Hash-Tables	53
12.1	Datatype	54
12.1.1	Definition	54
12.1.2	Storable on Heap	54
12.2	Assertions	54
12.2.1	Assertion for Hashtable	54
12.3	New	55
12.3.1	Definition	55
12.3.2	Complete Correctness	55
12.4	Lookup	56
12.4.1	Definition	56
12.4.2	Complete Correctness	56
12.5	Update	57
12.5.1	Definition	57
12.5.2	Complete Correctness	58
12.6	Delete	59
12.6.1	Definition	59
12.6.2	Complete Correctness	60
12.7	Re-Hashing	61
12.7.1	Auxiliary Functions	61
12.8	Conversion to List	64

13 Hash-Maps	64
13.1 Auxiliary Lemmas	64
13.2 Main Definitions and Lemmas	65
13.3 Iterators	72
13.3.1 Definitions	72
13.3.2 Auxiliary Lemmas	73
13.3.3 Main Lemmas	74
14 Hash-Maps (Interface Instantiations)	75
15 Interface for Sets	76
16 Hash-Sets	78
16.1 Auxiliary Definitions	79
16.2 Main Definitions	79
17 Generic Algorithm to Convert Sets to Lists	81
17.1 Algorithm	81
17.2 Sample Instantiation for hash set and open list	83
18 Union-Find Data-Structure	83
18.1 Abstract Union-Find on Lists	83
18.1.1 Representatives	83
18.1.2 Abstraction to Partial Equivalence Relation	85
18.1.3 Operations	85
18.2 Implementation with Imperative/HOL	86
19 Common Proof Methods and Idioms	89
19.1 The Method <code>sep_auto</code>	89
19.2 Applying Single Rules	90
19.3 Functions with Explicit Recursion	90
19.4 Functions with Recursion Involving the Heap	91
19.5 Precision Proofs	91
20 Conclusion	92

1 Introduction

We provide a separation logic framework for Imperative/HOL.

Imperative/HOL [3] is a framework for imperative monadic programs in Isabelle/HOL. It allows to combine imperative and functional concepts, and supports generation of efficient, verified code in various target languages, including SML, OCaml, Haskell, and Scala. Thus, it is the ideal platform for writing verified, efficient algorithms. However, it only has rudimentary

support for proving programs correct. We close this gap by providing a separation logic [7] for total correctness, and tools to automate canonical tasks, such as a verification condition generator, a frame inference method, and a set of simprocs for assertions. We test the applicability of our framework by formalizing various data structures, such as linked lists, hash-tables and union-find trees. Moreover, we provide abstract interfaces for lists, maps, and sets in the style of the Isabelle Collection Framework [5]. They allow to write generic imperative algorithms and use data refinement techniques.

Related Work This work is based on the diploma thesis of Rene Meis [6], that contains a preliminary version of the framework.

Independently of us, Klein et. al. [4] formalized a general separation algebra framework in Isabelle/HOL. It also contains a frame-inference algorithm, and is intended to be instantiated to various target languages. However, due to technical issues, we cannot use this framework, as it would require to change the formal foundation of Imperative/HOL, such that partial heaps are properly supported.

Recently several formalizations of separation logic in theorem provers were published. Jesper et. al. [1] formalized separation logic in Coq for object-oriented programs. Tuerk [8] formalized and extended smallfoot [2] in his PhD thesis in HOL4. These approaches are based on a deeply embedded programming and assertion language with a fixed finite set of constructs.

Organization of the Entry This entry consists of two parts, the main separation logic framework, and a bunch of examples. The theory *Sep-Main* is the entry point for the framework. The examples are contained in the *Examples*-subdirectory. They serve as documentation and to show the applicability of the framework. Moreover, the *Tools*-subdirectory contains some general prerequisites.

Documentation The methods provided by the framework are documented in Section 5.9. Moreover, Section 19 contains some heavily documented examples that show common idioms for using the framework.

2 Exception-Aware Relational Framework

```
theory Run
imports "HOL-Imperative_HOL.Imperative_HOL"
begin
```

With Imperative HOL comes a relational framework. However, this can only be used if exception freeness is already assumed. This results in some proof

duplication, because exception freeness and correctness need to be shown separately.

In this theory, we develop a relational framework that is aware of exceptions, and makes it possible to show correctness and exception freeness in one run.

There are two types of states:

1. A normal (Some) state contains the current heap.
2. An exception state is None

The two states exactly correspond to the option monad in Imperative HOL.

```
type_synonym state = "Heap.heap option"
```

```
primrec is_exn where
  "is_exn (Some _) = False" |
  "is_exn None = True"
```

```
primrec the_state where
  "the_state (Some h) = h"
```

— The exception-aware, relational semantics

```
inductive run :: "'a Heap  $\Rightarrow$  state  $\Rightarrow$  state  $\Rightarrow$  'a  $\Rightarrow$  bool" where
  push_exn: "is_exn  $\sigma \Longrightarrow$  run c  $\sigma \sigma$  r" |
  new_exn: "[ $\neg$  is_exn  $\sigma$ ; execute c (the_state  $\sigma$ ) = None]
 $\Longrightarrow$  run c  $\sigma$  None r" |
  regular: "[ $\neg$  is_exn  $\sigma$ ; execute c (the_state  $\sigma$ ) = Some (r, h')]
 $\Longrightarrow$  run c  $\sigma$  (Some h') r"
```

2.0.1 Link with effect and success

```
lemma run_effectE:
  assumes "run c  $\sigma \sigma'$  r"
  assumes " $\neg$ is_exn  $\sigma'$ "
  obtains h h' where
    " $\sigma$ =Some h" " $\sigma' =$  Some h'"
    "effect c h h' r"
  <proof>
```

```
lemma run_effectI:
  assumes "run c (Some h) (Some h') r"
  shows "effect c h h' r"
  <proof>
```

```
lemma effect_run:
  assumes "effect c h h' r"
  shows "run c (Some h) (Some h') r"
```

<proof>

lemma success_run:

assumes "success f h"

obtains h' r where "run f (Some h) (Some h') r"

<proof>

run always yields a result

lemma run_complete:

obtains σ' r where "run c σ σ' r"

<proof>

lemma run_detE:

assumes "run c σ σ' r" "run c σ τ s"

" \neg is_exn σ "

obtains "is_exn σ' " " $\sigma' = \tau$ " | " \neg is_exn σ' " " $\sigma' = \tau$ " "r = s"

<proof>

lemma run_detI:

assumes "run c (Some h) (Some h') r" "run c (Some h) σ s"

shows " $\sigma = \text{Some h}' \wedge r = s$ "

<proof>

lemma run_exn:

assumes "run f σ σ' r"

"is_exn σ "

obtains " $\sigma' = \sigma$ "

<proof>

2.0.2 Elimination Rules for Basic Combinators

named_theorems run_elims "elimination rules for run"

lemma runE[run_elims]:

assumes "run (f \ggg g) σ σ'' r"

obtains σ' r' where

"run f σ σ' r'"

"run (g r') σ' σ'' r"

<proof>

lemma runE'[run_elims]:

assumes "run (f \ggg g) σ σ'' res"

obtains σt rt where

"run f σ σt rt"

"run g σt σ'' res"

<proof>

lemma run_return[run_elims]:

assumes "run (return x) σ σ' r"
obtains "r = x" " $\sigma' = \sigma$ " " \neg is_exn σ " | " $\sigma = \text{None}$ "
 <proof>

lemma run_raise_iff: "run (raise s) σ σ' r \longleftrightarrow ($\sigma' = \text{None}$)"
 <proof>

lemma run_raise[run_elims]:
assumes "run (raise s) σ σ' r"
obtains " $\sigma' = \text{None}$ "
 <proof>

lemma run_raiseI:
 "run (raise s) σ None r" <proof>

lemma run_if[run_elims]:
assumes "run (if c then t else e) h h' r"
obtains "c" "run t h h' r"
 | " \neg c" "run e h h' r"
 <proof>

lemma run_case_option[run_elims]:
assumes "run (case x of None \Rightarrow n | Some y \Rightarrow s y) σ σ' r"
 " \neg is_exn σ "
obtains "x = None" "run n σ σ' r"
 | y where "x = Some y" "run (s y) σ σ' r"
 <proof>

lemma run_heap[run_elims]:
assumes "run (Heap_Monad.heap f) σ σ' res"
 " \neg is_exn σ "
obtains " $\sigma' = \text{Some}$ (snd (f (the_state σ)))"
and "res = (fst (f (the_state σ)))"
 <proof>

2.1 Array Commands

lemma run_length[run_elims]:
assumes "run (Array.len a) σ σ' r"
 " \neg is_exn σ "
obtains " \neg is_exn σ " " $\sigma' = \sigma$ " "r = Array.length (the_state σ) a"
 <proof>

lemma run_new_array[run_elims]:
assumes "run (Array.new n x) σ σ' r"
 " \neg is_exn σ "
obtains " $\sigma' = \text{Some}$ (snd (Array.alloc (replicate n x) (the_state σ)))"


```

and "r = fst (Array.alloc (replicate n x) (the_state  $\sigma$ ))"
and "Array.get (the_state  $\sigma'$ ) r = replicate n x"
⟨proof⟩

lemma run_make[run_elims]:
  assumes "run (Array.make n f)  $\sigma$   $\sigma'$  r"
          "¬is_exn  $\sigma$ "
  obtains " $\sigma' = \text{Some (snd (Array.alloc (map f [0 ..<n]) (the\_state \sigma)))}$ "
          "r = fst (Array.alloc (map f [0 ..<n]) (the\_state \sigma))"
          "Array.get (the\_state \sigma') r = (map f [0 ..<n])"
  ⟨proof⟩

lemma run_upd[run_elims]:
  assumes "run (Array.upd i x a)  $\sigma$   $\sigma'$  res"
          "¬is_exn  $\sigma$ "
  obtains " $\neg i < \text{Array.length (the\_state \sigma)}$  a"
          " $\sigma' = \text{None}$ "
  |
          "i < \text{Array.length (the\_state \sigma)} a"
          " $\sigma' = \text{Some (Array.update a i x (the\_state \sigma))}$ "
          "res = a"
  ⟨proof⟩

lemma run_nth[run_elims]:
  assumes "run (Array.nth a i)  $\sigma$   $\sigma'$  r"
          "¬is_exn  $\sigma$ "
  obtains "¬is_exn  $\sigma$ "
          "i < \text{Array.length (the\_state \sigma)} a"
          "r = (Array.get (the\_state \sigma) a) ! i"
          " $\sigma' = \sigma$ "
  |
          "¬ i < \text{Array.length (the\_state \sigma)} a"
          " $\sigma' = \text{None}$ "
  ⟨proof⟩

lemma run_of_list[run_elims]:
  assumes "run (Array.of_list xs)  $\sigma$   $\sigma'$  r"
          "¬is_exn  $\sigma$ "
  obtains " $\sigma' = \text{Some (snd (Array.alloc xs (the\_state \sigma)))}$ "
          "r = fst (Array.alloc xs (the\_state \sigma))"
          "Array.get (the\_state \sigma') r = xs"
  ⟨proof⟩

lemma run_freeze[run_elims]:
  assumes "run (Array.freeze a)  $\sigma$   $\sigma'$  r"
          "¬is_exn  $\sigma$ "
  obtains " $\sigma' = \sigma$ "

```

```

    "r = Array.get (the_state  $\sigma$ ) a"
  <proof>

```

2.2 Reference Commands

```

lemma run_new_ref[run_elims]:
  assumes "run (ref x)  $\sigma$   $\sigma'$  r"
          "\is_exn  $\sigma$ "
  obtains " $\sigma' = \text{Some} (\text{snd} (\text{Ref.alloc } x (\text{the\_state } \sigma)))$ "
          "r = fst (Ref.alloc x (the_state  $\sigma$ ))"
          "Ref.get (the_state  $\sigma'$ ) r = x"
  <proof>

```

```

lemma "fst (Ref.alloc x h) = Ref (lim h)"
  <proof>

```

```

lemma run_update[run_elims]:
  assumes "run (p := x)  $\sigma$   $\sigma'$  r"
          "\is_exn  $\sigma$ "
  obtains " $\sigma' = \text{Some} (\text{Ref.set } p \ x \ (\text{the\_state } \sigma))$ " "r = ()"
  <proof>

```

```

lemma run_lookup[run_elims]:
  assumes "run (!p)  $\sigma$   $\sigma'$  r"
          "\is_exn  $\sigma$ "
  obtains "\is_exn  $\sigma$ " " $\sigma' = \sigma$ " "r = Ref.get (the_state  $\sigma$ ) p"
  <proof>

```

end

3 Assertions

```

theory Assertions
imports
  "Tools/Imperative_HOL_Add"
  "Tools/Syntax_Match"
  Automatic_Refinement.Misc
begin

```

3.1 Partial Heaps

A partial heap is modeled by a heap and a set of valid addresses, with the side condition that the valid addresses have to be within the limit of the heap. This modeling is somewhat strange for separation logic, however, it allows us to solve some technical problems related to definition of Hoare triples, that will be detailed later.

```

type_synonym pheap = "heap  $\times$  addr set"

```

Predicate that expresses that the address set of a partial heap is within the heap's limit.

```
fun in_range :: "(heap × addr set) ⇒ bool"
  where "in_range (h,as) ⟷ (∀ a ∈ as. a < lim h)"
```

```
declare in_range.simps[simp del]
```

```
lemma in_range_empty[simp, intro!]: "in_range (h,{})"
  ⟨proof⟩
```

```
lemma in_range_dist_union[simp]:
  "in_range (h,as ∪ as') ⟷ in_range (h,as) ∧ in_range (h,as')"
  ⟨proof⟩
```

```
lemma in_range_subset:
  "[as ⊆ as'; in_range (h,as')]" ⟹ in_range (h,as)"
  ⟨proof⟩
```

Relation that holds if two heaps are identical on a given address range

```
definition relH :: "addr set ⇒ heap ⇒ heap ⇒ bool"
  where "relH as h h' ≡
    in_range (h,as)
  ∧ in_range (h',as)
  ∧ (∀ t. ∀ a ∈ as.
    refs h t a = refs h' t a
  ∧ arrays h t a = arrays h' t a
  )"

```

```
lemma relH_in_rangeI:
  assumes "relH as h h'"
  shows "in_range (h,as)" and "in_range (h',as)"
  ⟨proof⟩
```

Reflexivity

```
lemma relH_refl: "in_range (h,as) ⟹ relH as h h"
  ⟨proof⟩
```

Symmetry

```
lemma relH_sym: "relH as h h' ⟹ relH as h' h"
  ⟨proof⟩
```

Transitivity

```
lemma relH_trans[trans]: "[relH as h1 h2; relH as h2 h3]" ⟹ relH as
h1 h3"
  ⟨proof⟩
```

```
lemma relH_dist_union[simp]:
  "relH (as ∪ as') h h' ⟷ relH as h h' ∧ relH as' h h'"

```

<proof>

lemma *relH_subset*:
 assumes "relH bs h h'"
 assumes "as \subseteq bs"
 shows "relH as h h'"
 <proof>

lemma *relH_ref*:
 assumes "relH as h h'"
 assumes "addr_of_ref r \in as"
 shows "Ref.get h r = Ref.get h' r"
 <proof>

lemma *relH_array*:
 assumes "relH as h h'"
 assumes "addr_of_array r \in as"
 shows "Array.get h r = Array.get h' r"
 <proof>

lemma *relH_set_ref*: "[[addr_of_ref r \notin as; in_range (h,as)]]
 \implies relH as h (Ref.set r x h)"
 <proof>

lemma *relH_set_array*: "[[addr_of_array r \notin as; in_range (h,as)]]
 \implies relH as h (Array.set r x h)"
 <proof>

3.2 Assertions

Assertions are predicates on partial heaps, that fulfill a well-formedness condition called properness: They only depend on the part of the heap by the address set, and must be false for partial heaps that are not in range.

type_synonym *assn_raw* = "pheap \Rightarrow bool"

definition *proper* :: "assn_raw \Rightarrow bool" where
 "proper P $\equiv \forall h h' as. (P (h,as) \longrightarrow \text{in_range } (h,as))$
 $\wedge (P (h,as) \wedge \text{relH as h h}' \wedge \text{in_range } (h',as) \longrightarrow P (h',as))$ "

lemma *properI[intro?]*:
 assumes " $\bigwedge as h. P (h,as) \implies \text{in_range } (h,as)$ "
 assumes " $\bigwedge as h h'. P (h,as) \wedge \text{relH as h h}' \implies P (h',as)$ "
 shows "proper P"
 <proof>

lemma *properD1*:
 assumes "proper P"
 assumes "P (h,as)"

```
shows "in_range (h,as)"
⟨proof⟩
```

```
lemma properD2:
  assumes "proper P"
  assumes "P (h,as)"
  assumes "relH as h h'"
  assumes "in_range (h',as)"
  shows "P (h',as)"
⟨proof⟩
```

```
lemmas properD = properD1 properD2
```

```
lemma proper_iff:
  assumes "proper P"
  assumes "relH as h h'"
  assumes "in_range (h',as)"
  shows "P (h,as)  $\longleftrightarrow$  P (h',as)"
⟨proof⟩
```

We encapsulate assertions in their own type

```
typedef assn = "Collect proper"
⟨proof⟩
```

```
lemmas [simp] = Rep_assn_inverse Rep_assn_inject
lemmas [simp, intro!] = Rep_assn[unfolded mem_Collect_eq]
```

```
lemma Abs_assn_eqI[intro?]:
  " $(\bigwedge h. P h = \text{Rep\_assn } Pr h) \implies \text{Abs\_assn } P = Pr$ "
  " $(\bigwedge h. P h = \text{Rep\_assn } Pr h) \implies Pr = \text{Abs\_assn } P$ "
⟨proof⟩
```

```
abbreviation models :: "pheap  $\Rightarrow$  assn  $\Rightarrow$  bool" (infix " $\models$ " 50)
  where "h  $\models$  P  $\equiv$  Rep_assn P h"
```

```
lemma models_in_range: "h  $\models$  P  $\implies$  in_range h"
⟨proof⟩
```

3.2.1 Empty Partial Heap

The empty partial heap satisfies some special properties. We set up a simplification that tries to rewrite it to the standard empty partial heap h_{\perp}

```
abbreviation h_bot ("h $_{\perp}$ ") where "h $_{\perp}$   $\equiv$  (undefined, {})"
lemma mod_h_bot_indep: "(h, {})  $\models$  P  $\longleftrightarrow$  (h', {})  $\models$  P"
⟨proof⟩
```

```
lemma mod_h_bot_normalize[simp]:
```

```
"syntax_fo_nomatch undefined h  $\implies$  (h, { })  $\models$  P  $\longleftrightarrow$  h⊥  $\models$  P"
⟨proof⟩
```

Properness, lifted to the assertion type.

```
lemma mod_relH: "relH as h h'  $\implies$  (h, as)  $\models$  P  $\longleftrightarrow$  (h', as)  $\models$  P"
⟨proof⟩
```

3.3 Connectives

We define several operations on assertions, and instantiate some type classes.

3.3.1 Empty Heap and Separation Conjunction

The assertion that describes the empty heap, and the separation conjunction form a commutative monoid:

```
instantiation assn :: one begin
```

```
  fun one_assn_raw :: "pheap  $\Rightarrow$  bool"
    where "one_assn_raw (h, as)  $\longleftrightarrow$  as={}"
```

```
lemma one_assn_proper[intro!, simp]: "proper one_assn_raw"
⟨proof⟩
```

```
definition one_assn :: assn where "1  $\equiv$  Abs_assn one_assn_raw"
```

```
instance ⟨proof⟩
```

```
end
```

```
abbreviation one_assn::assn ("emp") where "one_assn  $\equiv$  1"
```

```
instantiation assn :: times begin
```

```
  fun times_assn_raw :: "assn_raw  $\Rightarrow$  assn_raw  $\Rightarrow$  assn_raw" where
    "times_assn_raw P Q (h, as)
  = ( $\exists$  as1 as2. as=as1 $\cup$ as2  $\wedge$  as1 $\cap$ as2={ }
     $\wedge$  P (h, as1)  $\wedge$  Q (h, as2))"
```

```
lemma times_assn_proper[intro!, simp]:
```

```
"proper P  $\implies$  proper Q  $\implies$  proper (times_assn_raw P Q)"
```

```
⟨proof⟩
```

```
definition times_assn where "P*Q  $\equiv$ 
```

```
Abs_assn (times_assn_raw (Rep_assn P) (Rep_assn Q))"
```

```
instance ⟨proof⟩
```

```
end
```

```
lemma mod_star_conv: "h  $\models$  A*B
```

```
 $\longleftrightarrow$  ( $\exists$  hr as1 as2. h=(hr, as1 $\cup$ as2)  $\wedge$  as1 $\cap$ as2={ }  $\wedge$  (hr, as1)  $\models$  A  $\wedge$  (hr, as2)  $\models$  B)"
```

```
⟨proof⟩
```

lemma mod_starD: $"h \models A * B \implies \exists h1\ h2. h1 \models A \wedge h2 \models B"$
 $\langle proof \rangle$

lemma star_assnI:
assumes $"(h, as) \models P"$ **and** $"(h, as') \models Q"$ **and** $"as \cap as' = \{\}"$
shows $"(h, as \cup as') \models P * Q"$
 $\langle proof \rangle$

instantiation $assn :: comm_monoid_mult$ **begin**

lemma $assn_one_left: "1 * P = (P :: assn)"$
 $\langle proof \rangle$

lemma $assn_times_comm: "P * Q = Q * (P :: assn)"$
 $\langle proof \rangle$

lemma $assn_times_assoc: "(P * Q) * R = P * (Q * (R :: assn))"$
 $\langle proof \rangle$

instance
 $\langle proof \rangle$

end

3.3.2 Magic Wand

fun $wand_raw :: "assn_raw \Rightarrow assn_raw \Rightarrow assn_raw"$ **where**
 $"wand_raw\ P\ Q\ (h, as) \longleftrightarrow in_range\ (h, as)$
 $\wedge (\forall h'\ as'. as \cap as' = \{\} \wedge relH\ as\ h\ h' \wedge in_range\ (h', as)$
 $\wedge P\ (h', as')$
 $\longrightarrow Q\ (h', as \cup as'))"$

lemma $wand_proper[simp, intro!]: "proper\ (wand_raw\ P\ Q)"$
 $\langle proof \rangle$

definition

$wand_assn :: "assn \Rightarrow assn \Rightarrow assn"$ (**infixl** $"-*"$ 56)
where $"P-*Q \equiv Abs_assn\ (wand_raw\ (Rep_assn\ P)\ (Rep_assn\ Q))"$

lemma $wand_assnI:$
assumes $"in_range\ (h, as)"$
assumes $"\wedge h'\ as'. \llbracket$
 $as \cap as' = \{\};$
 $relH\ as\ h\ h';$
 $in_range\ (h', as);$
 $(h', as') \models Q$
 $\rrbracket \implies (h', as \cup as') \models R"$
shows $"(h, as) \models Q -* R"$
 $\langle proof \rangle$

3.3.3 Boolean Algebra on Assertions

```

instantiation assn :: boolean_algebra begin
  definition top_assn where "top  $\equiv$  Abs_assn in_range"
  definition bot_assn where "bot  $\equiv$  Abs_assn ( $\lambda$ _. False)"
  definition sup_assn where "sup P Q  $\equiv$  Abs_assn ( $\lambda$ h. h|=P  $\vee$  h|=Q)"
  definition inf_assn where "inf P Q  $\equiv$  Abs_assn ( $\lambda$ h. h|=P  $\wedge$  h|=Q)"
  definition uminus_assn where
    "-P  $\equiv$  Abs_assn ( $\lambda$ h. in_range h  $\wedge$   $\neg$ h|=P)"

  lemma bool_assn_proper[simp, intro!]:
    "proper in_range"
    "proper ( $\lambda$ _. False)"
    "proper P  $\implies$  proper Q  $\implies$  proper ( $\lambda$ h. P h  $\vee$  Q h)"
    "proper P  $\implies$  proper Q  $\implies$  proper ( $\lambda$ h. P h  $\wedge$  Q h)"
    "proper P  $\implies$  proper ( $\lambda$ h. in_range h  $\wedge$   $\neg$ P h)"
    <proof>

```

(And, Or, True, False, Not) are a Boolean algebra. Due to idiosyncrasies of the Isabelle/HOL class setup, we have to also define a difference and an ordering:

```

  definition less_eq_assn where
    [simp]: "(a::assn)  $\leq$  b  $\equiv$  a = inf a b"

  definition less_assn where
    [simp]: "(a::assn) < b  $\equiv$  a  $\leq$  b  $\wedge$  a $\neq$ b"

  definition minus_assn where
    [simp]: "(a::assn) - b  $\equiv$  inf a (-b)"

  instance
    <proof>

```

end

We give the operations some more standard names

```

abbreviation top_assn::assn ("true") where "top_assn  $\equiv$  top"
abbreviation bot_assn::assn ("false") where "bot_assn  $\equiv$  bot"
abbreviation sup_assn::"assn $\Rightarrow$ assn $\Rightarrow$ assn" (infixr " $\vee_A$ " 61)
  where "sup_assn  $\equiv$  sup"
abbreviation inf_assn::"assn $\Rightarrow$ assn $\Rightarrow$ assn" (infixr " $\wedge_A$ " 62)
  where "inf_assn  $\equiv$  inf"
abbreviation uminus_assn::"assn  $\Rightarrow$  assn" (" $\neg_A$  _" [81] 80)
  where "uminus_assn  $\equiv$  uminus"

```

Now we prove some relations between the Boolean algebra operations and the (empty heap,separation conjunction) monoid

```

lemma star_false_left[simp]: "false * P = false"
  <proof>

```



```
lemma star_false_right[simp]: "P * false = false"
  <proof>
```

```
lemmas star_false = star_false_left star_false_right
```

```
lemma assn_basic_inequalities[simp, intro!]:
  "true ≠ emp" "emp ≠ true"
  "false ≠ emp" "emp ≠ false"
  "true ≠ false" "false ≠ true"
  <proof>
```

3.3.4 Existential Quantification

```
definition ex_assn :: "('a ⇒ assn) ⇒ assn" (binder "∃A" 11)
  where "(∃Ax. P x) ≡ Abs_assn (λh. ∃x. h|=P x)"
```

```
lemma ex_assn_proper[simp, intro!]:
  "(λx. proper (P x)) ⇒ proper (λh. ∃x. P x h)"
  <proof>
```

```
lemma ex_assn_const[simp]: "(∃Ax. c) = c"
  <proof>
```

```
lemma ex_one_point_gen:
  "[(λh x. h|=P x ⇒ x=v)] ⇒ (∃Ax. P x) = (P v)"
  <proof>
```

```
lemma ex_distrib_star: "(∃Ax. P x * Q) = (∃Ax. P x) * Q"
  <proof>
```

```
lemma ex_distrib_and: "(∃Ax. P x ∧A Q) = (∃Ax. P x) ∧A Q"
  <proof>
```

```
lemma ex_distrib_or: "(∃Ax. P x ∨A Q) = (∃Ax. P x) ∨A Q"
  <proof>
```

```
lemma ex_join_or: "(∃Ax. P x ∨A (∃Ax. Q x)) = (∃Ax. P x ∨A Q x)"
  <proof>
```

3.3.5 Pure Assertions

Pure assertions do not depend on any heap content.

```
fun pure_assn_raw where "pure_assn_raw b (h,as) ←→ as={ } ∧ b"
definition pure_assn :: "bool ⇒ assn" ("↑") where
  "↑b ≡ Abs_assn (pure_assn_raw b)"
```

```
lemma pure_assn_proper[simp, intro!]: "proper (pure_assn_raw b)"
  <proof>
```

```

lemma pure_true[simp]: " $\uparrow$ True = emp"
  <proof>

lemma pure_false[simp]: " $\uparrow$ False = false"
  <proof>

lemma pure_assn_eq_false_iff[simp]: " $\uparrow$ P = false  $\longleftrightarrow$   $\neg$ P" <proof>

lemma pure_assn_eq_emp_iff[simp]: " $\uparrow$ P = emp  $\longleftrightarrow$  P" <proof>

lemma merge_pure_star[simp]:
  " $\uparrow$ a *  $\uparrow$ b =  $\uparrow$ (a $\wedge$ b)"
  <proof>

lemma merge_true_star[simp]: "true*true = true"
  <proof>

lemma merge_pure_and[simp]:
  " $\uparrow$ a  $\wedge_A$   $\uparrow$ b =  $\uparrow$ (a $\wedge$ b)"
  <proof>

lemma merge_pure_or[simp]:
  " $\uparrow$ a  $\vee_A$   $\uparrow$ b =  $\uparrow$ (a $\vee$ b)"
  <proof>

lemma pure_assn_eq_conv[simp]: " $\uparrow$ P =  $\uparrow$ Q  $\longleftrightarrow$  P=Q" <proof>

definition "is_pure_assn a  $\equiv$   $\exists$ P. a= $\uparrow$ P"
lemma is_pure_assnE: assumes "is_pure_assn a" obtains P where "a= $\uparrow$ P"
  <proof>

lemma is_pure_assn_pure[simp, intro!]: "is_pure_assn ( $\uparrow$ P)"
  <proof>

lemma is_pure_assn_basic_simps[simp]:
  "is_pure_assn false"
  "is_pure_assn emp"
  <proof>

lemma is_pure_assn_starI[simp, intro!]:
  "[[is_pure_assn a; is_pure_assn b]]  $\implies$  is_pure_assn (a*b)"
  <proof>

```

3.3.6 Pointers

In Imperative HOL, we have to distinguish between pointers to single values and pointers to arrays. For both, we define assertions that describe the part of the heap that a pointer points to.

```
fun sngr_assn_raw :: "'a::heap ref ⇒ 'a ⇒ assn_raw" where
  "sngr_assn_raw r x (h,as) ⟷ Ref.get h r = x ∧ as = {addr_of_ref r}
  ∧
  addr_of_ref r < lim h"
```

```
lemma sngr_assn_proper[simp, intro!]: "proper (sngr_assn_raw r x)"
  ⟨proof⟩
```

```
definition sngr_assn :: "'a::heap ref ⇒ 'a ⇒ assn" (infix "↦r" 82)
  where "r↦rx ≡ Abs_assn (sngr_assn_raw r x)"
```

```
fun snga_assn_raw :: "'a::heap array ⇒ 'a list ⇒ assn_raw"
  where "snga_assn_raw r x (h,as)
  ⟷ Array.get h r = x ∧ as = {addr_of_array r}
  ∧ addr_of_array r < lim h"
```

```
lemma snga_assn_proper[simp, intro!]: "proper (snga_assn_raw r x)"
  ⟨proof⟩
```

```
definition
  snga_assn :: "'a::heap array ⇒ 'a list ⇒ assn" (infix "↦a" 82)
  where "r↦aa ≡ Abs_assn (snga_assn_raw r a)"
```

Two disjoint parts of the heap cannot be pointed to by the same pointer

```
lemma sngr_same_false[simp]:
  "p ↦r x * p ↦r y = false"
  ⟨proof⟩
```

```
lemma snga_same_false[simp]:
  "p ↦a x * p ↦a y = false"
  ⟨proof⟩
```

3.4 Properties of the Models-Predicate

```
lemma mod_true[simp]: "h|=true ⟷ in_range h"
  ⟨proof⟩
```

```
lemma mod_false[simp]: "¬ h|=false"
  ⟨proof⟩
```

```
lemma mod_emp: "h|=emp ⟷ snd h = {}"
  ⟨proof⟩
```

```
lemma mod_emp_simp[simp]: "(h,{})=emp"
  ⟨proof⟩
```

lemma mod_pure[simp]: $"h \models \uparrow b \longleftrightarrow \text{snd } h = \{\} \wedge b"$
 ⟨proof⟩

lemma mod_ex_dist[simp]: $"h \models (\exists_A x. P \ x) \longleftrightarrow (\exists x. h \models P \ x)"$
 ⟨proof⟩

lemma mod_exI: $"\exists x. h \models P \ x \implies h \models (\exists_A x. P \ x)"$
 ⟨proof⟩

lemma mod_exE: **assumes** $"h \models (\exists_A x. P \ x)"$ **obtains** x **where** $"h \models P \ x"$
 ⟨proof⟩

lemma mod_and_dist: $"h \models P \wedge_A Q \longleftrightarrow h \models P \wedge h \models Q"$
 ⟨proof⟩

lemma mod_or_dist[simp]: $"h \models P \vee_A Q \longleftrightarrow h \models P \vee h \models Q"$
 ⟨proof⟩

lemma mod_not_dist[simp]: $"h \models (\neg_A P) \longleftrightarrow \text{in_range } h \wedge \neg h \models P"$
 ⟨proof⟩

lemma mod_pure_star_dist[simp]: $"h \models P * \uparrow b \longleftrightarrow h \models P \wedge b"$
 ⟨proof⟩

lemmas mod_dist = mod_pure mod_pure_star_dist mod_ex_dist mod_and_dist
mod_or_dist mod_not_dist

lemma mod_star_trueI: $"h \models P \implies h \models P * \text{true}"$
 ⟨proof⟩

lemma mod_star_trueE': **assumes** $"h \models P * \text{true}"$ **obtains** h' **where**
 $"\text{fst } h' = \text{fst } h"$ **and** $"\text{snd } h' \subseteq \text{snd } h"$ **and** $"h' \models P"$
 ⟨proof⟩

lemma mod_star_trueE: **assumes** $"h \models P * \text{true}"$ **obtains** h' **where** $"h' \models P"$
 ⟨proof⟩

lemma mod_h_bot_iff[simp]:
 $"(h, \{\}) \models \uparrow b \longleftrightarrow b"$
 $"(h, \{\}) \models \text{true}"$
 $"(h, \{\}) \models p \mapsto_r x \longleftrightarrow \text{False}"$
 $"(h, \{\}) \models q \mapsto_a y \longleftrightarrow \text{False}"$
 $"(h, \{\}) \models P * Q \longleftrightarrow ((h, \{\}) \models P) \wedge ((h, \{\}) \models Q)"$
 $"(h, \{\}) \models P \wedge_A Q \longleftrightarrow ((h, \{\}) \models P) \wedge ((h, \{\}) \models Q)"$
 $"(h, \{\}) \models P \vee_A Q \longleftrightarrow ((h, \{\}) \models P) \vee ((h, \{\}) \models Q)"$
 $"(h, \{\}) \models (\exists_A x. R \ x) \longleftrightarrow (\exists x. (h, \{\}) \models R \ x)"$
 ⟨proof⟩

3.5 Entailment

definition *entails* :: "assn \Rightarrow assn \Rightarrow bool" (infix " \Longrightarrow_A " 10)
where " $P \Longrightarrow_A Q \equiv \forall h. h \models P \longrightarrow h \models Q$ "

lemma *entailsI*:
assumes " $\bigwedge h. h \models P \Longrightarrow h \models Q$ "
shows " $P \Longrightarrow_A Q$ "
<proof>

lemma *entailsD*:
assumes " $P \Longrightarrow_A Q$ "
assumes " $h \models P$ "
shows " $h \models Q$ "
<proof>

3.5.1 Properties

lemma *ent_fwd*:
assumes " $h \models P$ "
assumes " $P \Longrightarrow_A Q$ "
shows " $h \models Q$ " *<proof>*

lemma *ent_refl[simp]*: " $P \Longrightarrow_A P$ "
<proof>

lemma *ent_trans[trans]*: " $\llbracket P \Longrightarrow_A Q; Q \Longrightarrow_{AR} \rrbracket \Longrightarrow P \Longrightarrow_A R$ "
<proof>

lemma *ent_iffI*:
assumes " $A \Longrightarrow_A B$ "
assumes " $B \Longrightarrow_A A$ "
shows " $A=B$ "
<proof>

lemma *ent_false[simp]*: " $false \Longrightarrow_A P$ "
<proof>

lemma *ent_true[simp]*: " $P \Longrightarrow_A true$ "
<proof>

lemma *ent_false_iff[simp]*: " $(P \Longrightarrow_A false) \longleftrightarrow (\forall h. \neg h \models P)$ "
<proof>

lemma *ent_pure_pre_iff[simp]*: " $(P * \uparrow b \Longrightarrow_A Q) \longleftrightarrow (b \longrightarrow (P \Longrightarrow_A Q))$ "
<proof>

lemma *ent_pure_pre_iff_sng[simp]*:
" $(\uparrow b \Longrightarrow_A Q) \longleftrightarrow (b \longrightarrow (emp \Longrightarrow_A Q))$ "
<proof>

lemma ent_pure_post_iff[simp]:

$$(P \Longrightarrow_A Q * \uparrow b) \longleftrightarrow ((\forall h. h \models P \longrightarrow b) \wedge (P \Longrightarrow_A Q))$$
 $\langle proof \rangle$

lemma ent_pure_post_iff_sng[simp]:

$$(P \Longrightarrow_A \uparrow b) \longleftrightarrow ((\forall h. h \models P \longrightarrow b) \wedge (P \Longrightarrow_A emp))$$
 $\langle proof \rangle$

lemma ent_ex_preI: $(\bigwedge x. P x \Longrightarrow_A Q) \Longrightarrow \exists_{Ax}. P x \Longrightarrow_A Q$
 $\langle proof \rangle$

lemma ent_ex_postI: $(P \Longrightarrow_A Q x) \Longrightarrow P \Longrightarrow_A \exists_{Ax}. Q x$
 $\langle proof \rangle$

lemma ent_mp: $(P * (P -* Q)) \Longrightarrow_A Q$
 $\langle proof \rangle$

lemma ent_star_mono: $\llbracket P \Longrightarrow_A P'; Q \Longrightarrow_A Q' \rrbracket \Longrightarrow P * Q \Longrightarrow_A P' * Q'$
 $\langle proof \rangle$

lemma ent_wandI:
assumes $IMP: "Q * P \Longrightarrow_A R"$
shows $"P \Longrightarrow_A (Q -* R)"$
 $\langle proof \rangle$

lemma ent_disjI1:
assumes $"P \vee_A Q \Longrightarrow_A R"$
shows $"P \Longrightarrow_A R"$ $\langle proof \rangle$

lemma ent_disjI2:
assumes $"P \vee_A Q \Longrightarrow_A R"$
shows $"Q \Longrightarrow_A R"$ $\langle proof \rangle$

lemma ent_disjI1_direct[simp]: $"A \Longrightarrow_A A \vee_A B"$
 $\langle proof \rangle$

lemma ent_disjI2_direct[simp]: $"B \Longrightarrow_A A \vee_A B"$
 $\langle proof \rangle$

lemma ent_disjE: $\llbracket A \Longrightarrow_A C; B \Longrightarrow_A C \rrbracket \Longrightarrow A \vee_A B \Longrightarrow_A C$
 $\langle proof \rangle$

lemma ent_conjI: $\llbracket A \Longrightarrow_A B; A \Longrightarrow_A C \rrbracket \Longrightarrow A \Longrightarrow_A B \wedge_A C$
 $\langle proof \rangle$

lemma ent_conjE1: $\llbracket A \Longrightarrow_A C \rrbracket \Longrightarrow A \wedge_A B \Longrightarrow_A C$
 $\langle proof \rangle$

lemma ent_conjE2: $\llbracket B \Longrightarrow_A C \rrbracket \Longrightarrow A \wedge_A B \Longrightarrow_A C$
 $\langle proof \rangle$

```

lemma star_or_dist1:
  "(A  $\vee_A$  B)*C = (A*C  $\vee_A$  B*C)"
  <proof>

lemma star_or_dist2:
  "C*(A  $\vee_A$  B) = (C*A  $\vee_A$  C*B)"
  <proof>

lemmas star_or_dist = star_or_dist1 star_or_dist2

lemma ent_disjI1': "A  $\implies_A$  B  $\implies$  A  $\implies_{AB \vee_A C}$ "
  <proof>

lemma ent_disjI2': "A  $\implies_A$  C  $\implies$  A  $\implies_{AB \vee_A C}$ "
  <proof>

lemma triv_exI[simp, intro!]: "Q x  $\implies_A$   $\exists_{Ax}$ . Q x"
  <proof>

```

3.5.2 Weak Entails

Weakening of entails to allow arbitrary unspecified memory in conclusion

```

definition entailst :: "assn  $\Rightarrow$  assn  $\Rightarrow$  bool" (infix " $\implies_t$ " 10)
  where "entailst A B  $\equiv$  A  $\implies_A$  B * true"

```

```

lemma enttI: "A  $\implies_{AB*true} \implies$  A  $\implies_t B$ " <proof>
lemma enttD: "A  $\implies_t B \implies$  A  $\implies_{AB*true}$ " <proof>

lemma entt_trans:
  "entailst A B  $\implies$  entailst B C  $\implies$  entailst A C"
  <proof>

lemma entt_refl[simp, intro!]: "entailst A A"
  <proof>

lemma entt_true[simp, intro!]:
  "entailst A true"
  <proof>

lemma entt_emp[simp, intro!]:
  "entailst A emp"
  <proof>

lemma entt_star_true_simp[simp]:
  "entailst A (B*true)  $\longleftrightarrow$  entailst A B"
  "entailst (A*true) B  $\longleftrightarrow$  entailst A B"

```

$\langle proof \rangle$

lemma *entt_star_mono*: " $\llbracket entailst\ A\ B;\ entailst\ C\ D \rrbracket \implies entailst\ (A*C)\ (B*D)$ "
 $\langle proof \rangle$

lemma *entt_frame_fwd*:
assumes "*entailst P Q*"
assumes "*entailst A (P*F)*"
assumes "*entailst (Q*F) B*"
shows "*entailst A B*"
 $\langle proof \rangle$

lemma *enttI_true*: " $P*true \implies_A Q*true \implies P \implies_t Q$ "
 $\langle proof \rangle$

lemma *entt_def_true*: " $(P \implies_t Q) \equiv (P*true \implies_A Q*true)$ "
 $\langle proof \rangle$

lemma *ent_imp_entt*: " $P \implies_A Q \implies P \implies_t Q$ "
 $\langle proof \rangle$

lemma *entt_disjI1_direct[simp]*: " $A \implies_t A \vee_A B$ "
 $\langle proof \rangle$

lemma *entt_disjI2_direct[simp]*: " $B \implies_t A \vee_A B$ "
 $\langle proof \rangle$

lemma *entt_disjI1'*: " $A \implies_t B \implies A \implies_t B \vee_A C$ "
 $\langle proof \rangle$

lemma *entt_disjI2'*: " $A \implies_t C \implies A \implies_t B \vee_A C$ "
 $\langle proof \rangle$

lemma *entt_disjE*: " $\llbracket A \implies_t M;\ B \implies_t M \rrbracket \implies A \vee_A B \implies_t M$ "
 $\langle proof \rangle$

lemma *entt_disjD1*: " $A \vee_A B \implies_t C \implies A \implies_t C$ "
 $\langle proof \rangle$

lemma *entt_disjD2*: " $A \vee_A B \implies_t C \implies B \implies_t C$ "
 $\langle proof \rangle$

3.6 Precision

Precision rules describe that parts of an assertion may depend only on the underlying heap. For example, the data where a pointer points to is the same for the same heap.

Precision rules should have the form:

$$\forall x y. (h \models (P x * F1) \wedge_A (P y * F2)) \longrightarrow x=y$$

definition "precise R $\equiv \forall a a' h p F F'.$
 $h \models R a p * F \wedge_A R a' p * F' \longrightarrow a = a'$ "

lemma preciseI[*intro?*]:
assumes " $\wedge a a' h p F F'. h \models R a p * F \wedge_A R a' p * F' \implies a = a'$ "
shows "precise R"
 <proof>

lemma preciseD:
assumes "precise R"
assumes " $h \models R a p * F \wedge_A R a' p * F'$ "
shows " $a=a'$ "
 <proof>

lemma preciseD':
assumes "precise R"
assumes " $h \models R a p * F$ "
assumes " $h \models R a' p * F'$ "
shows " $a=a'$ "
 <proof>

lemma precise_extr_pure[*simp*]:
 "precise $(\lambda x y. \uparrow P * R x y) \longleftrightarrow (P \longrightarrow \text{precise } R)$ "
 "precise $(\lambda x y. R x y * \uparrow P) \longleftrightarrow (P \longrightarrow \text{precise } R)$ "
 <proof>

lemma sng_r_prec: "precise $(\lambda x p. p \mapsto_r x)$ "
 <proof>

lemma sng_a_prec: "precise $(\lambda x p. p \mapsto_a x)$ "
 <proof>

end

4 Hoare-Triples

theory Hoare_Triple
imports Run Assertions
begin

In this theory, we define Hoare-Triples, which are our basic tool for specifying properties of Imperative HOL programs.

4.1 Definition

Analyze the heap before and after executing a command, to add the allocated addresses to the covered address range.

definition `new_addrs` :: "heap \Rightarrow addr set \Rightarrow heap \Rightarrow addr set" where
 "new_addrs h as h' = as \cup {a. lim h \leq a \wedge a < lim h'}"

lemma `new_addr_refl[simp]`: "new_addrs h as h = as"
 <proof>

Apart from correctness of the program wrt. the pre- and post condition, a Hoare-triple also encodes some well-formedness conditions of the command: The command must not change addresses outside the address range of the precondition, and it must not decrease the heap limit.

Note that we do not require that the command only reads from heap locations inside the precondition's address range, as this condition would be quite complicated to express with the heap model of Imperative/HOL, and is not necessary in our formalization of partial heaps, that always contain the information for all addresses.

definition `hoare_triple`
 :: "assn \Rightarrow 'a Heap \Rightarrow ('a \Rightarrow assn) \Rightarrow bool" (" $\langle_ \rangle / _ / \langle_ \rangle$ ")
 where
 " $\langle P \rangle c \langle Q \rangle \equiv \forall h \text{ as } \sigma r. (h, \text{as}) \models P \wedge \text{run } c \text{ (Some } h) \sigma r$
 $\longrightarrow (\text{let } h' = \text{the_state } \sigma; \text{as}' = \text{new_addrs } h \text{ as } h' \text{ in}$
 $\neg \text{is_exn } \sigma \wedge (h', \text{as}') \models Q r \wedge \text{relH } (\{a . a < \text{lim } h \wedge a \notin \text{as}\}) h h'$
 $\wedge \text{lim } h \leq \text{lim } h')$ "

Sanity checking theorems for Hoare-Triples

lemma
 assumes " $\langle P \rangle c \langle Q \rangle$ "
 assumes " $(h, \text{as}) \models P$ "
 shows `hoare_triple_success`: "success c h"
 and `hoare_triple_effect`: " $\exists h' r. \text{effect } c h h' r \wedge (h', \text{new_addrs } h \text{ as } h') \models Q r$ "
 <proof>

lemma `hoare_tripleD`:
 fixes h h' as as' σr
 assumes " $\langle P \rangle c \langle Q \rangle$ "
 assumes " $(h, \text{as}) \models P$ "
 assumes " $\text{run } c \text{ (Some } h) \sigma r$ "
 defines " $h' \equiv \text{the_state } \sigma$ " and " $\text{as}' \equiv \text{new_addrs } h \text{ as } h'$ "
 shows " $\neg \text{is_exn } \sigma$ "
 and " $(h', \text{as}') \models Q r$ "
 and " $\text{relH } (\{a . a < \text{lim } h \wedge a \notin \text{as}\}) h h'$ "
 and " $\text{lim } h \leq \text{lim } h'$ "

<proof>

For garbage-collected languages, specifications usually allow for some arbitrary heap parts in the postcondition. The following abbreviation defines a handy shortcut notation for such specifications.

abbreviation *hoare_triple'*

:: "assn \Rightarrow 'r Heap \Rightarrow ('r \Rightarrow assn) \Rightarrow bool" ("*<_>* _ *<_>*_t")
where "*<P>* c *<Q>*_t \equiv *<P>* c *<\lambda r. Q r * true>*"

4.2 Rules

In this section, we provide a set of rules to prove Hoare-Triples correct.

4.2.1 Basic Rules

lemma *hoare_triple_preI*:

assumes " $\bigwedge h. h \models P \implies \langle P \rangle c \langle Q \rangle$ "
shows "*<P>* c *<Q>*"
<proof>

lemma *frame_rule*:

assumes A: "*<P>* c *<Q>*"
shows "*<P*R>* c *<\lambda x. Q x * R>*"
<proof>

lemma *false_rule*[*simp, intro!*]: "*<>false>* c *<Q>*"

<proof>

lemma *cons_rule*:

assumes CPRE: " $P \implies_A P'$ "
assumes CPOST: " $\bigwedge x. Q x \implies_A Q' x$ "
assumes R: "*<P'>* c *<Q>*"
shows "*<P>* c *<Q'>*"
<proof>

lemmas *cons_pre_rule* = *cons_rule*[*OF _ ent_refl*]

lemmas *cons_post_rule* = *cons_rule*[*OF ent_refl, rotated*]

lemma *cons_rulet*: " $\llbracket P \implies_t P'; \bigwedge x. Q x \implies_t Q' x; \langle P' \rangle c \langle Q \rangle_t \rrbracket \implies \langle P \rangle c \langle Q' \rangle_t$ "

<proof>

lemmas *cons_pre_rulet* = *cons_rulet*[*OF _ entt_refl*]

lemmas *cons_post_rulet* = *cons_rulet*[*OF entt_refl, rotated*]

lemma norm_pre_ex_rule:
 assumes A: " $\bigwedge x. \langle P \ x \rangle f \langle Q \rangle$ "
 shows " $\langle \exists_{Ax}. P \ x \rangle f \langle Q \rangle$ "
<proof>

lemma norm_pre_pure_iff[simp]:
 " $\langle P * \uparrow b \rangle f \langle Q \rangle \longleftrightarrow (b \longrightarrow \langle P \rangle f \langle Q \rangle)$ "
<proof>

lemma norm_pre_pure_iff_sng[simp]:
 " $\langle \uparrow b \rangle f \langle Q \rangle \longleftrightarrow (b \longrightarrow \langle \text{emp} \rangle f \langle Q \rangle)$ "
<proof>

lemma norm_pre_pure_rule1:
 " $\llbracket b \implies \langle P \rangle f \langle Q \rangle \rrbracket \implies \langle P * \uparrow b \rangle f \langle Q \rangle$ " *<proof>*

lemma norm_pre_pure_rule2:
 " $\llbracket b \implies \langle \text{emp} \rangle f \langle Q \rangle \rrbracket \implies \langle \uparrow b \rangle f \langle Q \rangle$ " *<proof>*

lemmas norm_pre_pure_rule = norm_pre_pure_rule1 norm_pre_pure_rule2

lemma post_exI_rule: " $\langle P \rangle c \langle \lambda r. Q \ r \ x \rangle \implies \langle P \rangle c \langle \lambda r. \exists_{Ax}. Q \ r \ x \rangle$ "
<proof>

4.2.2 Rules for Atomic Commands

lemma ref_rule:
 " $\langle \text{emp} \rangle \text{ref } x \langle \lambda r. r \mapsto_r x \rangle$ "
<proof>

lemma lookup_rule:
 " $\langle p \mapsto_r x \rangle !p \langle \lambda r. p \mapsto_r x * \uparrow (r = x) \rangle$ "
<proof>

lemma update_rule:
 " $\langle p \mapsto_r y \rangle p := x \langle \lambda r. p \mapsto_r x \rangle$ "
<proof>

lemma update_wp_rule:
 " $\langle r \mapsto_r y * ((r \mapsto_r x) -* (Q \ ())) \rangle r := x \langle Q \rangle$ "
<proof>

lemma new_rule:
 " $\langle \text{emp} \rangle \text{Array.new } n \ x \langle \lambda r. r \mapsto_a \text{replicate } n \ x \rangle$ "
<proof>

lemma make_rule: " $\langle \text{emp} \rangle \text{Array.make } n \ f \langle \lambda r. r \mapsto_a (\text{map } f \ [0 \ .. \ n]) \rangle$ "

<proof>

lemma of_list_rule: "*<emp> Array.of_list xs < $\lambda r. r \mapsto_a xs$ >*"
<proof>

lemma length_rule:
"*<a $\mapsto_a xs$ > Array.len a < $\lambda r. a \mapsto_a xs * \uparrow(r = \text{length } xs)$ >*"
<proof>

Note that the Boolean expression is placed at meta level and not inside the precondition. This makes frame inference simpler.

lemma nth_rule:
"*[i < length xs] \implies <a $\mapsto_a xs$ > Array.nth a i < $\lambda r. a \mapsto_a xs * \uparrow(r = xs ! i)$ >*"
<proof>

lemma upd_rule:
"*[i < length xs] \implies
<a $\mapsto_a xs$ >
Array.upd i x a
< $\lambda r. (a \mapsto_a (\text{list_update } xs \ i \ x)) * \uparrow(r = a)$ >*"
<proof>

lemma freeze_rule:
"*<a $\mapsto_a xs$ > Array.freeze a < $\lambda r. a \mapsto_a xs * \uparrow(r = xs)$ >*"
<proof>

lemma return_wp_rule:
"*<Q x> return x <Q>*"
<proof>

lemma return_sp_rule:
"*<P> return x < $\lambda r. P * \uparrow(r = x)$ >*"
<proof>

lemma raise_iff:
"*<P> raise s <Q> $\longleftrightarrow P = \text{false}$* "
<proof>

lemma raise_rule: "*<false> raise s <Q>*"
<proof>

4.2.3 Rules for Composed Commands

lemma bind_rule:
assumes T1: "*<P> f <R>*"
assumes T2: " *$\bigwedge x. <R \ x> g \ x \ <Q>$* "
shows "*<P> bind f g <Q>*"
<proof>

```

lemma if_rule:
  assumes "b  $\implies$  <P> f <Q>"
  assumes " $\neg$ b  $\implies$  <P> g <Q>"
  shows "<P> if b then f else g <Q>"
  <proof>

lemma if_rule_split:
  assumes B: "b  $\implies$  <P> f <Q1>"
  assumes NB: " $\neg$ b  $\implies$  <P> g <Q2>"
  assumes M: " $\bigwedge x. (Q1\ x * \uparrow b) \vee_A (Q2\ x * \uparrow(\neg b)) \implies_A Q\ x$ "
  shows "<P> if b then f else g <Q>"
  <proof>

lemma split_rule:
  assumes P: "<P> c <R>"
  assumes Q: "<Q> c <R>"
  shows "<P  $\vee_A$  Q> c <R>"
  <proof>

lemmas decon_if_split = if_rule_split split_rule
  — Use with care: Complete splitting of if statements

lemma case_prod_rule:
  " $(\bigwedge a\ b. x = (a, b) \implies <P> f\ a\ b <Q>) \implies <P> \text{case } x \text{ of } (a, b) \Rightarrow f\ a\ b <Q>$ "
  <proof>

lemma case_list_rule:
  " $[\ l=[] \implies <P> fn <Q>; \bigwedge x\ xs. l=x\#xs \implies <P> fc\ x\ xs <Q> ] \implies <P> \text{case\_list } fn\ fc\ l <Q>$ "
  <proof>

lemma case_option_rule:
  " $[\ v=None \implies <P> fn <Q>; \bigwedge x. v=Some\ x \implies <P> fs\ x <Q> ] \implies <P> \text{case\_option } fn\ fs\ v <Q>$ "
  <proof>

lemma case_sum_rule:
  " $[\ \bigwedge x. v=Inl\ x \implies <P> fl\ x <Q>; \bigwedge x. v=Inr\ x \implies <P> fr\ x <Q> ] \implies <P> \text{case\_sum } fl\ fr\ v <Q>$ "
  <proof>

lemma let_rule: " $(\bigwedge x. x = t \implies <P> f\ x <Q>) \implies <P> \text{Let } t\ f <Q>$ "
  <proof>

end

```

5 Automation

```
theory Automation
imports Hoare_Triple
begin
```

In this theory, we provide a set of tactics and a simplifier setup for easy reasoning with our separation logic.

5.1 Normalization of Assertions

In this section, we provide a set of lemmas and a simplifier setup to bring assertions to a normal form. We provide a simproc that detects pure parts of assertions and duplicate pointers. Moreover, we provide ac-rules for assertions. See Section 5.9 for a short overview of the available proof methods.

```
lemmas assn_aci =
  inf_aci[where 'a=assn]
  sup_aci[where 'a=assn]
  mult.left_ac[where 'a=assn]
```

```
lemmas star_assoc = mult.assoc[where 'a=assn]
lemmas assn_assoc =
  mult.left_assoc inf_assoc[where 'a=assn] sup_assoc[where 'a=assn]
```

```
lemma merge_true_star_ctx: "true * (true * P) = true * P"
  <proof>
```

```
lemmas star_aci =
  mult.assoc[where 'a=assn] mult.commute[where 'a=assn] mult.left_commute[where
'a=assn]
  assn_one_left mult_1_right[where 'a=assn]
  merge_true_star merge_true_star_ctx
```

Move existential quantifiers to the front of assertions

```
lemma ex_assn_move_out[simp]:
  " $\bigwedge Q R. (\exists_{Ax}. Q x) * R = (\exists_{Ax}. (Q x * R))$ "
  " $\bigwedge Q R. R * (\exists_{Ax}. Q x) = (\exists_{Ax}. (R * Q x))$ "

  " $\bigwedge P Q. (\exists_{Ax}. Q x) \wedge_A P = (\exists_{Ax}. (Q x \wedge_A P))$ "
  " $\bigwedge P Q. Q \wedge_A (\exists_{Ax}. P x) = (\exists_{Ax}. (Q \wedge_A P x))$ "

  " $\bigwedge P Q. (\exists_{Ax}. Q x) \vee_A P = (\exists_{Ax}. (Q x \vee_A P))$ "
  " $\bigwedge P Q. Q \vee_A (\exists_{Ax}. P x) = (\exists_{Ax}. (Q \vee_A P x))$ "
  <proof>
```

Extract pure assertions from and-clauses

```
lemma and_extract_pure_left_iff[simp]: " $\uparrow b \wedge_A Q = (\text{emp} \wedge_A Q) * \uparrow b$ "
  <proof>
```

```

lemma and_extract_pure_left_ctx_iff[simp]: "P*↑b ∧A Q = (P∧AQ)*↑b"
  ⟨proof⟩

lemma and_extract_pure_right_iff[simp]: "P ∧A ↑b = (emp∧AP)*↑b"
  ⟨proof⟩

lemma and_extract_pure_right_ctx_iff[simp]: "P ∧A Q*↑b = (P∧AQ)*↑b"
  ⟨proof⟩

lemmas and_extract_pure_iff =
  and_extract_pure_left_iff and_extract_pure_left_ctx_iff
  and_extract_pure_right_iff and_extract_pure_right_ctx_iff

lemmas norm_assertion_simps =

  mult_1[where 'a=assn] mult_1_right[where 'a=assn]
  inf_top_left[where 'a=assn] inf_top_right[where 'a=assn]
  sup_bot_left[where 'a=assn] sup_bot_right[where 'a=assn]

  star_false_left star_false_right
  inf_bot_left[where 'a=assn] inf_bot_right[where 'a=assn]
  sup_top_left[where 'a=assn] sup_top_right[where 'a=assn]

  mult.left_assoc[where 'a=assn]
  inf_assoc[where 'a=assn]
  sup_assoc[where 'a=assn]

  ex_assn_move_out ex_assn_const

  and_extract_pure_iff

  merge_pure_star merge_pure_and merge_pure_or
  merge_true_star
  inf_idem[where 'a=assn] sup_idem[where 'a=assn]

  sngr_same_false snga_same_false

```

5.1.1 Simplifier Setup Fine-Tuning

Imperative HOL likes to simplify pointer inequations to this strange operator. We do some additional simplifier setup here

```
lemma not_same_noteqr[simp]: "¬ a!=a"
```



```

  <proof>
  declare Ref.noteq_irrefl[dest!]

  lemma not_same_noteqa[simp]: "¬ a=!!=a"
  <proof>
  declare Array.noteq_irrefl[dest!]

```

However, it is safest to disable this rewriting, as there is a working standard simplifier setup for (\neq)

```

  declare Ref.unequal[simp del]
  declare Array.unequal[simp del]

```

5.2 Normalization of Entailments

Used by existential quantifier extraction tactic

```

  lemma enorm_exI':
  "( $\bigwedge x. Z x \longrightarrow (P \Longrightarrow_A Q x)$ )  $\Longrightarrow$  ( $\exists x. Z x$ )  $\longrightarrow$  ( $P \Longrightarrow_A (\exists Ax. Q x)$ )"
  <proof>

```

Example of how to build an extraction lemma.

```

  thm enorm_exI' [OF enorm_exI' [OF imp_refl]]

```

```

  lemmas ent_triv = ent_true ent_false

```

Dummy rule to detect Hoare triple goal

```

  lemma is_hoare_triple: "<P> c <Q>  $\Longrightarrow$  <P> c <Q>" <proof>

```

Dummy rule to detect entailment goal

```

  lemma is_entails: "P  $\Longrightarrow_A$  Q  $\Longrightarrow$  P  $\Longrightarrow_A$  Q" <proof>

```

5.3 Frame Matcher

Given star-lists P,Q and a frame F, this method tries to match all elements of Q with corresponding elements of P. The result is a partial match, that contains matching pairs and the unmatched content.

The frame-matcher internally uses syntactic lists separated by star, and delimited by the special symbol *SLN*, which is defined to be *emp*.

```

  definition [simp]: "SLN  $\equiv$  emp"
  lemma SLN_left: "SLN * P = P" <proof>
  lemma SLN_right: "P * SLN = P" <proof>

```

```

  lemmas SLN_normalize = SLN_right mult.left_assoc[where 'a=assn]
  lemmas SLN_strip = SLN_right SLN_left mult.left_assoc[where 'a=assn]

```

A query to the frame matcher. Contains the assertions P and Q that shall be matched, as well as a frame F, that is not touched.

definition *[simp]*: "*FI_QUERY P Q F* $\equiv P \implies_A Q * F$ "

abbreviation "*fi_m_fst M* $\equiv \text{foldr } (*) \text{ (map fst M) emp}$ "

abbreviation "*fi_m_snd M* $\equiv \text{foldr } (*) \text{ (map snd M) emp}$ "

abbreviation "*fi_m_match M* $\equiv (\forall (p,q) \in \text{set } M. p \implies_A q)$ "

A result of the frame matcher. Contains a list of matching pairs, as well as the unmatched parts of P and Q, and the frame F.

definition *[simp]*: "*FI_RESULT M UP UQ F* \equiv
fi_m_match M $\longrightarrow (fi_m_fst\ M * UP \implies_A fi_m_snd\ M * UQ * F)$ "

Internal structure used by the frame matcher: m contains the matched pairs; p,q the assertions that still needs to be matched; up,uq the assertions that could not be matched; and f the frame. p and q are SLN-delimited syntactic lists.

definition *[simp]*: "*FI m p q up uq f* \equiv
fi_m_match m $\longrightarrow (fi_m_fst\ m * p * up \implies_A fi_m_snd\ m * q * uq * f)$ "

Initialize processing of query

lemma *FI_init*:
assumes "*FI [] (SLN*P) (SLN*Q) SLN SLN F*"
shows "*FI_QUERY P Q F*"
 $\langle \text{proof} \rangle$

Construct result from internal representation

lemma *FI_finalize*:
assumes "*FI_RESULT m (p*up) (q*uq) f*"
shows "*FI m p q up uq f*"
 $\langle \text{proof} \rangle$

Auxiliary lemma to show that all matching pairs together form an entailment. This is required for most applications.

lemma *fi_match_entails*:
assumes "*fi_m_match m*"
shows "*fi_m_fst m \implies_A fi_m_snd m*"
 $\langle \text{proof} \rangle$

Internally, the frame matcher tries to match the first assertion of q with the first assertion of p. If no match is found, the first assertion of p is discarded. If no match for any assertion in p can be found, the first assertion of q is discarded.

Match

lemma *FI_match*:
assumes "*p \implies_A q*"
assumes "*FI ((p,q)#m) (ps*up) (qs*uq) SLN SLN f*"
shows "*FI m (ps*p) (qs*q) up uq f*"

<proof>

No match

```
lemma FI_p_nomatch:
  assumes "FI m ps (qs*q) (p*up) uq f"
  shows "FI m (ps*p) (qs*q) up uq f"
  <proof>
```

Head of q could not be matched

```
lemma FI_q_nomatch:
  assumes "FI m (SLN*up) qs SLN (q*uq) f"
  shows "FI m SLN (qs*q) up uq f"
  <proof>
```

5.4 Frame Inference

```
lemma frame_inference_init:
  assumes "FI_QUERY P Q F"
  shows "P  $\implies_A$  Q * F"
  <proof>
```

```
lemma frame_inference_finalize:
  shows "FI_RESULT M F emp F"
  <proof>
```

5.5 Entailment Solver

```
lemma entails_solve_init:
  "FI_QUERY P Q true  $\implies$  P  $\implies_A$  Q * true"
  "FI_QUERY P Q emp  $\implies$  P  $\implies_A$  Q"
  <proof>
```

```
lemma entails_solve_finalize:
  "FI_RESULT M P emp true"
  "FI_RESULT M emp emp emp"
  <proof>
```

```
lemmas solve_ent_preprocess_simps =
  ent_pure_post_iff ent_pure_post_iff_sng ent_pure_pre_iff ent_pure_pre_iff_sng
```

5.6 Verification Condition Generator

```
lemmas normalize_rules = norm_pre_ex_rule norm_pre_pure_rule
```

May be useful in simple, manual proofs, where the postcondition is no schematic variable.

```
lemmas return_cons_rule = cons_pre_rule[OF _ return_wp_rule]
```

Useful frame-rule variant for manual proof:

```

lemma frame_rule_left:
  "<P> c <Q>  $\implies$  <R * P> c < $\lambda x. R * Q x$ >"
  <proof>

lemmas deconstruct_rules =
  bind_rule if_rule false_rule return_sp_rule let_rule
  case_prod_rule case_list_rule case_option_rule case_sum_rule

lemmas heap_rules =
  ref_rule
  lookup_rule
  update_rule
  new_rule
  make_rule
  of_list_rule
  length_rule
  nth_rule
  upd_rule
  freeze_rule

lemma fi_rule:
  assumes CMD: "<P> c <Q>"
  assumes FRAME: " $P_s \implies_A P * F$ "
  shows "<P_s> c < $\lambda x. Q x * F$ >"
  <proof>

```

5.7 ML-setup

```

named_theorems sep_dflt_simps "Seplogic: Default simplification rules
for automated solvers"
named_theorems sep_eintros "Seplogic: Intro rules for entailment solver"
named_theorems sep_heap_rules "Seplogic: VCG heap rules"
named_theorems sep_decon_rules "Seplogic: VCG deconstruct rules"

<ML>

```

```

lemmas [sep_dflt_simps] = split

declare deconstruct_rules[sep_decon_rules]
declare heap_rules[sep_heap_rules]

lemmas [sep_eintros] = impI conjI exI

```

5.8 Semi-Automatic Reasoning

In this section, we provide some lemmas for semi-automatic reasoning

Forward reasoning with frame. Use *frame_inference*-method to discharge second assumption.

```

lemma ent_frame_fwd:
  assumes R: "P  $\implies_A$  R"
  assumes F: "Ps  $\implies_A$  P*F"
  assumes I: "R*F  $\implies_A$  Q"
  shows "Ps  $\implies_A$  Q"
  <proof>

```

```

lemma mod_frame_fwd:
  assumes M: "h  $\models$  Ps"
  assumes R: "P  $\implies_A$  R"
  assumes F: "Ps  $\implies_A$  P*F"
  shows "h  $\models$  R*F"
  <proof>

```

Apply precision rule with frame inference.

```

lemma prec_frame:
  assumes PREC: "precise P"
  assumes M1: "h  $\models$  (R1  $\wedge_A$  R2)"
  assumes F1: "R1  $\implies_A$  P x p * F1"
  assumes F2: "R2  $\implies_A$  P y p * F2"
  shows "x=y"
  <proof>

```

```

lemma prec_frame_expl:
  assumes PREC: " $\forall x y. (h \models (P x * F1) \wedge_A (P y * F2)) \longrightarrow x=y$ "
  assumes M1: "h  $\models$  (R1  $\wedge_A$  R2)"
  assumes F1: "R1  $\implies_A$  P x * F1"
  assumes F2: "R2  $\implies_A$  P y * F2"
  shows "x=y"
  <proof>

```

Variant that is useful within induction proofs, where induction goes over x or y

```

lemma prec_frame':
  assumes PREC: "(h  $\models$  (P x * F1)  $\wedge_A$  (P y * F2))  $\longrightarrow$  x=y"
  assumes M1: "h  $\models$  (R1  $\wedge_A$  R2)"
  assumes F1: "R1  $\implies_A$  P x * F1"
  assumes F2: "R2  $\implies_A$  P y * F2"
  shows "x=y"
  <proof>

```

```

lemma ent_wand_frameI:
  assumes "(Q -* R) * F  $\implies_A$  S"
  assumes "P  $\implies_A$  F * X"
  assumes "Q*X  $\implies_A$  R"
  shows "P  $\implies_A$  S"
  <proof>

```


simplify goals left over by verification condition generation or entailment solving.

(plain) Neither pre- nor postprocessing. Just applies vcg and entailment solver.

Entailment Solver. The entailment solver processes goals of the form $P \implies_A Q$. It is invoked by the method `solve_entails`. It first tries to pull out pure parts of P and Q . This may introduce quantifiers, conjunction, and implication into the goal, that are eliminated by resolving with rules declared as `sep_eintros` (method argument: `eintros[add/del]:`). Moreover, it simplifies with rules declared as `sep_dflt_simps` (section argument: `dflt_simps[add/del]:`).

Now, P and Q should have the form $X_1 * \dots * X_n$. Then, the frame-matcher is used to match all items of P with items of Q , and thus solve the implication. Matching is currently done syntactically, but can instantiate schematic variables.

Note that, by default, existential introduction is declared as `sep_eintros-rule`. This introduces schematic variables, that can later be matched against. However, in some cases, the matching may instantiate the schematic variables in an undesired way. In this case, the argument `eintros del: exI` should be passed to the entailment solver, and the existential quantifier should be instantiated manually.

Frame Inference The method `frame_inference` tries to solve a goal of the form $P \implies Q * ?F$, by matching Q against the parts of P , and instantiating $?F$ accordingly. Matching is done syntactically, possibly instantiating schematic variables. P and Q should be assertions separated by `*`. Note that frame inference does no simplification or other kinds of normalization.

The method `heap_rule` applies the specified heap rules, using frame inference if necessary. If no rules are specified, the default heap rules are used.

Verification Condition Generator The verification condition generator processes goals of the form $\langle P \rangle_c \langle Q \rangle$. It is invoked by the method `vcg`. First, it tries to pull out pure parts and simplifies with the default simplification rules. Then, it tries to resolve the goal with deconstruct rules (attribute: `sep_decon_rules`, section argument: `decon[add/del]:`), and if this does not succeed, it tries to resolve the goal with heap rules (attribute: `sep_heap_rules`, section argument: `heap[add/del]:`), using the frame rule and frame inference. If resolving is not possible, it also tries to apply the consequence rule to make the postcondition a schematic variable.

end

6 Separation Logic Framework Entrypoint

```
theory Sep_Main
imports Automation
begin
```

Import this theory to make available Imperative/HOL with separation logic.

```
end
```

7 Interface for Lists

```
theory Imp_List_Spec
imports "../Sep_Main"
begin
```

This file specifies an abstract interface for list data structures. It can be implemented by concrete list data structures, as demonstrated in the open and circular singly linked list examples.

```
locale imp_list =
  fixes is_list :: "'a list  $\Rightarrow$  'l  $\Rightarrow$  assn"
  assumes precise: "precise is_list"

locale imp_list_empty = imp_list +
  constrains is_list :: "'a list  $\Rightarrow$  'l  $\Rightarrow$  assn"
  fixes empty :: "'l Heap"
  assumes empty_rule[sep_heap_rules]: "<emp> empty <is_list []>t"

locale imp_list_is_empty = imp_list +
  constrains is_list :: "'a list  $\Rightarrow$  'l  $\Rightarrow$  assn"
  fixes is_empty :: "'l  $\Rightarrow$  bool Heap"
  assumes is_empty_rule[sep_heap_rules]:
    "<is_list l p> is_empty p < $\lambda$ r. is_list l p *  $\uparrow$ (r  $\longleftrightarrow$  l=[])>t"

locale imp_list_append = imp_list +
  constrains is_list :: "'a list  $\Rightarrow$  'l  $\Rightarrow$  assn"
  fixes append :: "'a  $\Rightarrow$  'l  $\Rightarrow$  'l Heap"
  assumes append_rule[sep_heap_rules]:
    "<is_list l p> append a p <is_list (l@[a])>t"

locale imp_list_prepend = imp_list +
  constrains is_list :: "'a list  $\Rightarrow$  'l  $\Rightarrow$  assn"
  fixes prepend :: "'a  $\Rightarrow$  'l  $\Rightarrow$  'l Heap"
  assumes prepend_rule[sep_heap_rules]:
    "<is_list l p> prepend a p <is_list (a#l)>t"

locale imp_list_head = imp_list +
  constrains is_list :: "'a list  $\Rightarrow$  'l  $\Rightarrow$  assn"
```



```

fixes head :: "'l ⇒ 'a Heap"
assumes head_rule[sep_heap_rules]:
  "l ≠ [] ⇒ <is_list l p> head p <λr. is_list l p * ↑(r=hd l)>t"

locale imp_list_pop = imp_list +
  constrains is_list :: "'a list ⇒ 'l ⇒ assn"
  fixes pop :: "'l ⇒ ('a × 'l) Heap"
  assumes pop_rule[sep_heap_rules]:
    "l ≠ [] ⇒
      <is_list l p>
      pop p
      <λ(r,p'). is_list (tl l) p' * ↑(r=hd l)>t"

locale imp_list_rotate = imp_list +
  constrains is_list :: "'a list ⇒ 'l ⇒ assn"
  fixes rotate :: "'l ⇒ 'l Heap"
  assumes rotate_rule[sep_heap_rules]:
    "<is_list l p> rotate p <is_list (rotate1 l)>t"

locale imp_list_reverse = imp_list +
  constrains is_list :: "'a list ⇒ 'l ⇒ assn"
  fixes reverse :: "'l ⇒ 'l Heap"
  assumes reverse_rule[sep_heap_rules]:
    "<is_list l p> reverse p <is_list (rev l)>t"

locale imp_list_iterate = imp_list +
  constrains is_list :: "'a list ⇒ 'l ⇒ assn"
  fixes is_it :: "'a list ⇒ 'l ⇒ 'a list ⇒ 'it ⇒ assn"
  fixes it_init :: "'l ⇒ ('it) Heap"
  fixes it_has_next :: "'it ⇒ bool Heap"
  fixes it_next :: "'it ⇒ ('a × 'it) Heap"
  assumes it_init_rule[sep_heap_rules]:
    "<is_list l p> it_init p <is_it l p l>t"
  assumes it_next_rule[sep_heap_rules]: "l' ≠ [] ⇒
    <is_it l p l' it>
    it_next it
    <λ(a,it'). is_it l p (tl l') it' * ↑(a=hd l')>t"
  assumes it_has_next_rule[sep_heap_rules]:
    "<is_it l p l' it>
    it_has_next it
    <λr. is_it l p l' it * ↑(r↔l' ≠ [])>t"
  assumes quit_iteration:
    "is_it l p l' it ⇒A is_list l p * true"

end

```

8 Singly Linked List Segments

theory *List_Seg*

```
imports "../Sep_Main"
begin
```

8.1 Nodes

We define a node of a list to contain a data value and a next pointer. As Imperative HOL does not support null-pointers, we make the next-pointer an optional value, *None* representing a null pointer.

Unfortunately, Imperative HOL requires some boilerplate code to define a datatype.

<ML>

```
datatype 'a node = Node "'a" "'a node ref option"
```

<ML>

Selector Functions

```
primrec val :: "'a::heap node ⇒ 'a" where
  [sep_dflt_simps]: "val (Node x _) = x"
```

```
primrec "next" :: "'a::heap node ⇒ 'a node ref option" where
  [sep_dflt_simps]: "next (Node _ r) = r"
```

Encoding to natural numbers, as required by Imperative/HOL

```
fun
  node_encode :: "'a::heap node ⇒ nat"
where
  "node_encode (Node x r) = to_nat (x, r)"
```

```
instance node :: (heap) heap
  <proof>
```

8.2 List Segment Assertion

Intuitively, *lseg l p s* describes a list starting at *p* and ending with a pointer *s*. The content of the list are *l*. Note that the pointer *s* may also occur earlier in the list, in which case it is handled as a usual next-pointer.

```
fun lseg
  :: "'a::heap list ⇒ 'a node ref option ⇒ 'a node ref option ⇒ assn"
  where
    "lseg [] p s = ↑(p=s)"
  | "lseg (x#l) (Some p) s = (∃ Aq. p ↦r Node x q * lseg l q s)"
  | "lseg (_#_) None _ = false"
```

```
lemma lseg_if_splitf1[simp, sep_dflt_simps]:
  "lseg l None None = ↑(l=[])"
  <proof>
```

```

lemma lseg_if_splitf2[simp, sep_dflt_simps]:
  "lseg (x#xs) p q
   = ( $\exists$  App n. pp  $\mapsto_r$  (Node x n) * lseg xs n q *  $\uparrow$ (p=Some pp))"
  <proof>

```

8.3 Lemmas

8.3.1 Concatenation

```

lemma lseg_prepend:
  "p $\mapsto_r$ Node x q * lseg l q s  $\implies_A$  lseg (x#l) (Some p) s"
  <proof>

```

```

lemma lseg_append:
  "lseg l p (Some s) * s $\mapsto_r$ Node x q  $\implies_A$  lseg (l@[x]) p q"
  <proof>

```

```

lemma lseg_conc: "lseg l1 p q * lseg l2 q r  $\implies_A$  lseg (l1@l2) p r"
  <proof>

```

8.3.2 Splitting

```

lemma lseg_split:
  "lseg (l1@l2) p r  $\implies_A$   $\exists_A$ q. lseg l1 p q * lseg l2 q r"
  <proof>

```

8.3.3 Precision

```

lemma lseg_prec1:
  " $\forall$  l l'. (h $\models$ 
    (lseg l p (Some q) * q  $\mapsto_r$  x * F1)
     $\wedge_A$  (lseg l' p (Some q) * q  $\mapsto_r$  x * F2))
   $\longrightarrow$  l=l'"
  <proof>

```

```

lemma lseg_prec2:
  " $\forall$  l l'. (h $\models$ 
    (lseg l p None * F1)  $\wedge_A$  (lseg l' p None * F2))
   $\longrightarrow$  l=l'"
  <proof>

```

```

lemma lseg_prec3:
  " $\forall$  q q'. h  $\models$  (lseg l p q * F1)  $\wedge_A$  (lseg l p q' * F2)  $\longrightarrow$  q=q'"
  <proof>

```

end

9 Open Singly Linked Lists

```
theory Open_List
imports List_Seg Imp_List_Spec
begin
```

9.1 Definitions

```
type_synonym 'a os_list = "'a node ref option"
```

```
abbreviation os_list :: "'a list  $\Rightarrow$  ('a::heap) os_list  $\Rightarrow$  assn" where
  "os_list l p  $\equiv$  lseg l p None"
```

9.2 Precision

```
lemma os_prec:
  "precise os_list"
  <proof>
```

```
lemma os_imp_list_impl: "imp_list os_list"
  <proof>
```

```
interpretation os: imp_list os_list <proof>
```

9.3 Operations

9.3.1 Allocate Empty List

```
definition os_empty :: "'a::heap os_list Heap" where
  "os_empty  $\equiv$  return None"
```

```
lemma os_empty_rule: "<emp> os_empty <os_list []>"
  <proof>
```

```
lemma os_empty_impl: "imp_list_empty os_list os_empty"
  <proof>
```

```
interpretation os: imp_list_empty os_list os_empty <proof>
```

9.3.2 Emptiness check

A linked list is empty, iff it is the null pointer.

```
definition os_is_empty :: "'a::heap os_list  $\Rightarrow$  bool Heap" where
  "os_is_empty b  $\equiv$  return (b = None)"
```

```
lemma os_is_empty_rule:
  "<os_list xs b> os_is_empty b < $\lambda$ r. os_list xs b *  $\uparrow$ (r  $\longleftrightarrow$  xs = [])>"
  <proof>
```

```
lemma os_is_empty_impl: "imp_list_is_empty os_list os_is_empty"
  <proof>
```

```
interpretation os: imp_list_is_empty os_list os_is_empty
```

<proof>

9.3.3 Prepend

To push an element to the front of a list we allocate a new node which stores the element and the old list as successor. The new list is the new allocated reference.

```
definition os_prepend :: "'a ⇒ 'a::heap os_list ⇒ 'a os_list Heap" where
  "os_prepend a n = do { p ← ref (Node a n); return (Some p) }"
```

```
lemma os_prepend_rule:
  "<os_list xs n> os_prepend x n <os_list (x # xs)>"
  <proof>
```

```
lemma os_prepend_impl: "imp_list_prepend os_list os_prepend"
  <proof>
```

```
interpretation os: imp_list_prepend os_list os_prepend
  <proof>
```

9.3.4 Pop

To pop the first element out of the list we look up the value and the reference of the node and return the pair of those.

```
fun os_pop :: "'a::heap os_list ⇒ ('a × 'a os_list) Heap" where
  "os_pop None = raise STR ''Empty Os_list''" |
  "os_pop (Some p) = do {m ← !p; return (val m, next m)}"
```

```
declare os_pop.simps[simp del]
```

```
lemma os_pop_rule:
  "xs ≠ [] ⇒ <os_list xs r>
  os_pop r
  <λ(x,r'). os_list (tl xs) r' * (the r) ↦r (Node x r') * ↑(x = hd xs)>"
  <proof>
```

```
lemma os_pop_impl: "imp_list_pop os_list os_pop"
  <proof>
```

```
interpretation os: imp_list_pop os_list os_pop <proof>
```

9.3.5 Reverse

The following reversal function is equivalent to the one from Imperative HOL. And gives a more difficult example.

```
partial_function (heap) os_reverse_aux
  :: "'a::heap os_list ⇒ 'a os_list ⇒ 'a os_list Heap"
  where [code]:
```

```
"os_reverse_aux q p = (case p of
  None ⇒ return q |
  Some r ⇒ do {
    v ← !r;
    r := Node (val v) q;
    os_reverse_aux p (next v) })"
```

```
lemma [simp, sep_dflt_simps]:
  "os_reverse_aux q None = return q"
  "os_reverse_aux q (Some r) = do {
    v ← !r;
    r := Node (val v) q;
    os_reverse_aux (Some r) (next v) }"
  ⟨proof⟩
```

```
definition "os_reverse p = os_reverse_aux None p"
```

```
lemma os_reverse_aux_rule:
  "<os_list xs p * os_list ys q>
  os_reverse_aux q p
  <os_list ((rev xs) @ ys) >"
  ⟨proof⟩
```

```
lemma os_reverse_rule: "<os_list xs p> os_reverse p <os_list (rev xs)>"
  ⟨proof⟩
```

```
lemma os_reverse_impl: "imp_list_reverse os_list os_reverse"
  ⟨proof⟩
```

```
interpretation os: imp_list_reverse os_list os_reverse
  ⟨proof⟩
```

9.3.6 Remove

Remove all appearances of an element from a linked list.

```
partial_function (heap) os_rem
  :: "'a::heap ⇒ 'a node ref option ⇒ 'a node ref option Heap"
  where [code]:
    "os_rem x b = (case b of
      None ⇒ return None |
      Some p ⇒ do {
        n ← !p;
        q ← os_rem x (next n);
        (if (val n = x)
          then return q
          else do {
            p := Node (val n) q;
            return (Some p) }) })"
```

```
lemma [simp, sep_dflt_simps]:
```

```

"os_rem x None = return None"
"os_rem x (Some p) = do {
  n ← !p;
  q ← os_rem x (next n);
  (if (val n = x)
    then return q
    else do {
      p := Node (val n) q;
      return (Some p) }) }"
⟨proof⟩

```

```

lemma os_rem_rule[sep_heap_rules]:
  "<os_list xs b> os_rem x b <λr. os_list (removeAll x xs) r * true>"
⟨proof⟩

```

```

lemma os_rem_rule_alt_proof:
  "<os_list xs b> os_rem x b <λr. os_list (removeAll x xs) r * true>"
⟨proof⟩

```

9.3.7 Iterator

```

type_synonym 'a os_list_it = "'a os_list"

```

```

definition "os_is_it l p l2 it
  ≡ ∃_A l1. ↑(l=l1@l2) * lseg l1 p it * os_list l2 it"

```

```

definition os_it_init :: "'a os_list ⇒ ('a os_list_it) Heap"
  where "os_it_init l = return l"

```

```

fun os_it_next where
  "os_it_next (Some p) = do {
    n ← !p;
    return (val n,next n)
  }"

```

```

definition os_it_has_next :: "'a os_list_it ⇒ bool Heap" where
  "os_it_has_next it ≡ return (it≠None)"

```

```

lemma os_iterate_impl:
  "imp_list_iterate os_list os_is_it os_it_init os_it_has_next os_it_next"
⟨proof⟩

```

```

interpretation os:
  imp_list_iterate os_list os_is_it os_it_init os_it_has_next os_it_next
⟨proof⟩

```

9.3.8 List-Sum

```

partial_function (heap) os_sum' :: "int os_list_it ⇒ int ⇒ int Heap"

```

```

where [code]:
  "os_sum' it s = do {

```

```

    b ← os_it_has_next it;
    if b then do {
      (x,it') ← os_it_next it;
      os_sum' it' (s+x)
    } else return s
  }"

```

```

lemma os_sum'_rule[sep_heap_rules]:
  "<os_is_it l p l' it>
   os_sum' it s
  < $\lambda r. \text{os\_list } l \text{ p} * \uparrow(r = s + \text{sum\_list } l')$ >t"
<proof>

```

```

definition "os_sum p  $\equiv$  do {
  it ← os_it_init p;
  os_sum' it 0}"

```

```

lemma os_sum_rule[sep_heap_rules]:
  "<os_list l p> os_sum p < $\lambda r. \text{os\_list } l \text{ p} * \uparrow(r = \text{sum\_list } l)$ >t"
<proof>

```

end

10 Circular Singly Linked Lists

```

theory Circ_List
imports List_Seg Imp_List_Spec
begin

```

Example of circular lists, with efficient append, prepend, pop, and rotate operations.

10.1 Datatype Definition

```

type_synonym 'a cs_list = "'a node ref option"

```

A circular list is described by a list segment, with special cases for the empty list:

```

fun cs_list :: "'a::heap list  $\Rightarrow$  'a node ref option  $\Rightarrow$  assn" where
  "cs_list [] None = emp"
| "cs_list (x#l) (Some p) = lseg (x#l) (Some p) (Some p)"
| "cs_list _ _ = false"

```

```

lemma [simp]: "cs_list l None =  $\uparrow(l = [])$ "
<proof>

```

```

lemma [simp]:
  "cs_list l (Some p)

```



```
= (∃A x ls. ↑(l=x#ls) * lseg (x#ls) (Some p) (Some p))"
⟨proof⟩
```

10.2 Precision

```
lemma cs_prec:
  "precise cs_list"
  ⟨proof⟩
```

```
lemma cs_imp_list_impl: "imp_list cs_list"
  ⟨proof⟩
```

```
interpretation cs: imp_list cs_list ⟨proof⟩
```

10.3 Operations

10.3.1 Allocate Empty List

```
definition cs_empty :: "'a::heap cs_list Heap" where
  "cs_empty ≡ return None"
```

```
lemma cs_empty_rule: "<emp> cs_empty <cs_list []>"
  ⟨proof⟩
```

```
lemma cs_empty_impl: "imp_list_empty cs_list cs_empty"
  ⟨proof⟩
```

```
interpretation cs: imp_list_empty cs_list cs_empty ⟨proof⟩
```

10.3.2 Prepend Element

```
fun cs_prepend :: "'a ⇒ 'a::heap cs_list ⇒ 'a cs_list Heap" where
  "cs_prepend x None = do {
    p ← ref (Node x None);
    p := Node x (Some p);
    return (Some p)
  }"
| "cs_prepend x (Some p) = do {
    n ← !p;
    q ← ref (Node (val n) (next n));
    p := Node x (Some q);
    return (Some p)
  }"
```

```
declare cs_prepend.simps [simp del]
```

```
lemma cs_prepend_rule:
  "<cs_list l p> cs_prepend x p <cs_list (x#l)>"
  ⟨proof⟩
```

```
lemma cs_prepend_impl: "imp_list_prepend cs_list cs_prepend"
  ⟨proof⟩
```

```
interpretation cs: imp_list_prepend cs_list cs_prepend
  <proof>
```

10.3.3 Append Element

```
fun cs_append :: "'a ⇒ 'a::heap cs_list ⇒ 'a cs_list Heap" where
  "cs_append x None = do {
    p ← ref (Node x None);
    p := Node x (Some p);
    return (Some p) }"
| "cs_append x (Some p) = do {
  n ← !p;
  q ← ref (Node (val n) (next n));
  p := Node x (Some q);
  return (Some q)
}"
```

```
declare cs_append.simps [simp del]
```

```
lemma cs_append_rule:
  "<cs_list l p> cs_append x p <cs_list (l@[x])>"
  <proof>
```

```
lemma cs_append_impl: "imp_list_append cs_list cs_append"
  <proof>
```

```
interpretation cs: imp_list_append cs_list cs_append
  <proof>
```

10.3.4 Pop First Element

```
fun cs_pop :: "'a::heap cs_list ⇒ ('a × 'a cs_list) Heap" where
  "cs_pop None = raise STR ''Pop from empty list''"
| "cs_pop (Some p) = do {
  n1 ← !p;
  if next n1 = Some p then
    return (val n1, None) — Singleton list becomes empty list
  else do {
    let p2 = the (next n1);
    n2 ← !p2;
    p := Node (val n2) (next n2);
    return (val n1, Some p)
  }
}"
```

```
declare cs_pop.simps [simp del]
```

```
lemma cs_pop_rule:
  "<cs_list (x#l) p> cs_pop p <λ(y,p'). cs_list l p' * true * ↑(y=x)>"
  <proof>
```

```

lemma cs_pop_impl: "imp_list_pop cs_list cs_pop"
  ⟨proof⟩
interpretation cs: imp_list_pop cs_list cs_pop ⟨proof⟩

```

10.3.5 Rotate

```

fun cs_rotate :: "'a::heap cs_list ⇒ 'a cs_list Heap" where
  "cs_rotate None = return None"
| "cs_rotate (Some p) = do {
  n ← !p;
  return (next n)
}"

```

```

declare cs_rotate.simps [simp del]

```

```

lemma cs_rotate_rule:
  "<cs_list l p> cs_rotate p <cs_list (rotate1 l)>"
  ⟨proof⟩

```

```

lemma cs_rotate_impl: "imp_list_rotate cs_list cs_rotate"
  ⟨proof⟩
interpretation cs: imp_list_rotate cs_list cs_rotate ⟨proof⟩

```

10.4 Test

```

definition "test ≡ do {
  l ← cs_empty;
  l ← cs_append ''a'' l;
  l ← cs_append ''b'' l;
  l ← cs_append ''c'' l;
  l ← cs_prepend ''0'' l;
  l ← cs_rotate l;
  (v1,l)←cs_pop l;
  (v2,l)←cs_pop l;
  (v3,l)←cs_pop l;
  (v4,l)←cs_pop l;
  return [v1,v2,v3,v4]
}"

```

```

definition "test_result ≡ [''a'', ''b'', ''c'', ''0'']"

```

```

lemma "<emp> test <λr. ↑(r=test_result) * true>"
  ⟨proof⟩

```

```

export_code test checking SML_imp

```

```

⟨ML⟩

```

```

hide_const (open) test test_result

```

end

11 Interface for Maps

```
theory Imp_Map_Spec
imports "../Sep_Main"
begin
```

This file specifies an abstract interface for map data structures. It can be implemented by concrete map data structures, as demonstrated in the hash map example.

```
locale imp_map =
  fixes is_map :: "('k  $\rightarrow$  'v)  $\Rightarrow$  'm  $\Rightarrow$  assn"
  assumes precise: "precise is_map"

locale imp_map_empty = imp_map +
  constrains is_map :: "('k  $\rightarrow$  'v)  $\Rightarrow$  'm  $\Rightarrow$  assn"
  fixes empty :: "'m Heap"
  assumes empty_rule[sep_heap_rules]: "<emp> empty <is_map Map.empty>t"

locale imp_map_is_empty = imp_map +
  constrains is_map :: "('k  $\rightarrow$  'v)  $\Rightarrow$  'm  $\Rightarrow$  assn"
  fixes is_empty :: "'m  $\Rightarrow$  bool Heap"
  assumes is_empty_rule[sep_heap_rules]:
    "<is_map m p> is_empty p < $\lambda$ r. is_map m p *  $\uparrow$ (r  $\longleftrightarrow$  m=Map.empty)>t"

locale imp_map_lookup = imp_map +
  constrains is_map :: "('k  $\rightarrow$  'v)  $\Rightarrow$  'm  $\Rightarrow$  assn"
  fixes lookup :: "'k  $\Rightarrow$  'm  $\Rightarrow$  ('v option) Heap"
  assumes lookup_rule[sep_heap_rules]:
    "<is_map m p> lookup k p < $\lambda$ r. is_map m p *  $\uparrow$ (r = m k)>t"

locale imp_map_update = imp_map +
  constrains is_map :: "('k  $\rightarrow$  'v)  $\Rightarrow$  'm  $\Rightarrow$  assn"
  fixes update :: "'k  $\Rightarrow$  'v  $\Rightarrow$  'm  $\Rightarrow$  'm Heap"
  assumes update_rule[sep_heap_rules]:
    "<is_map m p> update k v p <is_map (m(k  $\mapsto$  v))>t"

locale imp_map_delete = imp_map +
  constrains is_map :: "('k  $\rightarrow$  'v)  $\Rightarrow$  'm  $\Rightarrow$  assn"
  fixes delete :: "'k  $\Rightarrow$  'm  $\Rightarrow$  'm Heap"
  assumes delete_rule[sep_heap_rules]:
    "<is_map m p> delete k p <is_map (m |' (-{k})>>t"

locale imp_map_add = imp_map +
  constrains is_map :: "('k  $\rightarrow$  'v)  $\Rightarrow$  'm  $\Rightarrow$  assn"
  fixes add :: "'m  $\Rightarrow$  'm  $\Rightarrow$  'm Heap"
  assumes add_rule[sep_heap_rules]:
```

```

"<is_map m p * is_map m' p'> add p p'
  <\r. is_map m p * is_map m' p' * is_map (m ++ m') r>_t"

locale imp_map_size = imp_map +
  constrains is_map :: "('k  $\rightarrow$  'v)  $\Rightarrow$  'm  $\Rightarrow$  assn"
  fixes size :: "'m  $\Rightarrow$  nat Heap"
  assumes size_rule[sep_heap_rules]:
    "<is_map m p> size p <\r. is_map m p *  $\uparrow$ (r = card (dom m))>_t"

locale imp_map_iterate = imp_map +
  constrains is_map :: "('k  $\rightarrow$  'v)  $\Rightarrow$  'm  $\Rightarrow$  assn"
  fixes is_it :: "('k  $\rightarrow$  'v)  $\Rightarrow$  'm  $\Rightarrow$  ('k  $\rightarrow$  'v)  $\Rightarrow$  'it  $\Rightarrow$  assn"
  fixes it_init :: "'m  $\Rightarrow$  ('it) Heap"
  fixes it_has_next :: "'it  $\Rightarrow$  bool Heap"
  fixes it_next :: "'it  $\Rightarrow$  (('k  $\times$  'v)  $\times$  'it) Heap"
  assumes it_init_rule[sep_heap_rules]:
    "<is_map s p> it_init p <is_it s p s>_t"
  assumes it_next_rule[sep_heap_rules]: "m'  $\neq$  Map.empty  $\Rightarrow$ 
    <is_it m p m' it>
      it_next it
      < $\lambda$ ((k,v),it'). is_it m p (m' |' (-{k})) it' *  $\uparrow$ (m' k = Some v)>_t"
  assumes it_has_next_rule[sep_heap_rules]:
    "<is_it m p m' it> it_has_next it <\r. is_it m p m' it *  $\uparrow$ (r  $\longleftrightarrow$  m'  $\neq$  Map.empty)>_t"
  assumes quit_iteration:
    "is_it m p m' it  $\Rightarrow_A$  is_map m p * true"

locale imp_map_iterate' = imp_map +
  constrains is_map :: "('k  $\rightarrow$  'v)  $\Rightarrow$  'm  $\Rightarrow$  assn"
  fixes is_it :: "('k  $\rightarrow$  'v)  $\Rightarrow$  'm  $\Rightarrow$  ('k  $\times$  'v) list  $\Rightarrow$  'it  $\Rightarrow$  assn"
  fixes it_init :: "'m  $\Rightarrow$  ('it) Heap"
  fixes it_has_next :: "'it  $\Rightarrow$  bool Heap"
  fixes it_next :: "'it  $\Rightarrow$  (('k  $\times$  'v)  $\times$  'it) Heap"
  assumes it_init_rule[sep_heap_rules]:
    "<is_map s p> it_init p <\r.  $\exists_{A1} l. \uparrow$ (map_of l = s) * is_it s p l r>_t"
  assumes it_next_rule[sep_heap_rules]: "
    <is_it m p (kv#l) it>
      it_next it
      < $\lambda$ (kv',it'). is_it m p l it' *  $\uparrow$ (kv'=kv)>"
  assumes it_has_next_rule[sep_heap_rules]:
    "<is_it m p l it> it_has_next it <\r. is_it m p l it *  $\uparrow$ (r  $\longleftrightarrow$  l  $\neq$  [])>"
  assumes quit_iteration:
    "is_it m p l it  $\Rightarrow_A$  is_map m p * true"

end

```

12 Hash-Tables

theory Hash_Table

```

imports
  Collections.HashCode
  Collections.Code_Target_ICF
  "../Sep_Main"
begin

12.1 Datatype

12.1.1 Definition

datatype ('k, 'v) hashtable = HashTable "('k × 'v) list array" nat

primrec the_array :: "('k, 'v) hashtable ⇒ ('k × 'v) list array"
  where "the_array (HashTable a _) = a"

primrec the_size :: "('k, 'v) hashtable ⇒ nat"
  where "the_size (HashTable _ n) = n"

12.1.2 Storable on Heap

fun hs_encode :: "('k::countable, 'v::countable) hashtable ⇒ nat"
  where "hs_encode (HashTable a n) = to_nat (n, a)"

instance hashtable :: (countable, countable) countable
  ⟨proof⟩

instance hashtable :: (heap, heap) heap ⟨proof⟩

12.2 Assertions

12.2.1 Assertion for Hashtable

definition ht_table :: "('k::heap × 'v::heap) list list ⇒ ('k, 'v) hashtable
  ⇒ assn"
  where "ht_table l ht = (the_array ht) ↦a l"

definition ht_size :: "'a list list ⇒ nat ⇒ bool"
  where "ht_size l n ≡ n = sum_list (map length l)"

definition ht_hash :: "('k::hashable × 'v) list list ⇒ bool" where
  "ht_hash l ≡ ∀ i < length l. ∀ x ∈ set (l!i).
    bounded_hashcode_nat (length l) (fst x) = i"

definition ht_distinct :: "('k × 'v) list list ⇒ bool" where
  "ht_distinct l ≡ ∀ i < length l. distinct (map fst (l!i))"

definition is_hashtable :: "('k::{heap,hashable} × 'v::heap) list list
  ⇒ ('k, 'v) hashtable ⇒ assn"
  where

```

```

"is_hashtable l ht =
(the_array ht ↦a l) *
↑(ht_size l (the_size ht)
  ^ ht_hash l
  ^ ht_distinct l
  ^ 1 < length l)"

```

lemma *is_hashtable_prec*: "precise is_hashtable"
 ⟨*proof*⟩

These rules are quite useful for automated methods, to avoid unfolding of definitions, that might be used folded in other lemmas, like induction hypothesis. However, they show in some sense a possibility for modularization improvement, as it should be enough to show an implication and know that the *nth* and *len* operations do not change the heap.

lemma *ht_array_nth_rule*[*sep_heap_rules*]:
 "i < length l ⇒ <is_hashtable l ht>
 Array.nth (the_array ht) i
 <λr. is_hashtable l ht * ↑(r = l!i)>"
 ⟨*proof*⟩

lemma *ht_array_length_rule*[*sep_heap_rules*]:
 "<is_hashtable l ht>
 Array.len (the_array ht)
 <λr. is_hashtable l ht * ↑(r = length l)>"
 ⟨*proof*⟩

12.3 New

12.3.1 Definition

definition *ht_new_sz* :: "nat ⇒ ('k::{heap,hashable}, 'v::heap) hashtable Heap"
 where
 "ht_new_sz n ≡ do { let l = replicate n [];
 a ← Array.of_list l;
 return (HashTable a 0) }"

definition *ht_new* :: "('k::{heap,hashable}, 'v::heap) hashtable Heap"
 where "ht_new ≡ ht_new_sz (def_hashmap_size TYPE('k))"

12.3.2 Complete Correctness

lemma *ht_hash_replicate*[*simp*, *intro!*]: "ht_hash (replicate n [])"
 ⟨*proof*⟩

lemma *ht_distinct_replicate*[*simp*, *intro!*]: "ht_distinct (replicate n [])"
 ⟨*proof*⟩

<proof>

lemma *ht_size_replicate*[simp, intro!]: "ht_size (replicate n []) 0"

<proof>

lemma *complete_ht_new_sz*: "1 < n \implies *<emp>* ht_new_sz n *<is_hashtable* (replicate n [])>"

<proof>

lemma *complete_ht_new*:

"<emp>

ht_new::('k::{heap,hashable}, 'v::heap) hashtable Heap

<is_hashtable (replicate (def_hashmap_size TYPE('k)) [])>"

<proof>

12.4 Lookup

12.4.1 Definition

fun *ls_lookup* :: "'k \implies ('k \times 'v) list \implies 'v option"

where

"ls_lookup x [] = None" |

"ls_lookup x ((k, v) # l) = (if x = k then Some v else ls_lookup x l)"

definition *ht_lookup* :: "'k \implies ('k::{heap,hashable}, 'v::heap) hashtable \implies 'v option Heap"

where

"ht_lookup x ht = do {

m \leftarrow Array.len (the_array ht);

let i = bounded_hashcode_nat m x;

l \leftarrow Array.nth (the_array ht) i;

return (ls_lookup x l)

}"

12.4.2 Complete Correctness

lemma *complete_ht_lookup*:

"<is_hashtable l ht> ht_lookup x ht

*< λ r. is_hashtable l ht **

\uparrow (r = ls_lookup x (l!(bounded_hashcode_nat (length l) x))) >"

<proof>

Alternative, more automatic proof

lemma *complete_ht_lookup_alt_proof*:

"<is_hashtable l ht> ht_lookup x ht

*< λ r. is_hashtable l ht **

\uparrow (r = ls_lookup x (l!(bounded_hashcode_nat (length l) x)))>"

<proof>

12.5 Update

12.5.1 Definition

```
fun ls_update :: "'k ⇒ 'v ⇒ ('k × 'v) list ⇒ (('k × 'v) list × bool)"
where
  "ls_update k v [] = [(k, v)], False" |
  "ls_update k v ((l, w) # ls) = (
    if k = l then
      ((k, v) # ls, True)
    else
      (let r = ls_update k v ls in ((l, w) # fst r, snd r))
  )"

```

definition *abs_update*

```
:: "'k::hashable ⇒ 'v ⇒ ('k × 'v) list list ⇒ ('k × 'v) list list"
where
  "abs_update k v l =
    l[bounded_hashcode_nat (length l) k
      := fst (ls_update k v (l ! bounded_hashcode_nat (length l) k))]"

```

lemma *ls_update_snd_set*: "snd (ls_update k v l) \longleftrightarrow k \in set (map fst l)"

<proof>

lemma *ls_update_fst_set*: "set (fst (ls_update k v l)) \subseteq insert (k, v) (set l)"

<proof>

lemma *ls_update_fst_map_set*: "set (map fst (fst (ls_update k v l))) = insert k (set (map fst l))"

<proof>

lemma *ls_update_distinct*: "distinct (map fst l) \implies distinct (map fst (fst (ls_update k v l)))"

<proof>

lemma *ls_update_length*: "length (fst (ls_update k v l)) = (if k \in set (map fst l) then length l else Suc (length l))"

<proof>

lemma *ls_update_length_snd_True*:

```
"snd (ls_update k v l)  $\implies$  length (fst (ls_update k v l)) = length l"
```

<proof>

lemma *ls_update_length_snd_False*:

```
" $\neg$  snd (ls_update k v l)  $\implies$  length (fst (ls_update k v l)) = Suc (length l)"
```

<proof>

```

definition ht_upd
  :: "'k ⇒ 'v
    ⇒ ('k::{heap,hashable}, 'v::heap) hashtable
    ⇒ ('k, 'v) hashtable Heap"
where
  "ht_upd k v ht = do {
    m ← Array.len (the_array ht);
    let i = bounded_hashcode_nat m k;
    l ← Array.nth (the_array ht) i;
    let l = ls_update k v l;
    Array.upd i (fst l) (the_array ht);
    let n = (if (snd l) then the_size ht else Suc (the_size ht));
    return (HashTable (the_array ht) n)
  }"

```

12.5.2 Complete Correctness

```

lemma ht_hash_update:
  assumes "ht_hash ls"
  shows "ht_hash (abs_update k v ls)"
  <proof>

```

```

lemma ht_distinct_update:
  assumes "ht_distinct l"
  shows "ht_distinct (abs_update k v l)"
  <proof>

```

```

lemma length_update:
  assumes "1 < length l"
  shows "1 < length (abs_update k v l)"
  <proof>

```

```

lemma ht_size_update1:
  assumes size: "ht_size l n"
  assumes i: "i < length l"
  assumes snd: "snd (ls_update k v (l ! i))"
  shows "ht_size (l[i := fst (ls_update k v (l!i))]) n"
  <proof>

```

```

lemma ht_size_update2:
  assumes size: "ht_size l n"
  assumes i: "i < length l"
  assumes snd: "¬ snd (ls_update k v (l ! i))"
  shows "ht_size (l[i := fst (ls_update k v (l!i))]) (Suc n)"
  <proof>

```

```

lemma complete_ht_upd: "<is_hashtable l ht> ht_upd k v ht
  <is_hashtable (abs_update k v l)>"
  <proof>

```

Alternative, more automatic proof

```

lemma complete_ht_upd_alt_proof:
  "<is_hashtable l ht> ht_upd k v ht <is_hashtable (abs_update k v l)>"
  <proof>

```

12.6 Delete

12.6.1 Definition

```

fun ls_delete :: "'k ⇒ ('k × 'v) list ⇒ (('k × 'v) list × bool)" where
  "ls_delete k [] = ([], False)" |
  "ls_delete k ((l, w) # ls) = (
    if k = l then
      (ls, True)
    else
      (let r = ls_delete k ls in ((l, w) # fst r, snd r)))"

```

```

lemma ls_delete_snd_set: "snd (ls_delete k l) ⟷ k ∈ set (map fst l)"
  <proof>

```

```

lemma ls_delete_fst_set: "set (fst (ls_delete k l)) ⊆ set l"
  <proof>

```

```

lemma ls_delete_fst_map_set:
  "distinct (map fst l) ⇒
  set (map fst (fst (ls_delete k l))) = (set (map fst l)) - {k}"
  <proof>

```

```

lemma ls_delete_distinct: "distinct (map fst l) ⇒ distinct (map fst
  (fst (ls_delete k l)))"
  <proof>

```

```

lemma ls_delete_length:
  "length (fst (ls_delete k l)) = (
    if (k ∈ set (map fst l)) then
      (length l - 1)
    else
      length l)"
  <proof>

```

```

lemma ls_delete_length_snd_True:
  "snd (ls_delete k l) ⇒ length (fst (ls_delete k l)) = length l - 1"
  <proof>

```

```

lemma ls_delete_length_snd_False:

```

```
"¬ snd (ls_delete k l) ==> length (fst (ls_delete k l)) = length l"
⟨proof⟩
```

definition ht_delete

```
:: "'k
  => ('k::{heap,hashable}, 'v::heap) hashtable
  => ('k, 'v) hashtable Heap"
where
"ht_delete k ht = do {
  m ← Array.len (the_array ht);
  let i = bounded_hashcode_nat m k;
  l ← Array.nth (the_array ht) i;
  let l = ls_delete k l;
  Array.upd i (fst l) (the_array ht);
  let n = (if (snd l) then (the_size ht - 1) else the_size ht);
  return (HashTable (the_array ht) n)
}"
```

12.6.2 Complete Correctness

lemma ht_hash_delete:

```
assumes "ht_hash ls"
shows "ht_hash (
  ls[bounded_hashcode_nat (length ls) k
    := fst (ls_delete k
      (ls ! bounded_hashcode_nat (length ls) k)
    )
  ]
)"
⟨proof⟩
```

lemma ht_distinct_delete:

```
assumes "ht_distinct l"
shows "ht_distinct (
  l[bounded_hashcode_nat (length l) k
    := fst (ls_delete k (l ! bounded_hashcode_nat (length l) k))]
)"
⟨proof⟩
```

lemma ht_size_delete1:

```
assumes size: "ht_size l n"
assumes i: "i < length l"
assumes snd: "snd (ls_delete k (l ! i))"
shows "ht_size (l[i := fst (ls_delete k (l!i))]) (n - 1)"
⟨proof⟩
```

lemma ht_size_delete2:

```
assumes size: "ht_size l n"
assumes i: "i < length l"
```

```

assumes snd: "¬ snd (ls_delete k (l ! i))"
shows "ht_size (l[i := fst (ls_delete k (l!i))]) n"
<proof>

```

```

lemma complete_ht_delete: "<is_hashtable l ht> ht_delete k ht
  <is_hashtable (l[bounded_hashcode_nat (length l) k
    := fst (ls_delete k (l ! bounded_hashcode_nat (length l) k)])>"
  <proof>

```

Alternative, more automatic proof

```

lemma "<is_hashtable l ht> ht_delete k ht
  <is_hashtable (l[bounded_hashcode_nat (length l)
    k := fst (ls_delete k (l ! bounded_hashcode_nat (length l) k)])>"
  <proof>

```

12.7 Re-Hashing

12.7.1 Auxiliary Functions

```

fun ht_insls
  :: "('k × 'v) list
    ⇒ ('k::{heap,hashable}, 'v::heap) hashtable
    ⇒ ('k, 'v::heap) hashtable Heap"
where
  "ht_insls [] ht = return ht" |
  "ht_insls ((k, v) # l) ht = do { h ← ht_upd k v ht; ht_insls l h }"

```

Abstract version

```

fun ls_insls :: "('k::hashable × 'v) list
  ⇒ ('k × 'v) list list ⇒ ('k × 'v) list list"
where
  "ls_insls [] l = l" |
  "ls_insls ((k, v) # ls) l =
    ls_insls ls (abs_update k v l)"

```

```

lemma ht_hash_ls_insls:
  assumes "ht_hash l"
  shows "ht_hash (ls_insls ls l)"
  <proof>

```

```

lemma ht_distinct_ls_insls:
  assumes "ht_distinct l"
  shows "ht_distinct (ls_insls ls l)"
  <proof>

```

```

lemma length_ls_insls:
  assumes "1 < length l"
  shows "1 < length (ls_insls ls l)"

```

<proof>

lemma *complete_ht_insls*:

"<*is_hashtable* *ls ht*> *ht_insls xs ht <is_hashtable (ls_insls xs ls)>*"
<proof>

fun *ht_copy* :: "nat \Rightarrow ('k::{heap,hashable}, 'v::heap) hashtable
 \Rightarrow ('k, 'v) hashtable \Rightarrow ('k, 'v) hashtable Heap"
where
"ht_copy 0 src dst = return dst" |
"ht_copy (Suc n) src dst = do {
 l \leftarrow Array.nth (the_array src) n;
 ht \leftarrow ht_insls *l* dst;
 ht_copy n src ht
}"

Abstract version

fun *ls_copy* :: "nat \Rightarrow ('k::hashable \times 'v) list list
 \Rightarrow ('k \times 'v) list list \Rightarrow ('k \times 'v) list list"
where
"ls_copy 0 ss ds = ds" |
"ls_copy (Suc n) ss ds = ls_copy n ss (ls_insls (ss ! n) ds)"

lemma *ht_hash_ls_copy*:
assumes "ht_hash *l*"
shows "ht_hash (ls_copy n ss *l*)"
<proof>

lemma *ht_distinct_ls_copy*:
assumes "ht_distinct *l*"
shows "ht_distinct (ls_copy n ss *l*)"
<proof>

lemma *length_ls_copy*:
assumes "1 < length *l*"
shows "1 < length (ls_copy n ss *l*)"
<proof>

lemma *complete_ht_copy*: "n \leq List.length ss \implies
<*is_hashtable* *ss src* * *is_hashtable* *ds dst*>
ht_copy n *src* *dst*
< λ r. *is_hashtable* *ss src* * *is_hashtable* (ls_copy n ss ds) *r*>"
<proof>

Alternative, more automatic proof

lemma *complete_ht_copy_alt_proof*: "n \leq List.length ss \implies

```

<is_hashtable ss src * is_hashtable ds dst>
ht_copy n src dst
<\r. is_hashtable ss src * is_hashtable (ls_copy n ss ds) r>"
<proof>

```

```

definition ht_rehash
  :: "('k::{heap,hashable}, 'v::heap) hashtable  $\Rightarrow$  ('k, 'v) hashtable
  Heap"
  where
    "ht_rehash ht = do {
      n  $\leftarrow$  Array.len (the_array ht);
      h  $\leftarrow$  ht_new_sz (2 * n);
      ht_copy n ht h
    }"

```

Operation on Abstraction

```

definition ls_rehash :: "('k::hashable  $\times$  'v) list list  $\Rightarrow$  ('k  $\times$  'v) list
  list"
  where "ls_rehash l = ls_copy (List.length l) l (replicate (2 * length
  l) [])"

```

```

lemma ht_hash_ls_rehash: "ht_hash (ls_rehash l)"
  <proof>

```

```

lemma ht_distinct_ls_rehash: "ht_distinct (ls_rehash l)"
  <proof>

```

```

lemma length_ls_rehash:
  assumes "1 < length l"
  shows "1 < length (ls_rehash l)"
  <proof>

```

```

lemma ht_imp_len: "is_hashtable l ht  $\Rightarrow_A$  is_hashtable l ht *  $\uparrow$ (length
  l > 0)"
  <proof>

```

```

lemma complete_ht_rehash:
  "<is_hashtable l ht> ht_rehash ht
  <\r. is_hashtable l ht * is_hashtable (ls_rehash l) r>"
  <proof>

```

```

definition load_factor :: nat — in percent
  where "load_factor = 75"

```

```

definition ht_update
  :: "'k::{heap,hashable}  $\Rightarrow$  'v::heap  $\Rightarrow$  ('k, 'v) hashtable
   $\Rightarrow$  ('k, 'v) hashtable Heap"
  where
    "ht_update k v ht = do {

```

```

    m ← Array.len (the_array ht);
    ht ← (if m * load_factor ≤ (the_size ht) * 100 then
          ht_rehash ht
        else return ht);
    ht_upd k v ht
  }"

```

```

lemma complete_ht_update_normal:
  "¬ length l * load_factor ≤ (the_size ht)* 100 ⇒
  <is_hashtable l ht>
  ht_update k v ht
  <is_hashtable (abs_update k v l)>"
  <proof>

```

```

lemma complete_ht_update_rehash:
  "length l * load_factor ≤ (the_size ht)* 100 ⇒
  <is_hashtable l ht>
  ht_update k v ht
  <λr. is_hashtable l ht
    * is_hashtable (abs_update k v (ls_rehash l)) r>"
  <proof>

```

12.8 Conversion to List

```

definition ht_to_list ::
  "('k::heap, 'v::heap) hashtable ⇒ ('k × 'v) list Heap" where
  "ht_to_list ht = do {
    l ← (Array.freeze (the_array ht));
    return (concat l)
  }"

```

```

lemma complete_ht_to_list: "<is_hashtable l ht> ht_to_list ht
  <λr. is_hashtable l ht * ↑(r = concat l)>"
  <proof>

```

end
Documentation

13 Hash-Maps

```

theory Hash_Map
  imports Hash_Table
begin

```

13.1 Auxiliary Lemmas

```

lemma map_of_ls_update:
  "map_of (fst (ls_update k v l)) = (map_of l)(k ↦ v)"

```


<proof>

lemma *map_of_concat*:

" $k \in \text{dom} (\text{map_of}(\text{concat } l))$
 $\implies \exists i. k \in \text{dom} (\text{map_of}(l!i)) \wedge i < \text{length } l$ "
<proof>

lemma *map_of_concat'*:

" $k \in \text{dom} (\text{map_of}(l!i)) \wedge i < \text{length } l \implies k \in \text{dom} (\text{map_of}(\text{concat } l))$ "
<proof>

lemma *map_of_concat''*:

assumes " $\exists i. k \in \text{dom} (\text{map_of}(l!i)) \wedge i < \text{length } l$ "
shows " $k \in \text{dom} (\text{map_of}(\text{concat } l))$ "
<proof>

lemma *map_of_concat'''*:

" $(k \in \text{dom} (\text{map_of}(\text{concat } l)))$
 $\longleftrightarrow (\exists i. k \in \text{dom} (\text{map_of}(l!i)) \wedge i < \text{length } l)$ "
<proof>

lemma *abs_update_length*: " $\text{length} (\text{abs_update } k \ v \ l) = \text{length } l$ "

<proof>

lemma *ls_update_map_of_eq*:

" $\text{map_of} (\text{fst} (\text{ls_update } k \ v \ ls)) \ k = \text{Some } v$ "
<proof>

lemma *ls_update_map_of_neq*:

" $x \neq k \implies \text{map_of} (\text{fst} (\text{ls_update } k \ v \ ls)) \ x = \text{map_of } ls \ x$ "
<proof>

13.2 Main Definitions and Lemmas

definition *is_hashmap'*

":: "('k, 'v) map
 $\implies ('k \times 'v) \text{ list list}$
 $\implies ('k::\{\text{heap}, \text{hashable}\}, 'v::\text{heap}) \text{ hashtable}$
 $\implies \text{assn}$ "

where

"*is_hashmap'* $m \ l \ ht = \text{is_hashtable } l \ ht * \uparrow (\text{map_of} (\text{concat } l) = m)$ "

definition *is_hashmap*

":: "('k, 'v) map $\implies ('k::\{\text{heap}, \text{hashable}\}, 'v::\text{heap}) \text{ hashtable} \implies \text{assn}$ "

where

"*is_hashmap* $m \ ht = (\exists l. \text{is_hashmap}' \ m \ l \ ht)$ "

```

lemma is_hashmap'_prec:
  "∀ s s'. h |= (is_hashmap' m l ht * F1) ∧A (is_hashmap' m' l' ht * F2)
  → l=l' ∧ m=m'"
⟨proof⟩

```

```

lemma is_hashmap_prec: "precise is_hashmap"
⟨proof⟩

```

```

abbreviation "hm_new ≡ ht_new"

```

```

lemma hm_new_rule':
  "<emp>
  hm_new::('k::{heap,hashable}, 'v::heap) hashtable Heap
  <is_hashmap' Map.empty (replicate (def_hashmap_size TYPE('k)) [])>"
⟨proof⟩

```

```

lemma hm_new_rule:
  "<emp> hm_new <is_hashmap Map.empty>"
⟨proof⟩

```

```

lemma ht_hash_distinct:
  "ht_hash l
  ⇒ ∀ i j . i≠j ∧ i < length l ∧ j < length l
  → set (l!i) ∩ set (l!j) = {}"
⟨proof⟩

```

```

lemma ht_hash_in_dom_in_dom_bounded_hashcode_nat:
  assumes "ht_hash l"
  assumes "k ∈ dom (map_of(concat l))"
  shows "k ∈ dom (map_of(l!bounded_hashcode_nat (length l) k))"
⟨proof⟩

```

```

lemma ht_hash_in_dom_bounded_hashcode_nat_in_dom:
  assumes "ht_hash l"
  assumes "1 < length l"
  assumes "k ∈ dom (map_of(l!bounded_hashcode_nat (length l) k))"
  shows "k ∈ dom (map_of(concat l))"
⟨proof⟩

```

```

lemma ht_hash_in_dom_in_dom_bounded_hashcode_nat_eq:
  assumes "ht_hash l"
  assumes "1 < length l"
  shows "(k ∈ dom (map_of(concat l)))
  = (k ∈ dom (map_of(l!bounded_hashcode_nat (length l) k)))"
⟨proof⟩

```

```

lemma ht_hash_in_dom_i_bounded_hashcode_nat_i:
  assumes "ht_hash l"

```

```

    assumes "1 < length l"
    assumes "i < length l"
    assumes "k ∈ dom (map_of (l!i))"
    shows "i = bounded_hashcode_nat (length l) k"
    ⟨proof⟩

lemma ht_hash_in_bounded_hashcode_nat_not_i_not_in_dom_i:
    assumes "ht_hash l"
    assumes "1 < length l"
    assumes "i < length l"
    assumes "i ≠ bounded_hashcode_nat (length l) k"
    shows "k ∉ dom (map_of (l!i))"
    ⟨proof⟩

lemma ht_hash_ht_distinct_in_dom_unique_value:
    assumes "ht_hash l"
    assumes "ht_distinct l"
    assumes "1 < length l"
    assumes "k ∈ dom (map_of (concat l))"
    shows "∃!v. (k,v) ∈ set (concat l)"
    ⟨proof⟩

lemma ht_hash_ht_distinct_map_of:
    assumes "ht_hash l"
    assumes "ht_distinct l"
    assumes "1 < length l"
    shows "map_of (concat l) k
    = map_of (l!bounded_hashcode_nat (length l) k) k"
    ⟨proof⟩

lemma ls_lookup_map_of_pre:
    "distinct (map fst l) ⇒ ls_lookup k l = map_of l k"
    ⟨proof⟩

lemma ls_lookup_map_of:
    assumes "ht_hash l"
    assumes "ht_distinct l"
    assumes "1 < length l"
    shows "ls_lookup k (l ! bounded_hashcode_nat (length l) k)
    = map_of (concat l) k"
    ⟨proof⟩

abbreviation "hm_lookup ≡ ht_lookup"
lemma hm_lookup_rule':
    "<is_hashmap' m l ht> hm_lookup k ht
    <λr. is_hashmap' m l ht *
    ↑(r = m k)>"
    ⟨proof⟩

```

```

lemma hm_lookup_rule:
  "<is_hashmap m ht> hm_lookup k ht
   < $\lambda r. is\_hashmap\ m\ ht\ *$ 
      $\uparrow(r = m\ k)$ >"
  <proof>

lemma abs_update_map_of'':
  assumes "ht_hash l"
  assumes "ht_distinct l"
  assumes "1 < length l"
  shows "map_of (concat (abs_update k v l)) k = Some v"
  <proof>

lemma abs_update_map_of_hceq:
  assumes "ht_hash l"
  assumes "ht_distinct l"
  assumes "1 < length l"
  assumes "x  $\neq$  k"
  assumes "bounded_hashcode_nat (length l) x
           = bounded_hashcode_nat (length l) k"
  shows "map_of (concat (abs_update k v l)) x = map_of (concat l) x"
  <proof>

lemma abs_update_map_of_hcneq:
  assumes "ht_hash l"
  assumes "ht_distinct l"
  assumes "1 < length l"
  assumes "x  $\neq$  k"
  assumes "bounded_hashcode_nat (length l) x
            $\neq$  bounded_hashcode_nat (length l) k"
  shows "map_of (concat (abs_update k v l)) x = map_of (concat l) x"
  <proof>

lemma abs_update_map_of''':
  assumes "ht_hash l"
  assumes "ht_distinct l"
  assumes "1 < length l"
  assumes "x  $\neq$  k"
  shows "map_of (concat (abs_update k v l)) x = map_of (concat l) x"
  <proof>

lemma abs_update_map_of':
  assumes "ht_hash l"
  assumes "ht_distinct l"
  assumes "1 < length l"
  shows "map_of (concat (abs_update k v l)) x
         = ((map_of (concat l))(k  $\mapsto$  v)) x"

```

<proof>

```
lemma abs_update_map_of:  
  assumes "ht_hash l"  
  assumes "ht_distinct l"  
  assumes "1 < length l"  
  shows "map_of (concat (abs_update k v l))  
        = (map_of (concat l))(k ↦ v) "  
  <proof>
```

```
lemma ls_insls_map_of:  
  assumes "ht_hash ld"  
  assumes "ht_distinct ld"  
  assumes "1 < length ld"  
  assumes "distinct (map fst xs)"  
  shows "map_of (concat (ls_insls xs ld)) = map_of (concat ld) ++ map_of  
xs"  
  <proof>
```

```
lemma ls_insls_map_of':  
  assumes "ht_hash ls"  
  assumes "ht_distinct ls"  
  assumes "ht_hash ld"  
  assumes "ht_distinct ld"  
  assumes "1 < length ld"  
  assumes "n < length ls"  
  shows "map_of (concat (ls_insls (ls ! n) ld))  
        ++ map_of (concat (take n ls))  
        = map_of (concat ld) ++ map_of (concat (take (Suc n) ls))"  
  <proof>
```

```
lemma ls_copy_map_of:  
  assumes "ht_hash ls"  
  assumes "ht_distinct ls"  
  assumes "ht_hash ld"  
  assumes "ht_distinct ld"  
  assumes "1 < length ld"  
  assumes "n ≤ length ls"  
  shows "map_of (concat (ls_copy n ls ld)) = map_of (concat ld) ++ map_of  
(concat (take n ls))"  
  <proof>
```

```
lemma ls_rehash_map_of:  
  assumes "ht_hash l"  
  assumes "ht_distinct l"  
  assumes "1 < length l"  
  shows "map_of (concat (ls_rehash l)) = map_of (concat l)"
```

<proof>

```
lemma abs_update_rehash_map_of:
  assumes "ht_hash l"
  assumes "ht_distinct l"
  assumes "1 < length l"
  shows "map_of (concat (abs_update k v (ls_rehash l)))
        = (map_of (concat l))(k ↦ v)"
<proof>
```

```
abbreviation "hm_update ≡ ht_update"
lemma hm_update_rule':
  "<is_hashmap' m l ht>
   hm_update k v ht
  <λr. is_hashmap (m(k ↦ v)) r * true>"
<proof>
```

```
lemma hm_update_rule:
  "<is_hashmap m ht>
   hm_update k v ht
  <λr. is_hashmap (m(k ↦ v)) r * true>"
<proof>
```

```
lemma ls_delete_map_of:
  assumes "distinct (map fst l)"
  shows "map_of (fst (ls_delete k l)) x = ((map_of l) |' (- {k})) x"
<proof>
```

```
lemma update_ls_delete_map_of:
  assumes "ht_hash l"
  assumes "ht_distinct l"
  assumes "ht_hash (l[bounded_hashcode_nat (length l) k
    := fst (ls_delete k (l ! bounded_hashcode_nat (length l) k))])"
  assumes "ht_distinct (l[bounded_hashcode_nat (length l) k
    := fst (ls_delete k (l ! bounded_hashcode_nat (length l) k))])"
  assumes "1 < length l"
  shows "map_of (concat (l[bounded_hashcode_nat (length l) k
    := fst (ls_delete k (l ! bounded_hashcode_nat (length l) k))])) x
        = ((map_of (concat l)) |' (- {k})) x"
<proof>
```

```
abbreviation "hm_delete ≡ ht_delete"
lemma hm_delete_rule':
  "<is_hashmap' m l ht> hm_delete k ht <is_hashmap (m |' (-{k}))>"
<proof>
```

```
lemma hm_delete_rule:
  "<is_hashmap m ht> hm_delete k ht <is_hashmap (m |' (-{k}))>"
```

```

⟨proof⟩

definition hm_isEmpty :: "('k, 'v) hashtable ⇒ bool Heap" where
  "hm_isEmpty ht ≡ return (the_size ht = 0)"

lemma hm_isEmpty_rule':
  "<is_hashmap' m l ht>
  hm_isEmpty ht
  <λr. is_hashmap' m l ht * ↑(r ⟷ m=Map.empty)>"
  ⟨proof⟩

lemma hm_isEmpty_rule:
  "<is_hashmap m ht> hm_isEmpty ht <λr. is_hashmap m ht * ↑(r ⟷ m=Map.empty)>"
  ⟨proof⟩

definition hm_size :: "('k, 'v) hashtable ⇒ nat Heap" where
  "hm_size ht ≡ return (the_size ht)"

lemma length_card_dom_map_of:
  assumes "distinct (map fst l)"
  shows "length l = card (dom (map_of l))"
  ⟨proof⟩

lemma ht_hash_dom_map_of_disj:
  assumes "ht_hash l"
  assumes "i < length l"
  assumes "j < length l"
  assumes "i ≠ j"
  shows "dom (map_of (l!i)) ∩ dom (map_of (l!j)) = {}"
  ⟨proof⟩

lemma ht_hash_dom_map_of_disj_drop:
  assumes "ht_hash l"
  assumes "i < length l"
  shows "dom (map_of (l!i)) ∩ dom (map_of (concat (drop (Suc i) l)))
  = {}"
  ⟨proof⟩

lemma sum_list_length_card_dom_map_of_concat:
  assumes "ht_hash l"
  assumes "ht_distinct l"
  shows "sum_list (map length l) = card (dom (map_of (concat l)))"
  ⟨proof⟩

lemma hm_size_rule':
  "<is_hashmap' m l ht>
  hm_size ht"

```

```
<\r. is_hashmap' m l ht * ↑(r = card (dom m))>"
⟨proof⟩
```

```
lemma hm_size_rule:
  "<is_hashmap m ht>
   hm_size ht
  <\r. is_hashmap m ht * ↑(r = card (dom m))>"
  ⟨proof⟩
```

13.3 Iterators

13.3.1 Definitions

```
type_synonym ('k,'v) hm_it = "(nat × ('k×'v) list × ('k,'v) hashtable)"
```

```
fun hm_it_adjust
  :: "nat ⇒ ('k::{heap,hashable},'v::heap) hashtable ⇒ nat Heap"
  where
    "hm_it_adjust 0 ht = return 0"
  | "hm_it_adjust n ht = do {
    l ← Array.nth (the_array ht) n;
    case l of
      [] ⇒ hm_it_adjust (n - 1) ht
    | _ ⇒ return n
  }"
```

```
definition hm_it_init
  :: "('k::{heap,hashable},'v::heap) hashtable ⇒ ('k,'v) hm_it Heap"
  where
    "hm_it_init ht ≡ do {
    n←Array.len (the_array ht);
    if n = 0 then return (0, [],ht)
    else do {
      i←hm_it_adjust (n - 1) ht;
      l←Array.nth (the_array ht) i;
      return (i,l,ht)
    }
  }"
```

```
definition hm_it_has_next
  :: "('k::{heap,hashable},'v::heap) hm_it ⇒ bool Heap"
  where "hm_it_has_next it
  ≡ return (case it of (0, [],_) ⇒ False | _ ⇒ True)"
```

```
definition hm_it_next ::
  "('k::{heap,hashable},'v::heap) hm_it
  ⇒ (('k×'v)×('k,'v) hm_it) Heap"
  where "hm_it_next it ≡ case it of
    (i,a#b#l,ht) ⇒ return (a,(i,b#l,ht))
  | (0,[a],ht) ⇒ return (a,(0,[],ht))"
```



```

| (Suc i, [a], ht) ⇒ do {
  i ← hm_it_adjust i ht;
  l ← Array.nth (the_array ht) i;
  return (a, (i, rev l, ht))
}
"

```

```

definition "hm_is_it' l ht l' it ≡
  is_hashtable l ht *
  ↑(let (i,r,ht')=it in
    ht = ht'
    ∧ l' = (concat (take i l) @ rev r)
    ∧ distinct (map fst (l')))
  ∧ i ≤ length l ∧ (r=[] → i=0)
)"

```

```

definition "hm_is_it m ht m' it ≡ ∃_A l l'.
  hm_is_it' l ht l' it
  * ↑(map_of (concat l) = m ∧ map_of l' = m')
"

```

13.3.2 Auxiliary Lemmas

```

lemma concat_take_Suc_empty: "[[ n < length l; l!n=[] ]]"
  ⇒ concat (take (Suc n) l) = concat (take n l)"
  ⟨proof⟩

```

```

lemma nth_concat_splitE:
  assumes "i < length (concat ls)"
  obtains j k where
    "j < length ls"
    and "k < length (ls!j)"
    and "concat ls ! i = ls!j!k"
    and "i = length (concat (take j ls)) + k"
  ⟨proof⟩

```

```

lemma is_hashmap'_distinct:
  "is_hashtable l ht
  ⇒A is_hashtable l ht * ↑(distinct (map fst (concat l)))"
  ⟨proof⟩

```

```

lemma take_set: "set (take n l) = { l!i | i. i < n ∧ i < length l }"
  ⟨proof⟩

```

```

lemma skip_empty_aux:
  assumes A: "concat (take (Suc n) l) = concat (take (Suc x) l)"
  assumes L[simp]: "Suc n ≤ length l" "x ≤ n"
  shows "∀ i. x < i ∧ i ≤ n → l!i=[]"
  ⟨proof⟩

```

```

lemma take_Suc0:
  "l ≠ [] ⇒ take (Suc 0) l = [l!0]"
  "0 < length l ⇒ take (Suc 0) l = [l!0]"
  "Suc n ≤ length l ⇒ take (Suc 0) l = [l!0]"
  ⟨proof⟩

lemma concat_take_Suc_app_nth:
  assumes "x < length l"
  shows "concat (take (Suc x) l) = concat (take x l) @ l ! x"
  ⟨proof⟩

lemma hm_hashcode_eq:
  assumes "j < length (l!i)"
  assumes "i < length l"
  assumes "h ⊨ is_hashtable l ht"
  shows "bounded_hashcode_nat (length l) (fst (l!i!j)) = i"
  ⟨proof⟩

lemma distinct_imp_distinct_take:
  "distinct (map fst (concat l))
  ⇒ distinct (map fst (concat (take x l)))"
  ⟨proof⟩

lemma hm_it_adjust_rule:
  "i < length l ⇒ <is_hashtable l ht>
  hm_it_adjust i ht
  <λj. is_hashtable l ht * ↑(
    j ≤ i ∧
    (concat (take (Suc i) l) = concat (take (Suc j) l)) ∧
    (j=0 ∨ l!j ≠ [])
  )
  >"
  ⟨proof⟩

lemma hm_it_next_rule': "l' ≠ [] ⇒
  <hm_is_it' l ht l' it>
  hm_it_next it
  <λ((k,v),it').
  hm_is_it' l ht (butlast l') it'
  * ↑(last l' = (k,v) ∧ distinct (map fst l')) >"
  ⟨proof⟩

```

13.3.3 Main Lemmas

```

lemma hm_it_next_rule: "m' ≠ Map.empty ⇒
  <hm_is_it m ht m' it>
  hm_it_next it

```

$\langle \lambda((k,v),it'). hm_is_it\ m\ ht\ (m' \mid' (-\{k\}))\ it' * \uparrow(m' \ k = \text{Some } v) \rangle$ "
 $\langle proof \rangle$

lemma *hm_it_init_rule*:
 fixes *ht* :: "('k::{heap,hashable}, 'v::heap) hashtable"
 shows " $\langle is_hashmap\ m\ ht \rangle hm_it_init\ ht\ \langle hm_is_it\ m\ ht\ m \rangle_t$ "
 $\langle proof \rangle$

lemma *hm_it_has_next_rule*:
 " $\langle hm_is_it\ m\ ht\ m' \ it \rangle hm_it_has_next\ it$
 $\langle \lambda r. hm_is_it\ m\ ht\ m' \ it * \uparrow(r \longleftrightarrow m' \neq \text{Map.empty}) \rangle$ "
 $\langle proof \rangle$

lemma *hm_it_finish*: " $hm_is_it\ m\ p\ m' \ it \implies_A is_hashmap\ m\ p$ "
 $\langle proof \rangle$

end

14 Hash-Maps (Interface Instantiations)

theory *Hash_Map_Impl*
imports *Imp_Map_Spec Hash_Map*
begin

lemma *hm_map_impl*: "*imp_map is_hashmap*"
 $\langle proof \rangle$

interpretation *hm*: *imp_map is_hashmap* $\langle proof \rangle$

lemma *hm_empty_impl*: "*imp_map_empty is_hashmap hm_new*"
 $\langle proof \rangle$

interpretation *hm*: *imp_map_empty is_hashmap hm_new* $\langle proof \rangle$

lemma *hm_lookup_impl*: "*imp_map_lookup is_hashmap hm_lookup*"
 $\langle proof \rangle$

interpretation *hm*: *imp_map_lookup is_hashmap hm_lookup* $\langle proof \rangle$

lemma *hm_update_impl*: "*imp_map_update is_hashmap hm_update*"
 $\langle proof \rangle$

interpretation *hm*: *imp_map_update is_hashmap hm_update* $\langle proof \rangle$

lemma *hm_delete_impl*: "*imp_map_delete is_hashmap hm_delete*"
 $\langle proof \rangle$

interpretation *hm*: *imp_map_delete is_hashmap hm_delete* $\langle proof \rangle$

lemma *hm_is_empty_impl*: "*imp_map_is_empty is_hashmap hm_isEmpty*"
 $\langle proof \rangle$

interpretation *hm*: *imp_map_is_empty is_hashmap hm_isEmpty*
 $\langle proof \rangle$

```

lemma hm_size_impl: "imp_map_size is_hashmap hm_size"
  <proof>
interpretation hm: imp_map_size is_hashmap hm_size <proof>

lemma hm_iterate_impl:
  "imp_map_iterate is_hashmap hm_is_it hm_it_init hm_it_has_next hm_it_next"
  <proof>
interpretation hm:
  imp_map_iterate is_hashmap hm_is_it hm_it_init hm_it_has_next hm_it_next
  <proof>

export_code hm_new hm_lookup hm_update hm_delete hm_isEmpty hm_size
  hm_it_init hm_it_has_next hm_it_next
  checking SML_imp

end

```

15 Interface for Sets

```

theory Imp_Set_Spec
imports "../Sep_Main"
begin

```

This file specifies an abstract interface for set data structures. It can be implemented by concrete set data structures, as demonstrated in the hash set example.

```

locale imp_set =
  fixes is_set :: "'a set ⇒ 's ⇒ assn"
  assumes precise: "precise is_set"

locale imp_set_empty = imp_set +
  constrains is_set :: "'a set ⇒ 's ⇒ assn"
  fixes empty :: "'s Heap"
  assumes empty_rule[sep_heap_rules]: "<emp> empty <is_set {}>_t"

locale imp_set_is_empty = imp_set +
  constrains is_set :: "'a set ⇒ 's ⇒ assn"
  fixes is_empty :: "'s ⇒ bool Heap"
  assumes is_empty_rule[sep_heap_rules]:
    "<is_set s p> is_empty p <\r. is_set s p * ↑(r ↔ s={})>_t"

locale imp_set_memb = imp_set +
  constrains is_set :: "'a set ⇒ 's ⇒ assn"
  fixes memb :: "'a ⇒ 's ⇒ bool Heap"
  assumes memb_rule[sep_heap_rules]:
    "<is_set s p> memb a p <\r. is_set s p * ↑(r ↔ a ∈ s)>_t"

```

```

locale imp_set_ins = imp_set +
  constrains is_set :: "'a set ⇒ 's ⇒ assn"
  fixes ins :: "'a ⇒ 's ⇒ 's Heap"
  assumes ins_rule[sep_heap_rules]:
    "<is_set s p> ins a p <is_set (Set.insert a s)>_t"

locale imp_set_delete = imp_set +
  constrains is_set :: "'a set ⇒ 's ⇒ assn"
  fixes delete :: "'a ⇒ 's ⇒ 's Heap"
  assumes delete_rule[sep_heap_rules]:
    "<is_set s p> delete a p <is_set (s - {a})>_t"

locale imp_set_size = imp_set +
  constrains is_set :: "'a set ⇒ 's ⇒ assn"
  fixes size :: "'s ⇒ nat Heap"
  assumes size_rule[sep_heap_rules]:
    "<is_set s p> size p <λr. is_set s p * ↑(r = card s)>_t"

locale imp_set_iterate = imp_set +
  constrains is_set :: "'a set ⇒ 's ⇒ assn"
  fixes is_it :: "'a set ⇒ 's ⇒ 'a set ⇒ 'it ⇒ assn"
  fixes it_init :: "'s ⇒ ('it) Heap"
  fixes it_has_next :: "'it ⇒ bool Heap"
  fixes it_next :: "'it ⇒ ('a × 'it) Heap"
  assumes it_init_rule[sep_heap_rules]:
    "<is_set s p> it_init p <is_it s p s>_t"
  assumes it_next_rule[sep_heap_rules]: "s' ≠ {} ⇒
    <is_it s p s' it>
      it_next it
    <λ(a,it'). is_it s p (s' - {a}) it' * ↑(a ∈ s')>_t"
  assumes it_has_next_rule[sep_heap_rules]:
    "<is_it s p s' it> it_has_next it <λr. is_it s p s' it * ↑(r ↔ s' ≠ {})>_t"
  assumes quit_iteration:
    "is_it s p s' it ⇒A is_set s p * true"

locale imp_set_union = imp_set_iterate +
  fixes union :: "'s ⇒ 's ⇒ 's Heap"
  assumes union_rule[sep_heap_rules]:
    "finite se ⇒ <(is_set s p) * (is_set se q)> union p q <λr.
    ∃As'. is_set s' r * (is_set se q)* true * ↑(s' = s ∪ se)>_t"

partial_function (heap) set_it_union
  where [code]: "set_it_union
    it_has_next it_next set_ins it a = do {
      co ← it_has_next it;

```

```

    if co then do {
      (x,it') ← it_next it;
      insx ← set_ins x a;
      set_it_union it_has_next it_next set_ins it' (insx)
    } else return a
  }"

```

```

lemma set_it_union_rule:
  assumes "imp_set_iterate is_set is_it it_init it_has_next it_next"
  assumes "imp_set_ins is_set set_ins"
  assumes FIN: "finite it"
  shows "
    <is_it b q it iti * is_set a p>
      set_it_union it_has_next it_next set_ins iti p
    <λr. ∃As'. is_set s' r * is_set b q * true * ↑(s' = a ∪ it)>"
  <proof>

```

```

definition union_loop_ins where
  "union_loop_ins it_init it_has_next it_next set_ins a b ≡ do {
    it ← (it_init b);
    set_it_union it_has_next it_next set_ins it a
  }"

```

```

lemma set_union_rule:
  assumes IT: "imp_set_iterate is_set is_it it_init it_has_next it_next"
  assumes INS: "imp_set_ins is_set set_ins"
  assumes finb: "finite b"
  shows "
    <is_set a p * is_set b q>
    union_loop_ins it_init it_has_next it_next set_ins p q
    <λr. ∃As'. is_set s' r * true * is_set b q * ↑(s' = a ∪ b)>"
  <proof>

```

end

16 Hash-Sets

```

theory Hash_Set_Impl
imports Imp_Set_Spec Hash_Map_Impl
begin

```

16.1 Auxiliary Definitions

```
definition map_of_set :: "'a set  $\Rightarrow$  'a  $\rightarrow$  unit"  
  where "map_of_set S x  $\equiv$  if x $\in$ S then Some () else None"
```

```
lemma ne_some_unit_eq: "x $\neq$ Some ()  $\longleftrightarrow$  x=None"  
  <proof>
```

```
lemma map_of_set_simps[simp]:  
  "dom (map_of_set s) = s"  
  "map_of_set (dom m) = m"  
  "map_of_set {} = Map.empty"  
  "map_of_set s x = None  $\longleftrightarrow$  x $\notin$ s"  
  "map_of_set s x = Some u  $\longleftrightarrow$  x $\in$ s"  
  "(map_of_set s) (x $\mapsto$ ()) = map_of_set (insert x s)"  
  "(map_of_set s) |' (-{x}) = map_of_set (s -{x})"  
  <proof>
```

```
lemma map_of_set_eq':  
  "map_of_set a = map_of_set b  $\longleftrightarrow$  a=b"  
  <proof>
```

```
lemma map_of_set_eq[simp]:  
  "map_of_set s = m  $\longleftrightarrow$  dom m=s"  
  <proof>
```

16.2 Main Definitions

```
type_synonym 'a hashset = "('a,unit) hashtable"  
definition "is_hashset s ht  $\equiv$  is_hashmap (map_of_set s) ht"
```

```
lemma hs_set_impl: "imp_set is_hashset"  
  <proof>
```

```
interpretation hs: imp_set is_hashset <proof>
```

```
definition hs_new :: "'a::{heap,hashable} hashset Heap"  
  where "hs_new = hm_new"
```

```
lemma hs_new_impl: "imp_set_empty is_hashset hs_new"  
  <proof>
```

```
interpretation hs: imp_set_empty is_hashset hs_new <proof>
```

```
definition hs_memb :: "'a::{heap,hashable}  $\Rightarrow$  'a hashset  $\Rightarrow$  bool Heap"  
  where "hs_memb x s  $\equiv$  do {  
    r $\leftarrow$ hm_lookup x s;  
    return (case r of Some _  $\Rightarrow$  True | None  $\Rightarrow$  False)  
  }"
```

```
lemma hs_memb_impl: "imp_set_memb is_hashset hs_memb"  
  <proof>
```

```

interpretation hs: imp_set_memb is_hashset hs_memb <proof>

definition hs_ins:: "'a::{heap,hashable} => 'a hashset => 'a hashset Heap"
  where "hs_ins x ht ≡ hm_update x () ht"

lemma hs_ins_impl: "imp_set_ins is_hashset hs_ins"
  <proof>
interpretation hs: imp_set_ins is_hashset hs_ins <proof>

definition hs_delete
  :: "'a::{heap,hashable} => 'a hashset => 'a hashset Heap"
  where "hs_delete x ht ≡ hm_delete x ht"

lemma hs_delete_impl: "imp_set_delete is_hashset hs_delete"
  <proof>
interpretation hs: imp_set_delete is_hashset hs_delete
  <proof>

definition "hs_isEmpty == hm_isEmpty"

lemma hs_is_empty_impl: "imp_set_is_empty is_hashset hs_isEmpty"
  <proof>
interpretation hs: imp_set_is_empty is_hashset hs_isEmpty
  <proof>

definition "hs_size == hm_size"

lemma hs_size_impl: "imp_set_size is_hashset hs_size"
  <proof>
interpretation hs: imp_set_size is_hashset hs_size <proof>

type_synonym ('a) hs_it = "('a,unit) hm_it"

definition "hs_is_it s hs its it
  ≡ hm_is_it (map_of_set s) hs (map_of_set its) it"

definition hs_it_init :: "('a::{heap,hashable}) hashset => 'a hs_it Heap"
  where "hs_it_init ≡ hm_it_init"

definition hs_it_has_next :: "('a::{heap,hashable}) hs_it => bool Heap"
  where "hs_it_has_next ≡ hm_it_has_next"

definition hs_it_next
  :: "('a::{heap,hashable}) hs_it => ('a×'a hs_it) Heap"
  where
    "hs_it_next it ≡ do {
      ((x,_),it) ← hm_it_next it;
      return (x,it)
    }"

```



```

lemma hs_iterate_impl: "imp_set_iterate
  is_hashset hs_is_it hs_it_init hs_it_has_next hs_it_next"
  <proof>

interpretation hs: imp_set_iterate
  is_hashset hs_is_it hs_it_init hs_it_has_next hs_it_next
  <proof>

definition "hs_union
  ≡ union_loop_ins hs_it_init hs_it_has_next hs_it_next hs_ins"

lemmas hs_union_rule[sep_heap_rules] =
  set_union_rule[OF hs_iterate_impl hs_ins_impl,
    folded hs_union_def]

lemma hs_union_impl: "imp_set_union
  is_hashset hs_is_it hs_it_init hs_it_has_next hs_it_next hs_union"
  <proof>

interpretation hs: imp_set_union
  is_hashset hs_is_it hs_it_init hs_it_has_next hs_it_next hs_union
  <proof>

export_code hs_new hs_memb hs_ins hs_delete hs_isEmpty hs_size
  hs_it_init hs_it_has_next hs_it_next hs_union
  checking SML_imp

end

```

17 Generic Algorithm to Convert Sets to Lists

```

theory To_List_GA
imports Imp_Set_Spec Imp_List_Spec Hash_Set_Impl Open_List
begin

```

This theory demonstrates how to develop a generic to-list algorithm, and gives a sample instantiation for hash sets and open lists.

17.1 Algorithm

```

partial_function (heap) to_list_ga_rec where [code]:
  "to_list_ga_rec
    it_has_next it_next
    l_prepend
    it l

```

```

=
do {
  b ← it_has_next it;
  if b then do {
    (x,it) ← it_next it;
    l ← l_prepend x l;
    to_list_ga_rec it_has_next it_next
      l_prepend it l
  } else
    return l
}
}
"

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"
  shows "
  < is_it s si it iti * is_list l li >
    to_list_ga_rec it_has_next it_next l_prepend iti li
  < λr. ∃_A1'. is_set s si
    * is_list l' r
    * ↑(set l' = set l ∪ it) >_t"
  <proof>

definition "to_list_ga
  it_init it_has_next it_next
  l_empty l_prepend s
  ≡ do {
    it ← it_init s;
    l ← l_empty;
    l ← to_list_ga_rec it_has_next it_next l_prepend it l;
    return l
  }"

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
  <λr. ∃_A1. is_set s si * is_list l r * true * ↑(set l = s)>"
  <proof>

```

17.2 Sample Instantiation for hash set and open list

```
definition "hs_to_ol
  ≡ to_list_ga hs_it_init hs_it_has_next hs_it_next
    os_empty os_prepend"

lemmas hs_to_ol_rule[sep_heap_rules] =
  to_list_ga_rule[OF hs_iterate_impl os_empty_impl os_prepend_impl,
    folded hs_to_ol_def]

export_code hs_to_ol checking SML_imp

end
```

18 Union-Find Data-Structure

```
theory Union_Find
imports
  "../Sep_Main"
  Collections.Partial_Equivalence_Relation
  "HOL-Library.Code_Target_Numerals"
begin
```

We implement a simple union-find data-structure based on an array. It uses path compression and a size-based union heuristics.

18.1 Abstract Union-Find on Lists

We first formulate union-find structures on lists, and later implement them using Imperative/HOL. This is a separation of proof concerns between proving the algorithmic idea correct and generating the verification conditions.

18.1.1 Representatives

We define a function that searches for the representative of an element. This function is only partially defined, as it does not terminate on all lists. We use the domain of this function to characterize valid union-find lists.

```
function (domintros) rep_of
  where "rep_of l i = (if l!i = i then i else rep_of l (l!i))"
  <proof>
```

A valid union-find structure only contains valid indexes, and the *rep_of* function terminates for all indexes.

```
definition
  "ufa_invar l ≡ ∀i<length l. rep_of_dom (l,i) ∧ l!i<length l"
```

```

lemma ufa_invarD:
  "[[ ufa_invar l; i < length l ]] ==> rep_of_dom (l, i)"
  "[[ ufa_invar l; i < length l ]] ==> l!i < length l"
  <proof>

```

We derive the following equations for the *rep-of* function.

```

lemma rep_of_refl: "l!i=i ==> rep_of l i = i"
  <proof>

```

```

lemma rep_of_step:
  "[[ ufa_invar l; i < length l; l!i ≠ i ]] ==> rep_of l i = rep_of l (l!i)"
  <proof>

```

```

lemmas rep_of_simps = rep_of_refl rep_of_step

```

```

lemma rep_of_iff: "[[ ufa_invar l; i < length l ]]
  ==> rep_of l i = (if l!i=i then i else rep_of l (l!i))"
  <proof>

```

We derive a custom induction rule, that is more suited to our purposes.

```

lemma rep_of_induct[case_names base step, consumes 2]:
  assumes I: "ufa_invar l"
  assumes L: "i < length l"
  assumes BASE: "∧i. [[ ufa_invar l; i < length l; l!i=i ]] ==> P l i"
  assumes STEP: "∧i. [[ ufa_invar l; i < length l; l!i ≠ i; P l (l!i) ]]
    ==> P l i"
  shows "P l i"
  <proof>

```

In the following, we define various properties of *rep_of*.

```

lemma rep_of_min:
  "[[ ufa_invar l; i < length l ]] ==> l!(rep_of l i) = rep_of l i"
  <proof>

```

```

lemma rep_of_bound:
  "[[ ufa_invar l; i < length l ]] ==> rep_of l i < length l"
  <proof>

```

```

lemma rep_of_idem:
  "[[ ufa_invar l; i < length l ]] ==> rep_of l (rep_of l i) = rep_of l i"
  <proof>

```

```

lemma rep_of_min_upd: "[[ ufa_invar l; x < length l; i < length l ]] ==>
  rep_of (l[rep_of l x := rep_of l x]) i = rep_of l i"
  <proof>

```

```

lemma rep_of_idx:
  "[[ ufa_invar l; i < length l ]] ==> rep_of l (l!i) = rep_of l i"
  <proof>

```

18.1.2 Abstraction to Partial Equivalence Relation

definition `ufa_α` :: "nat list ⇒ (nat×nat) set"

where "ufa_α l

≡ {(x,y). x<length l ∧ y<length l ∧ rep_of l x = rep_of l y}"

lemma `ufa_α_equiv[simp, intro!]`: "part_equiv (ufa_α l)"

⟨proof⟩

lemma `ufa_α_lenD`:

"(x,y)∈ufa_α l ⇒ x<length l"

"(x,y)∈ufa_α l ⇒ y<length l"

⟨proof⟩

lemma `ufa_α_dom[simp]`: "Domain (ufa_α l) = {0..

⟨proof⟩

lemma `ufa_α_refl[simp]`: "(i,i)∈ufa_α l ↔ i<length l"

⟨proof⟩

lemma `ufa_α_len_eq`:

assumes "ufa_α l = ufa_α l'"

shows "length l = length l'"

⟨proof⟩

18.1.3 Operations

lemma `ufa_init_invar`: "ufa_invar [0..

⟨proof⟩

lemma `ufa_init_correct`: "ufa_α [0..

⟨proof⟩

lemma `ufa_find_correct`: "⟦ufa_invar l; x<length l; y<length l⟧

⇒ rep_of l x = rep_of l y ↔ (x,y)∈ufa_α l"

⟨proof⟩

abbreviation "ufa_union l x y ≡ l[rep_of l x := rep_of l y]"

lemma `ufa_union_invar`:

assumes I: "ufa_invar l"

assumes L: "x<length l" "y<length l"

shows "ufa_invar (ufa_union l x y)"

⟨proof⟩

lemma `ufa_union_aux`:

assumes I: "ufa_invar l"

assumes L: "x<length l" "y<length l"

assumes IL: "i<length l"

shows "rep_of (ufa_union l x y) i =

```

      (if rep_of l i = rep_of l x then rep_of l y else rep_of l i)"
    <proof>

lemma ufa_union_correct: "[ ufa_invar l; x<length l; y<length l ]
  => ufa_α (ufa_union l x y) = per_union (ufa_α l) x y"
  <proof>

lemma ufa_compress_aux:
  assumes I: "ufa_invar l"
  assumes L[simp]: "x<length l"
  shows "ufa_invar (l[x := rep_of l x])"
  and "∀i<length l. rep_of (l[x := rep_of l x]) i = rep_of l i"
  <proof>

lemma ufa_compress_invar:
  assumes I: "ufa_invar l"
  assumes L[simp]: "x<length l"
  shows "ufa_invar (l[x := rep_of l x])"
  <proof>

lemma ufa_compress_correct:
  assumes I: "ufa_invar l"
  assumes L[simp]: "x<length l"
  shows "ufa_α (l[x := rep_of l x]) = ufa_α l"
  <proof>

```

18.2 Implementation with Imperative/HOL

In this section, we implement the union-find data-structure with two arrays, one holding the next-pointers, and another one holding the size information. Note that we do not prove that the array for the size information contains any reasonable values, as the correctness of the algorithm is not affected by this. We leave it future work to also estimate the complexity of the algorithm.

```

type_synonym uf = "nat array × nat array"

definition is_uf :: "(nat×nat) set ⇒ uf ⇒ assn" where
  "is_uf R u ≡ case u of (s,p) ⇒
  ∃_A l szl. p↦_a l * s↦_a szl
  * ↑(ufa_invar l ∧ ufa_α l = R ∧ length szl = length l)"

definition uf_init :: "nat ⇒ uf Heap" where
  "uf_init n ≡ do {
  l ← Array.of_list [0..<n];
  szl ← Array.new n (1::nat);
  return (szl,l)
}"

```

```

lemma uf_init_rule[sep_heap_rules]:
  "<emp> uf_init n <is_uf {(i,i) | i. i<n}>"
  <proof>

```

```

partial_function (heap) uf_rep_of :: "nat array  $\Rightarrow$  nat  $\Rightarrow$  nat Heap"
  where [code]:
    "uf_rep_of p i = do {
      n  $\leftarrow$  Array.nth p i;
      if n=i then return i else uf_rep_of p n
    }"

```

```

lemma uf_rep_of_rule[sep_heap_rules]: "[[ ufa_invar l; i<length l ]]  $\Longrightarrow$ 
  <p $\mapsto_a$ l> uf_rep_of p i < $\lambda$ r. p $\mapsto_a$ l *  $\uparrow$ (r=rep_of l i)>"
  <proof>

```

We chose a non tail-recursive version here, as it is easier to prove.

```

partial_function (heap) uf_compress :: "nat  $\Rightarrow$  nat  $\Rightarrow$  nat array  $\Rightarrow$  unit
Heap"
  where [code]:
    "uf_compress i ci p = (
      if i=ci then return ()
      else do {
        ni $\leftarrow$ Array.nth p i;
        uf_compress ni ci p;
        Array.upd i ci p;
        return ()
      })"

```

```

lemma uf_compress_rule: "[[ ufa_invar l; i<length l; ci=rep_of l i ]]  $\Longrightarrow$ 
  <p $\mapsto_a$ l> uf_compress i ci p
  < $\lambda$ _.  $\exists_A l'$ . p $\mapsto_a$ l' *  $\uparrow$ (ufa_invar l'  $\wedge$  length l' = length l
     $\wedge$  ( $\forall i$ <length l. rep_of l' i = rep_of l i))>"
  <proof>

```

```

definition uf_rep_of_c :: "nat array  $\Rightarrow$  nat  $\Rightarrow$  nat Heap"
  where "uf_rep_of_c p i  $\equiv$  do {
    ci $\leftarrow$ uf_rep_of p i;
    uf_compress i ci p;
    return ci
  }"

```

```

lemma uf_rep_of_c_rule[sep_heap_rules]: "[[ ufa_invar l; i<length l ]]  $\Longrightarrow$ 
  <p $\mapsto_a$ l> uf_rep_of_c p i < $\lambda$ r.  $\exists_A l'$ . p $\mapsto_a$ l'
    *  $\uparrow$ (r=rep_of l i  $\wedge$  ufa_invar l'
       $\wedge$  length l' = length l
       $\wedge$  ( $\forall i$ <length l. rep_of l' i = rep_of l i))>"
  <proof>

```

```

definition uf_cmp :: "uf  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  bool Heap" where

```

```

"uf_cmp u i j ≡ do {
  let (s,p)=u;
  n←Array.len p;
  if (i≥n ∨ j≥n) then return False
  else do {
    ci←uf_rep_of_c p i;
    cj←uf_rep_of_c p j;
    return (ci=cj)
  }
}"

```

lemma *cnv_to_ufa_α_eq*:
 $\llbracket (\forall i < \text{length } l. \text{rep_of } l' \ i = \text{rep_of } l \ i); \text{length } l = \text{length } l' \rrbracket$
 $\implies (\text{ufa}_\alpha \ l = \text{ufa}_\alpha \ l')$
<proof>

lemma *uf_cmp_rule[sep_heap_rules]*:
 $\langle \text{is_uf } R \ u \rangle \text{uf_cmp } u \ i \ j \langle \lambda r. \text{is_uf } R \ u \ * \ \uparrow(r \longleftrightarrow (i, j) \in R) \rangle$
<proof>

definition *uf_union* :: "uf ⇒ nat ⇒ nat ⇒ uf Heap" **where**

```

"uf_union u i j ≡ do {
  let (s,p)=u;
  ci ← uf_rep_of p i;
  cj ← uf_rep_of p j;
  if (ci=cj) then return (s,p)
  else do {
    si ← Array.nth s ci;
    sj ← Array.nth s cj;
    if si<sj then do {
      Array.upd ci cj p;
      Array.upd cj (si+sj) s;
      return (s,p)
    } else do {
      Array.upd cj ci p;
      Array.upd ci (si+sj) s;
      return (s,p)
    }
  }
}"

```

lemma *uf_union_rule[sep_heap_rules]*: " $\llbracket i \in \text{Domain } R; j \in \text{Domain } R \rrbracket$
 $\implies \langle \text{is_uf } R \ u \rangle \text{uf_union } u \ i \ j \langle \text{is_uf } (\text{per_union } R \ i \ j) \rangle$
<proof>

export_code *uf_init uf_cmp uf_union checking SML_imp*


```
export_code uf_init uf_cmp uf_union checking Scala_imp
```

```
end
```

19 Common Proof Methods and Idioms

```
theory Idioms
imports "../Sep_Main" Open_List Circ_List Hash_Set_Impl
begin
```

This theory gives a short documentation of common proof techniques and idioms for the separation logic framework. For this purpose, it presents some proof snippets (inspired by the other example theories), and heavily comments on them.

19.1 The Method *sep_auto*

The most versatile method of our framework is *sep_auto*, which integrates the verification condition generator, the entailment solver and some pre- and postprocessing tactics based on the simplifier and classical reasoner. It can be applied to a Hoare-triple or entailment subgoal, and will try to solve it, and any emerging new goals. It stops when the goal is either solved or it gets stuck somewhere.

As a simple example for *sep_auto* consider the following program that does some operations on two circular lists:

```
definition "test ≡ do {
  l1 ← cs_empty;
  l2 ← cs_empty;
  l1 ← cs_append ''a'' l1;
  l2 ← cs_append ''c'' l2;
  l1 ← cs_append ''b'' l1;
  l2 ← cs_append ''e'' l2;
  l2 ← cs_prepend ''d'' l2;
  l2 ← cs_rotate l2;
  return (l1,l2)
}"
```

The *sep_auto* method does all the necessary frame-inference automatically, and thus manages to prove the following lemma in one step:

```
lemma "<emp>
  test
  <λ(l1,l2). cs_list [''a'', ''b''] l1
    * cs_list [''c'', ''e'', ''d''] l2>_t"
  <proof>
```

`sep_auto` accepts all the section-options of the classical reasoner and simplifier, e.g., `simp add/del:`, `intro:`. Moreover, it has some more section options, the most useful being `heap add/del:` to add or remove Hoare-rules that are applied with frame-inference. A complete documentation of the accepted options can be found in Section 5.9.

As a typical example, consider the following proof:

```
lemma complete_ht_rehash:
  "<is_hashtable l ht> ht_rehash ht
   < $\lambda r. is\_hashtable\ l\ ht * is\_hashtable\ (ls\_rehash\ l)\ r$ >"
  <proof>
```

19.2 Applying Single Rules

Hoare Triples In this example, we show how to do a proof step-by-step.

```
lemma
  "<os_list xs n> os_prepend x n <os_list (x # xs)>"
  <proof>
```

Note that the proof above can be done with `sep_auto`, the "Swiss army knife" of our framework

```
lemma
  "<os_list xs n> os_prepend x n <os_list (x # xs)>"
  <proof>
```

Entailment This example presents an actual proof from the circular list theory, where we have to manually apply a rule and give some hints to frame inference

```
lemma cs_append_rule:
  "<cs_list l p> cs_append x p <cs_list (l@[x])>"
  <proof>
```

19.3 Functions with Explicit Recursion

If the termination argument of a function depends on one of its parameters, we can use the function package. For example, the following function inserts elements from a list into a hash-set:

```
fun ins_from_list
  :: "('x::{heap,hashable}) list  $\Rightarrow$  'x hashset  $\Rightarrow$  'x hashset Heap"
  where
    "ins_from_list [] hs = return hs" |
    "ins_from_list (x # l) hs = do { hs  $\leftarrow$  hs_ins x hs; ins_from_list
  l hs }"
```

Proofs over such functions are usually done by structural induction on the explicit parameter, in this case, on the list

```

lemma ins_from_list_correct:
  "<is_hashset s hs> ins_from_list l hs <is_hashset (s∪set l)>t"
  <proof>

```

19.4 Functions with Recursion Involving the Heap

If the termination argument of a function depends on data stored on the heap, *partial_function* is a useful tool.

Note that, despite the name, proving a Hoare-Triple $\langle \dots \rangle \dots \langle \dots \rangle$ for something defined with *partial_function* implies total correctness.

In the following example, we compute the sum of a list, using an iterator. Note that the partial-function package does not provide a code generator setup by default, so we have to add a *[code]* attribute manually

```

partial_function (heap) os_sum' :: "int os_list_it ⇒ int ⇒ int Heap"

  where [code]:
    "os_sum' it s = do {
      b ← os_it_has_next it;
      if b then do {
        (x,it') ← os_it_next it;
        os_sum' it' (s+x)
      } else return s
    }"

```

The proof that the function is correct can be done by induction over the representation of the list that we still have to iterate over. Note that for iterators over sets, we need induction on finite sets, cf. also *To_List_Ga.thy*

```

lemma os_sum'_rule:
  "<os_is_it l p l' it>
  os_sum' it s
  <λr. os_list l p * ↑(r = s + sum_list l')>t"
  <proof>

```

19.5 Precision Proofs

Precision lemmas show that an assertion uniquely determines some of its parameters. Our example shows that two list segments from the same start pointer and with the same list, also have to end at the same end pointer.

```

lemma lseg_prec3:
  "∀q q'. h ⊨ (lseg l p q * F1) ∧A (lseg l p q' * F2) → q=q'"
  <proof>
end

```

20 Conclusion

We have presented a separation logic framework for Imperative HOL. It provides powerful proof methods for reasoning over imperative monadic programs, thus rectifying the lack of good proof support in the original Imperative HOL formalization.

We verified the applicability of our framework by proving algorithms on various data structures. Moreover, we showed how to construct an imperative collection framework, that supports generic algorithms and data refinement.

Acknowledgments We thank Thomas Tuerk, the author of Holfoot [8], for useful discussions on the automation of separation logic. Moreover, we thank Lukas Bulwahn and Brian Huffman for help with the Isabelle ML interface.

References

- [1] J. Bengtson, J. B. Jensen, F. Sieczkowski, and L. Birkedal. Verifying object-oriented programs with higher-order separation logic in coq. In M. C. J. D. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *ITP*, volume 6898 of *Lecture Notes in Computer Science*, pages 22–38. Springer, 2011.
- [2] J. Berdine, C. Calcagno, and P. W. O’Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.
- [3] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews. Imperative functional programming with isabelle/hol. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 134–149. Springer, 2008.
- [4] G. Klein, R. Kolanski, and A. Boyton. Separation algebra. *Archive of Formal Proofs*, 2012, 2012.
- [5] P. Lammich and A. Lochbihler. The isabelle collections framework. In M. Kaufmann and L. C. Paulson, editors, *ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2010.
- [6] R. Meis. Integration von Separation Logic in das Imperative HOL-Framework. Diplomarbeit, University of Münster, April 2011.
- [7] P. W. O’Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *CSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.
- [8] T. Tuerk. A separation logic framework for HOL. Technical Report UCAM-CL-TR-799, University of Cambridge, Computer Laboratory, June 2011.