A Separation Logic Framework for Imperative HOL

Peter Lammich and Rene Meis

March 19, 2025

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

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 ⇒ state ⇒ state ⇒ 'a ⇒ bool" where
  push_exn: "is_exn σ ⟹ run c σ σ r " |
  new_exn:  "⟦¬ is_exn σ; execute c (the_state σ) = None⟧
    ⟹ run c σ None r" |
  regular:  "⟦¬ is_exn σ; execute c (the_state σ) = Some (r, h')⟧
    ⟹ run c σ (Some h') r"
```

2.0.1 Link with *effect* and *success*

```
lemma run_effectE:
  assumes "run c σ σ' r"
  assumes "¬is_exn σ'"
  obtains h h' where
    "σ=Some h" "σ' = Some h'"
    "effect c h h' r"
  using assms
  unfolding effect_def
  apply (cases σ)
  by (auto simp add: run.simps)


lemma run_effectI:
  assumes  "run c (Some h) (Some h') r"
  shows  "effect c h h' r"
  using run_effectE[OF assms] by auto
```

```
lemma effect_run:
  assumes "effect c h h' r"
  shows "run c (Some h) (Some h') r"
  using assms
  unfolding effect_def
  by (auto intro: run.intros)

lemma success_run:
  assumes "success f h"
  obtains h' r where "run f (Some h) (Some h') r"
proof -
  from assms obtain r h'
    where "Heap_Monad.execute f h = Some (r, h')"
    unfolding success_def by auto
  then show thesis by (rule that[OF regular[of "Some h", simplified]])
qed
```

run always yields a result

```
lemma run_complete:
  obtains σ' r where "run c σ σ' r"
  apply (cases "is_exn σ")
  apply (auto intro: run.intros)
  apply (cases "execute c (the_state σ)")
  by (auto intro: run.intros)

lemma run_detE:
  assumes "run c σ σ' r" "run c σ τ s"
          "¬is_exn σ"
  obtains "is_exn σ'" "σ' = τ" | "¬ is_exn σ'" "σ' = τ" "r = s"
  using assms
  by (auto simp add: run.simps)

lemma run_detI:
  assumes "run c (Some h) (Some h') r" "run c (Some h) σ s"
  shows "σ = Some h' ∧ r = s"
  using assms
  by (auto simp add: run.simps)

lemma run_exn:
  assumes "run f σ σ' r"
          "is_exn σ"
  obtains "σ'=σ"
  using assms
  apply (cases σ)
  apply (auto elim!: run.cases intro: that)
  done
```

2.0.2 Elimination Rules for Basic Combinators

named_theorems *run_elims "elemination rules for run"*

lemma *runE[run_elims]:*
  assumes *"run (f ≫= g) σ σ'' r"*
  obtains *σ' r'* **where**
    "run f σ σ' r'"
    "run (g r') σ' σ'' r"
using *assms*
apply *(cases "is_exn σ")*
apply *(simp add: run.simps)*
apply *(cases "execute f (the_state σ)")*
apply *(simp add: run.simps bind_def)*
by *(auto simp add: bind_def run.simps)*

lemma *runE'[run_elims]:*
  assumes *"run (f ≫ g) σ σ'' res"*
  obtains *σt rt* **where**
    "run f σ σt rt"
    "run g σt σ'' res"
  using *assms*
  by *(rule_tac runE)*

lemma *run_return[run_elims]:*
  assumes *"run (return x) σ σ' r"*
  obtains *"r = x" "σ' = σ" "¬ is_exn σ" | "σ = None"*
  using *assms* **apply** *(cases σ)* **apply** *(simp add: run.simps)*
  by *(auto simp add: run.simps execute_simps)*

lemma *run_raise_iff: "run (raise s) σ σ' r ⟷ (σ'=None)"*
  apply *(cases σ)*
  by *(auto simp add: run.simps execute_simps)*

lemma *run_raise[run_elims]:*
  assumes *"run (raise s) σ σ' r"*
  obtains *"σ' = None"*
  using *assms* **by** *(simp add: run_raise_iff)*

lemma *run_raiseI:*
  "run (raise s) σ None r" **by** *(simp add: run_raise_iff)*

lemma *run_if[run_elims]:*
  assumes *"run (if c then t else e) h h' r"*
  obtains *"c" "run t h h' r"*
        | "¬c" "run e h h' r"
  using *assms*
  by *(auto split: if_split_asm)*

```
lemma run_case_option[run_elims]:
  assumes "run (case x of None ⇒ n | Some y ⇒ s y) σ σ' r"
          "¬is_exn σ"
  obtains "x = None" "run n σ σ' r"
        | y where "x = Some y" "run (s y) σ σ' r"
  using assms
  by (cases x) simp_all

lemma run_heap[run_elims]:
  assumes "run (Heap_Monad.heap f) σ σ' res"
          "¬is_exn σ"
  obtains "σ' = Some (snd (f (the_state σ)))"
  and "res = (fst (f (the_state σ)))"
  using assms
  apply (cases σ)
  apply simp
  apply (auto simp add: run.simps)
  apply (simp add: execute_simps)

  apply (simp only: execute_simps)
  apply hypsubst_thin
  subgoal premises prems for a h'
  proof -
    from prems(2) have "h' = snd (f a)" "res = fst (f a)" by simp_all
    from prems(1)[OF this] show ?thesis .
  qed
  done
```

2.1 Array Commands

```
lemma run_length[run_elims]:
  assumes "run (Array.len a) σ σ' r"
          "¬is_exn σ"
  obtains "¬is_exn σ" "σ' = σ" "r = Array.length (the_state σ) a"
  using assms
  apply (cases σ)
  by (auto simp add: run.simps execute_simps)


lemma run_new_array[run_elims]:
  assumes "run (Array.new n x) σ σ' r"
          "¬is_exn σ"
  obtains "σ' = Some (snd (Array.alloc (replicate n x) (the_state σ)))"
  and "r = fst (Array.alloc (replicate n x) (the_state σ))"
  and "Array.get (the_state σ') r = replicate n x"
  using assms
  apply (cases σ)
  apply simp
```

```
  apply (auto simp add: run.simps)
  apply (simp add: execute_simps)
  apply (simp add: Array.get_alloc)
  apply hypsubst_thin
  subgoal premises prems for a h'
  proof -
    from prems(2) have "h' = snd (Array.alloc (replicate n x) a)"
      "r = fst (Array.alloc (replicate n x) a)" by (auto simp add: execute_simps)
    then show ?thesis by (rule prems(1))
  qed
  done

lemma run_make[run_elims]:
  assumes "run (Array.make n f) σ σ' r"
          "¬is_exn σ"
  obtains "σ' = Some (snd (Array.alloc (map f [0 ..< n]) (the_state σ)))"
          "r = fst (Array.alloc (map f [0 ..< n]) (the_state σ))"
          "Array.get (the_state σ') r = (map f [0 ..< n])"
  using assms
  apply (cases σ)
  subgoal by simp
  subgoal by (simp add: run.simps execute_simps Array.get_alloc; fastforce)
  done

lemma run_upd[run_elims]:
  assumes "run (Array.upd i x a) σ σ' res"
          "¬is_exn σ"
  obtains "¬ i < Array.length (the_state σ) a"
          "σ' = None"
  |
          "i < Array.length (the_state σ) a"
          "σ' = Some (Array.update a i x (the_state σ))"
          "res = a"
  using assms
  apply (cases σ)
  apply simp
  apply (cases "i < Array.length (the_state σ) a")
  apply (auto simp add: run.simps)
  apply (simp_all only: execute_simps)
  prefer 3
  apply auto[2]
  apply hypsubst_thin
  subgoal premises prems for aa h'
  proof -
    from prems(3) have "h' = Array.update a i x aa" "res = a" by auto
    then show ?thesis by (rule prems(1))
  qed
  done
```

```
lemma run_nth[run_elims]:
  assumes "run (Array.nth a i) σ σ' r"
          "¬is_exn σ"
  obtains "¬is_exn σ"
    "i < Array.length (the_state σ) a"
    "r = (Array.get (the_state σ) a) ! i"
    "σ' = σ"
  |
    "¬ i < Array.length (the_state σ) a"
    "σ' = None"
  using assms
  apply (cases σ)
  apply simp
  apply (cases "i < Array.length (the_state σ) a")
  apply (auto simp add: run.simps)
  apply (simp_all only: execute_simps)
  prefer 3
  apply auto[2]
  apply hypsubst_thin
  subgoal premises prems for aa h'
  proof -
    from prems(3) have "r = Array.get aa a ! i" "h' = aa" by auto
    then show ?thesis by (rule prems(1))
  qed
  done


lemma run_of_list[run_elims]:
  assumes "run (Array.of_list xs) σ σ' r"
          "¬is_exn σ"
  obtains "σ' = Some (snd (Array.alloc xs (the_state σ)))"
          "r = fst (Array.alloc xs (the_state σ))"
          "Array.get (the_state σ') r = xs"
  using assms
  apply (cases σ)
  apply simp
  apply (auto simp add: run.simps)
  apply (simp add: execute_simps)
  apply (simp add: Array.get_alloc)
  apply hypsubst_thin
  subgoal premises prems for a h'
  proof -
    from prems(2) have "h' = snd (Array.alloc xs a)"
      "r = fst (Array.alloc xs a)" by (auto simp add: execute_simps)
    then show ?thesis by (rule prems(1))
  qed
  done
```

```
lemma run_freeze[run_elims]:
  assumes "run (Array.freeze a) σ σ' r"
          "¬is_exn σ"
  obtains "σ' = σ"
          "r = Array.get (the_state σ) a"
  using assms
  apply (cases σ)
  by (auto simp add: run.simps execute_simps)
```

2.2 Reference Commands

```
lemma run_new_ref[run_elims]:
  assumes "run (ref x) σ σ' r"
          "¬is_exn σ"
  obtains "σ' = Some (snd (Ref.alloc x (the_state σ)))"
          "r = fst (Ref.alloc x (the_state σ))"
          "Ref.get (the_state σ') r = x"
  using assms
  apply (cases σ)
  apply simp
  apply (auto simp add: run.simps)
  apply (simp add: execute_simps)
  apply hypsubst_thin
  subgoal premises prems for a h'
  proof -
    from prems(2) have
      "h' = snd (Ref.alloc x a)"
      "r = fst (Ref.alloc x a)"
      by (auto simp add: execute_simps)
    then show ?thesis by (rule prems(1))
  qed
  done

lemma "fst (Ref.alloc x h) = Ref (lim h)"
  unfolding alloc_def
  by (simp add: Let_def)


lemma run_update[run_elims]:
  assumes "run (p := x) σ σ' r"
          "¬is_exn σ"
  obtains "σ' = Some (Ref.set p x (the_state σ))" "r = ()"
  using assms
  unfolding Ref.update_def
  by (auto elim: run_heap)

lemma run_lookup[run_elims]:
  assumes "run (!p) σ σ' r"
          "¬ is_exn σ"
```

obtains *"¬is_exn σ" "σ' = σ" "r = Ref.get (the_state σ) p"*
using *assms*
apply *(cases σ)*
by *(auto simp add: run.simps execute_simps)*

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 × 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,{})"*
by *(auto simp: in_range.simps)*

lemma *in_range_dist_union[simp]:*
"in_range (h,as ∪ as') ⟷ in_range (h,as) ∧ in_range (h,as')"
by *(auto simp: in_range.simps)*

lemma *in_range_subset:*
"⟦as ⊆ as'; in_range (h,as')⟧ ⟹ in_range (h,as)"
by *(auto simp: in_range.simps)*

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)"*
using *assms* **unfolding** *relH_def* **by** *auto*

Reflexivity

lemma *relH_refl: "in_range (h,as) ⟹ relH as h h"*
unfolding *relH_def* **by** *simp*

Symmetry

lemma *relH_sym: "relH as h h' ⟹ relH as h' h"*
unfolding *relH_def*
by *auto*

Transitivity

lemma *relH_trans[trans]: "⟦relH as h1 h2; relH as h2 h3⟧ ⟹ relH as h1 h3"*
unfolding *relH_def*
by *auto*

lemma *relH_dist_union[simp]:*
"relH (as∪as') h h' ⟷ relH as h h' ∧ relH as' h h'"
unfolding *relH_def*
by *auto*

lemma *relH_subset:*
assumes *"relH bs h h'"*
assumes *"as ⊆ bs"*
shows *"relH as h h'"*
using *assms* **unfolding** *relH_def* **by** *(auto intro: in_range_subset)*

lemma *relH_ref:*
assumes *"relH as h h'"*
assumes *"addr_of_ref r ∈ as"*
shows *"Ref.get h r = Ref.get h' r"*
using *assms* **unfolding** *relH_def Ref.get_def* **by** *auto*

lemma *relH_array:*
assumes *"relH as h h'"*
assumes *"addr_of_array r ∈ as"*
shows *"Array.get h r = Array.get h' r"*
using *assms* **unfolding** *relH_def Array.get_def* **by** *auto*

```
lemma relH_set_ref: "⟦ addr_of_ref r ∉ as; in_range (h,as)⟧
  ⟹ relH as h (Ref.set r x h)"
  unfolding relH_def Ref.set_def
  by (auto simp: in_range.simps)

lemma relH_set_array: "⟦addr_of_array r ∉ as; in_range (h,as)⟧
  ⟹ relH as h (Array.set r x h)"
  unfolding relH_def Array.set_def
  by (auto simp: in_range.simps)
```

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 ⇒ bool"

definition proper :: "assn_raw ⇒ bool" where
  "proper P ≡ ∀h h' as. (P (h,as) ⟶ in_range (h,as))
    ∧ (P (h,as) ∧ relH as h h' ∧ in_range (h',as) ⟶ P (h',as))"

lemma properI[intro?]:
  assumes "⋀as h. P (h,as) ⟹ in_range (h,as)"
  assumes "⋀as h h'.
    ⟦P (h,as); relH as h h'; in_range (h',as)⟧ ⟹ P (h',as)"
  shows "proper P"
  unfolding proper_def using assms by blast

lemma properD1:
  assumes "proper P"
  assumes "P (h,as)"
  shows "in_range (h,as)"
  using assms unfolding proper_def by blast

lemma properD2:
  assumes "proper P"
  assumes "P (h,as)"
  assumes "relH as h h'"
  assumes "in_range (h',as)"
  shows "P (h',as)"
  using assms unfolding proper_def by blast

lemmas properD = properD1 properD2

lemma proper_iff:
  assumes "proper P"
  assumes "relH as h h'"
  assumes "in_range (h',as)"
  shows "P (h,as) ⟷ P (h',as)"
```

```
  using assms
  by (metis properD2 relH_in_rangeI(1) relH_sym)
```

We encapsulate assertions in their own type

```
typedef assn = "Collect proper"
  apply simp
  unfolding proper_def
  by fastforce

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

lemma Abs_assn_eqI[intro?]:
  "(⋀h. P h = Rep_assn Pr h) ⟹ Abs_assn P = Pr"
  "(⋀h. P h = Rep_assn Pr h) ⟹ Pr = Abs_assn P"
  by (metis Rep_assn_inverse predicate1I xt1(5))+

abbreviation models :: "pheap ⇒ assn ⇒ bool" (infix ‹⊨› 50)
  where "h⊨P ≡ Rep_assn P h"


lemma models_in_range: "h⊨P ⟹ in_range h"
  apply (cases h)
  by (metis mem_Collect_eq Rep_assn properD1)
```

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 $h_\bot$

```
abbreviation h_bot (‹h⊥›) where "h⊥ ≡ (undefined,{})"
lemma mod_h_bot_indep: "(h,{})⊨P ⟷ (h',{})⊨P"
  by (metis mem_Collect_eq Rep_assn emptyE in_range_empty
    proper_iff relH_def)

lemma mod_h_bot_normalize[simp]:
  "syntax_fo_nomatch undefined h ⟹ (h,{})⊨P ⟷ h⊥ ⊨ P"
  using mod_h_bot_indep[where h'=undefined] by simp
```

Properness, lifted to the assertion type.

```
lemma mod_relH: "relH as h h' ⟹ (h,as)⊨P ⟷ (h',as)⊨P"
  by (metis mem_Collect_eq Rep_assn proper_iff relH_in_rangeI(2))
```

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 ⇒ bool"
    where "one_assn_raw (h,as) ⟷ as={}"

  lemma one_assn_proper[intro!,simp]: "proper one_assn_raw"
    by (auto intro!: properI)

  definition one_assn :: assn where "1 ≡ Abs_assn one_assn_raw"
  instance ..
end

abbreviation one_assn::assn (<emp>) where "one_assn ≡ 1"

instantiation assn :: times begin
  fun times_assn_raw :: "assn_raw ⇒ assn_raw ⇒ assn_raw" where
    "times_assn_raw P Q (h,as)
    = (∃as1 as2. as=as1∪as2 ∧ as1∩as2={}
        ∧ P (h,as1) ∧ Q (h,as2))"

  lemma times_assn_proper[intro!,simp]:
    "proper P ⟹ proper Q ⟹ proper (times_assn_raw P Q)"
    apply (rule properI)
    apply (auto dest: properD1) []
    apply clarsimp
    apply (drule (3) properD2)
    apply (drule (3) properD2)
    apply blast
    done

  definition times_assn where "P*Q ≡
    Abs_assn (times_assn_raw (Rep_assn P) (Rep_assn Q))"

  instance ..
end

lemma mod_star_conv: "h⊨A*B
  ⟷ (∃hr as1 as2. h=(hr,as1∪as2) ∧ as1∩as2={} ∧ (hr,as1)⊨A ∧ (hr,as2)⊨B)"
  unfolding times_assn_def
  apply (cases h)
  by (auto simp: Abs_assn_inverse)

lemma mod_starD: "h⊨A*B ⟹ ∃h1 h2. h1⊨A ∧ h2⊨B"
  by (auto simp: mod_star_conv)

lemma star_assnI:
```

```
  assumes "(h,as)⊨P" and "(h,as')⊨Q" and "as∩as'={}"
  shows "(h,as∪as')⊨P*Q"
  using assms unfolding times_assn_def
  by (auto simp: Abs_assn_inverse)

instantiation assn :: comm_monoid_mult begin
  lemma assn_one_left: "1*P = (P::assn)"
    unfolding one_assn_def times_assn_def
    apply (rule)
    apply (auto simp: Abs_assn_inverse)
    done

  lemma assn_times_comm: "P*Q = Q*(P::assn)"
    unfolding times_assn_def
    apply rule
    apply (fastforce simp add: Abs_assn_inverse Un_ac)
    done

  lemma assn_times_assoc: "(P*Q)*R = P*(Q*(R::assn))"
    unfolding times_assn_def
    apply rule
    apply (auto simp: Abs_assn_inverse)
    apply (rule_tac x="as1∪as1a" in exI)
    apply (rule_tac x="as2a" in exI)
    apply (auto simp add: Un_ac) []

    apply (rule_tac x="as1a" in exI)
    apply (rule_tac x="as2a∪as2" in exI)
    apply (fastforce simp add: Un_ac) []
    done

  instance
    apply standard
    apply (rule assn_times_assoc)
    apply (rule assn_times_comm)
    apply (rule assn_one_left)
    done

end
```

3.3.2 Magic Wand

```
fun wand_raw :: "assn_raw ⇒ assn_raw ⇒ assn_raw" where
  "wand_raw P Q (h,as) ⟷ in_range (h,as)
  ∧ (∀h' as'. as∩as'={} ∧ relH as h h' ∧ in_range (h',as)
    ∧ P (h',as')
    ⟶ Q (h',as∪as'))"

lemma wand_proper[simp, intro!]: "proper (wand_raw P Q)"
```

```
  apply (rule properI)
  apply simp
  apply (auto dest: relH_trans)
  done

definition
  wand_assn :: "assn ⇒ assn ⇒ assn" (infixl ‹-*› 56)
  where "P-*Q ≡ Abs_assn (wand_raw (Rep_assn P) (Rep_assn Q))"

lemma wand_assnI:
  assumes "in_range (h,as)"
  assumes "⋀h' as'. ⟦
    as ∩ as' = {};
    relH as h h';
    in_range (h',as);
    (h',as')⊨Q
  ⟧ ⟹ (h',as∪as') ⊨ R"
  shows "(h,as) ⊨ Q -* R"
  using assms
  unfolding wand_assn_def
  apply (auto simp: Abs_assn_inverse)
  done
```

3.3.3 Boolean Algebra on Assertions

```
instantiation assn :: boolean_algebra begin
  definition top_assn where "top ≡ Abs_assn in_range"
  definition bot_assn where "bot ≡ Abs_assn (λ_. False)"
  definition sup_assn where "sup P Q ≡ Abs_assn (λh. h⊨P ∨ h⊨Q)"
  definition inf_assn where "inf P Q ≡ Abs_assn (λh. h⊨P ∧ h⊨Q)"
  definition uminus_assn where
    "-P ≡ Abs_assn (λh. in_range h ∧ ¬h⊨P)"

  lemma bool_assn_proper[simp, intro!]:
    "proper in_range"
    "proper (λ_. False)"
    "proper P ⟹ proper Q ⟹ proper (λh. P h ∨ Q h)"
    "proper P ⟹ proper Q ⟹ proper (λh. P h ∧ Q h)"
    "proper P ⟹ proper (λh. in_range h ∧ ¬P h)"
    apply (auto
      intro!: properI
      intro: relH_in_rangeI
      dest: properD1
      simp: proper_iff)
    done
```

(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) ≤ b ≡ a = inf a b"

definition less_assn where
[simp]: "(a::assn) < b ≡ a ≤ b ∧ a≠b"

definition minus_assn where
[simp]: "(a::assn) - b ≡ inf a (-b)"

instance
  apply standard
  unfolding
    top_assn_def bot_assn_def sup_assn_def inf_assn_def uminus_assn_def
    less_eq_assn_def less_assn_def minus_assn_def
  apply (auto
    simp: Abs_assn_inverse conj_commute conj_ac
    intro: Abs_assn_eqI models_in_range)
  apply rule
  apply (metis (mono_tags) Abs_assn_inverse[unfolded mem_Collect_eq]
    Rep_assn[unfolded mem_Collect_eq] bool_assn_proper(4))
  apply rule
  apply (metis (mono_tags)
    Abs_assn_inverse[unfolded mem_Collect_eq]
    Rep_assn[unfolded mem_Collect_eq] bool_assn_proper(4))
  apply rule
  apply (simp add: Abs_assn_inverse)
  apply (metis (mono_tags)
    Abs_assn_inverse[unfolded mem_Collect_eq]
    Rep_assn[unfolded mem_Collect_eq] bool_assn_proper(4))
  done

end
```

We give the operations some more standard names

```
abbreviation top_assn::assn (<true>) where "top_assn ≡ top"
abbreviation bot_assn::assn (<false>) where "bot_assn ≡ bot"
abbreviation sup_assn::"assn⇒assn⇒assn" (infixr <∨A> 61)
  where "sup_assn ≡ sup"
abbreviation inf_assn::"assn⇒assn⇒assn" (infixr <∧A> 62)
  where "inf_assn ≡ inf"
abbreviation uminus_assn::"assn ⇒ assn" (<¬A _> [81] 80)
  where "uminus_assn ≡ 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"
  unfolding times_assn_def bot_assn_def
  apply rule
  apply (auto simp add: Abs_assn_inverse)
```

```
  done

lemma star_false_right[simp]: "P * false = false"
  using star_false_left by (simp add: assn_times_comm)

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"
  subgoal
    unfolding one_assn_def top_assn_def
    proof (subst Abs_assn_inject; simp?)
      have "in_range (⦇arrays = (λ_ _. []), refs = (λ_ _. 0), lim = 1⦈,{0})"
(is "in_range ?h")
        by (auto simp: in_range.simps)
      moreover have "¬one_assn_raw ?h" by auto
      ultimately show "in_range ≠ one_assn_raw" by auto
    qed
  subgoal
    by (simp add: ‹true ≠ emp›)
  subgoal
  using star_false_left ‹true ≠ emp› by force
  subgoal
    by (simp add: ‹false ≠ emp›)
  subgoal
    by (metis inf_bot_right inf_top.right_neutral ‹true ≠ emp›)
  subgoal
    using ‹true ≠ false› by auto
  done
```

3.3.4 Existential Quantification

```
definition ex_assn :: "('a ⇒ assn) ⇒ assn" (binder ‹∃_A› 11)
  where "(∃_A x. P x) ≡ Abs_assn (λh. ∃x. h⊨P x)"

lemma ex_assn_proper[simp, intro!]:
  "(⋀x. proper (P x)) ⟹ proper (λh. ∃x. P x h)"
  by (auto intro!: properI dest: properD1 simp: proper_iff)

lemma ex_assn_const[simp]: "(∃_A x. c) = c"
  unfolding ex_assn_def by auto

lemma ex_one_point_gen:
  "⟦⋀h x. h⊨P x ⟹ x=v⟧ ⟹ (∃_A x. P x) = (P v)"
  unfolding ex_assn_def
  apply rule
  apply auto
```

```
  done

lemma ex_distrib_star: "(∃A x. P x * Q) = (∃A x. P x) * Q"
  unfolding ex_assn_def times_assn_def
  apply rule
  apply (simp add: Abs_assn_inverse)
  apply fastforce
  done

lemma ex_distrib_and: "(∃A x. P x ∧A Q) = (∃A x. P x) ∧A Q"
  unfolding ex_assn_def inf_assn_def
  apply rule
  apply (simp add: Abs_assn_inverse)
  done

lemma ex_distrib_or: "(∃A x. P x ∨A Q) = (∃A x. P x) ∨A Q"
  unfolding ex_assn_def sup_assn_def
  apply rule
  apply (auto simp add: Abs_assn_inverse)
  done

lemma ex_join_or: "(∃A x. P x ∨A (∃A x. Q x)) = (∃A x. P x ∨A Q x)"
  unfolding ex_assn_def sup_assn_def
  apply rule
  apply (auto simp add: Abs_assn_inverse)
  done
```

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)"
  by (auto intro!: properI intro: relH_in_rangeI)



lemma pure_true[simp]: "↑True = emp"
  unfolding pure_assn_def one_assn_def
  apply rule
  apply (simp add: Abs_assn_inverse)
  apply (auto)
  done

lemma pure_false[simp]: "↑False = false"
  unfolding pure_assn_def bot_assn_def
  apply rule
```

```
  apply (auto simp: Abs_assn_inverse)
  done

lemma pure_assn_eq_false_iff[simp]: "↑P = false ⟷ ¬P" by auto

lemma pure_assn_eq_emp_iff[simp]: "↑P = emp ⟷ P" by (cases P) auto

lemma merge_pure_star[simp]:
  "↑a * ↑b = ↑(a∧b)"
  unfolding times_assn_def
  apply rule
  unfolding pure_assn_def
  apply (simp add: Abs_assn_inverse)
  apply fastforce
  done

lemma merge_true_star[simp]: "true*true = true"
  unfolding times_assn_def top_assn_def
  apply rule
  apply (simp add: Abs_assn_inverse)
  apply (fastforce simp: in_range.simps)
  done

lemma merge_pure_and[simp]:
  "↑a ∧_A ↑b = ↑(a∧b)"
  unfolding inf_assn_def
  apply rule
  unfolding pure_assn_def
  apply (simp add: Abs_assn_inverse)
  apply fastforce
  done

lemma merge_pure_or[simp]:
  "↑a ∨_A ↑b = ↑(a∨b)"
  unfolding sup_assn_def
  apply rule
  unfolding pure_assn_def
  apply (simp add: Abs_assn_inverse)
  apply fastforce
  done


lemma pure_assn_eq_conv[simp]: "↑P = ↑Q ⟷ P=Q" by auto

definition "is_pure_assn a ≡ ∃P. a=↑P"
lemma is_pure_assnE: assumes "is_pure_assn a" obtains P where "a=↑P"
  using assms
  by (auto simp: is_pure_assn_def)
```

```
lemma is_pure_assn_pure[simp, intro!]: "is_pure_assn (↑P)"
  by (auto simp add: is_pure_assn_def)

lemma is_pure_assn_basic_simps[simp]:
  "is_pure_assn false"
  "is_pure_assn emp"
proof -
  have "is_pure_assn (↑False)" by rule thus "is_pure_assn false" by
simp
  have "is_pure_assn (↑True)" by rule thus "is_pure_assn emp" by simp
qed

lemma is_pure_assn_starI[simp,intro!]:
  "⟦is_pure_assn a; is_pure_assn b⟧ ⟹ is_pure_assn (a*b)"
  by (auto elim!: is_pure_assnE)
```

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)"
  apply (auto intro!: properI simp: relH_ref)
  apply (simp add: in_range.simps)
  apply (auto simp add: in_range.simps dest: relH_in_rangeI)
  done

definition sngr_assn :: "'a::heap ref ⇒ 'a ⇒ assn" (infix ‹↦$_r$› 82)
  where "r↦$_r$x ≡ 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)"
  apply (auto intro!: properI simp: relH_array)
  apply (simp add: in_range.simps)
  apply (auto simp add: in_range.simps dest: relH_in_rangeI)
  done

definition
  snga_assn :: "'a::heap array ⇒ 'a list ⇒ assn" (infix ‹↦$_a$› 82)
  where "r↦$_a$a ≡ 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"
  unfolding times_assn_def bot_assn_def sngr_assn_def
  apply rule
  apply (auto simp: Abs_assn_inverse)
  done

lemma snga_same_false[simp]:
  "p ↦a x * p ↦a y = false"
  unfolding times_assn_def bot_assn_def snga_assn_def
  apply rule
  apply (auto simp: Abs_assn_inverse)
  done
```

3.4 Properties of the Models-Predicate

```
lemma mod_true[simp]: "h⊨true ⟷ in_range h"
  unfolding top_assn_def by (simp add: Abs_assn_inverse)
lemma mod_false[simp]: "¬ h⊨false"
  unfolding bot_assn_def by (simp add: Abs_assn_inverse)

lemma mod_emp: "h⊨emp ⟷ snd h = {}"
  unfolding one_assn_def by (cases h) (simp add: Abs_assn_inverse)

lemma mod_emp_simp[simp]: "(h,{})⊨emp"
  by (simp add: mod_emp)

lemma mod_pure[simp]: "h⊨↑b ⟷ snd h = {} ∧ b"
  unfolding pure_assn_def
  apply (cases h)
  apply (auto simp add: Abs_assn_inverse)
  done

lemma mod_ex_dist[simp]: "h⊨(∃A x. P x) ⟷ (∃x. h⊨P x)"
  unfolding ex_assn_def by (auto simp: Abs_assn_inverse)

lemma mod_exI: "∃x. h⊨P x ⟹ h⊨(∃A x. P x)"
  by (auto simp: mod_ex_dist)
lemma mod_exE: assumes "h⊨(∃A x. P x)" obtains x where "h⊨P x"
  using assms by (auto simp: mod_ex_dist)


lemma mod_and_dist: "h⊨P∧AQ ⟷ h⊨P ∧ h⊨Q"
  unfolding inf_assn_def by (simp add: Abs_assn_inverse)

lemma mod_or_dist[simp]: "h⊨P∨AQ ⟷ h⊨P ∨ h⊨Q"
  unfolding sup_assn_def by (simp add: Abs_assn_inverse)
```

lemma *mod_not_dist[simp]: "h⊨(¬$_A$P) ⟷ in_range h ∧ ¬ h⊨P"*
unfolding *uminus_assn_def* **by** *(simp add: Abs_assn_inverse)*

lemma *mod_pure_star_dist[simp]: "h⊨P*↑b ⟷ h⊨P ∧ b"*
by *(metis (full_types) mod_false mult_1_right pure_false*
pure_true star_false_right)

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⊨P ⟹ h⊨P*true"*
unfolding *times_assn_def top_assn_def*
apply *(simp add: Abs_assn_inverse)*
apply *(cases h)*
apply *auto*
done

lemma *mod_star_trueE':* **assumes** *"h⊨P*true"* **obtains** *h'* **where**
"fst h' = fst h" **and** *"snd h' ⊆ snd h"* **and** *"h'⊨P"*
using *assms*
unfolding *times_assn_def top_assn_def*
apply *(cases h)*
apply *(fastforce simp add: Abs_assn_inverse)*
done

lemma *mod_star_trueE:* **assumes** *"h⊨P*true"* **obtains** *h'* **where** *"h'⊨P"*
using *assms* **by** *(blast elim: mod_star_trueE')*

lemma *mod_h_bot_iff[simp]:*
"(h,{}) ⊨ ↑b ⟷ b"
"(h,{}) ⊨ true"
"(h,{}) ⊨ p↦$_r$x ⟷ False"
"(h,{}) ⊨ q↦$_a$y ⟷ False"
*"(h,{}) ⊨ P*Q ⟷ ((h,{}) ⊨ P) ∧ ((h,{}) ⊨ Q)"*
"(h,{}) ⊨ P∧$_A$Q ⟷ ((h,{}) ⊨ P) ∧ ((h,{}) ⊨ Q)"
"(h,{}) ⊨ P∨$_A$Q ⟷ ((h,{}) ⊨ P) ∨ ((h,{}) ⊨ Q)"
"(h,{}) ⊨ (∃$_A$x. R x) ⟷ (∃x. (h,{}) ⊨ R x)"
apply *(simp add: pure_assn_def Abs_assn_inverse)*
apply *simp*
apply *(simp add: sngr_assn_def Abs_assn_inverse)*
apply *(simp add: snga_assn_def Abs_assn_inverse)*
apply *(simp add: times_assn_def Abs_assn_inverse)*
apply *(simp add: inf_assn_def Abs_assn_inverse)*
apply *(simp add: sup_assn_def Abs_assn_inverse)*
apply *(simp add: ex_assn_def Abs_assn_inverse)*
done

3.5 Entailment

definition *entails :: "assn ⇒ assn ⇒ bool"* **(infix** *<⟹$_A$>* *10)*
  where *"P ⟹$_A$ Q ≡ ∀h. h⊨P ⟶ h⊨Q"*

lemma *entailsI:*
  assumes *"⋀h. h⊨P ⟹ h⊨Q"*
  shows *"P ⟹$_A$ Q"*
  using *assms* **unfolding** *entails_def* **by** *auto*

lemma *entailsD:*
  assumes *"P ⟹$_A$ Q"*
  assumes *"h⊨P"*
  shows *"h⊨Q"*
  using *assms* **unfolding** *entails_def* **by** *blast*

3.5.1 Properties

lemma *ent_fwd:*
  assumes *"h⊨P"*
  assumes *"P ⟹$_A$ Q"*
  shows *"h⊨Q"* **using** *assms(2,1)* **by** *(rule entailsD)*

lemma *ent_refl[simp]: "P ⟹$_A$ P"*
  by *(auto simp: entailsI)*

lemma *ent_trans[trans]: "⟦ P ⟹$_A$ Q; Q ⟹$_A$R ⟧ ⟹ P ⟹$_A$ R"*
  by *(auto intro: entailsI dest: entailsD)*

lemma *ent_iffI:*
  assumes *"A⟹$_A$B"*
  assumes *"B⟹$_A$A"*
  shows *"A=B"*
  apply *(subst Rep_assn_inject[symmetric])*
  apply *(rule ext)*
  using *assms* **unfolding** *entails_def*
  by *blast*

lemma *ent_false[simp]: "false ⟹$_A$ P"*
  by *(auto intro: entailsI)*
lemma *ent_true[simp]: "P ⟹$_A$ true"*
  by *(auto intro!: entailsI simp: models_in_range)*

lemma *ent_false_iff[simp]: "(P ⟹$_A$ false) ⟷ (∀h. ¬h⊨P)"*
  unfolding *entails_def*
  by *auto*

lemma *ent_pure_pre_iff[simp]: "(P*↑b ⟹$_A$ Q) ⟷ (b ⟶ (P ⟹$_A$ Q))"*
  unfolding *entails_def*
  by *(auto simp add: mod_dist)*

```
lemma ent_pure_pre_iff_sng[simp]:
  "(↑b ⟹A Q) ⟷ (b ⟶ (emp ⟹A Q))"
  using ent_pure_pre_iff[where P=emp]
  by simp

lemma ent_pure_post_iff[simp]:
  "(P ⟹A Q*↑b) ⟷ ((∀h. h⊨P ⟶ b) ∧ (P ⟹A Q))"
  unfolding entails_def
  by (auto simp add: mod_dist)

lemma ent_pure_post_iff_sng[simp]:
  "(P ⟹A ↑b) ⟷ ((∀h. h⊨P ⟶ b) ∧ (P ⟹A emp))"
  using ent_pure_post_iff[where Q=emp]
  by simp

lemma ent_ex_preI: "(⋀x. P x ⟹A Q) ⟹ ∃Ax. P x ⟹A Q"
  unfolding entails_def ex_assn_def
  by (auto simp: Abs_assn_inverse)

lemma ent_ex_postI: "(P ⟹A Q x) ⟹ P ⟹A ∃Ax. Q x"
  unfolding entails_def ex_assn_def
  by (auto simp: Abs_assn_inverse)

lemma ent_mp: "(P * (P -* Q)) ⟹A Q"
  apply (rule entailsI)
  unfolding times_assn_def wand_assn_def
  apply (clarsimp simp add: Abs_assn_inverse)
  apply (drule_tac x="a" in spec)
  apply (drule_tac x="as1" in spec)
  apply (auto simp: Un_ac relH_refl)
  done

lemma ent_star_mono: "⟦ P ⟹A P'; Q ⟹A Q'⟧ ⟹ P*Q ⟹A P'*Q'"
  unfolding entails_def times_assn_def
  apply (simp add: Abs_assn_inverse)
  apply metis
  done

lemma ent_wandI:
  assumes IMP: "Q*P ⟹A R"
  shows "P ⟹A (Q -* R)"
  unfolding entails_def
  apply clarsimp
  apply (rule wand_assnI)
  apply (blast intro: models_in_range)
proof -
  fix h as h' as'
  assume "(h,as)⊨P"
```

```
    and "as∩as'={}"
    and "relH as h h'"
    and "in_range (h',as)"
    and "(h',as') ⊨ Q"

  from ‹(h,as)⊨P› and ‹relH as h h'› have "(h',as)⊨P"
    by (simp add: mod_relH)
  with ‹(h',as') ⊨ Q› and ‹as∩as'={}› have "(h',as∪as')⊨Q*P"
    by (metis star_assnI Int_commute Un_commute)
  with IMP show "(h',as∪as') ⊨ R" by (blast dest: ent_fwd)
qed

lemma ent_disjI1:
  assumes "P ∨A Q ⟹A R"
  shows "P ⟹A R" using assms unfolding entails_def by simp

lemma ent_disjI2:
  assumes "P ∨A Q ⟹A R"
  shows "Q ⟹A R" using assms unfolding entails_def by simp

lemma ent_disjI1_direct[simp]: "A ⟹A A ∨A B"
  by (simp add: entails_def)

lemma ent_disjI2_direct[simp]: "B ⟹A A ∨A B"
  by (simp add: entails_def)

lemma ent_disjE: "⟦ A⟹AC; B⟹AC ⟧ ⟹ A∨AB ⟹AC"
  unfolding entails_def by auto

lemma ent_conjI: "⟦ A⟹AB; A⟹AC ⟧ ⟹ A ⟹A B ∧A C"
  unfolding entails_def by (auto simp: mod_and_dist)

lemma ent_conjE1: "⟦A⟹AC⟧ ⟹ A∧AB⟹AC"
  unfolding entails_def by (auto simp: mod_and_dist)
lemma ent_conjE2: "⟦B⟹AC⟧ ⟹ A∧AB⟹AC"
  unfolding entails_def by (auto simp: mod_and_dist)



lemma star_or_dist1:
  "(A ∨A B)*C = (A*C ∨A B*C)"
  apply (rule ent_iffI)
  unfolding entails_def
  by (auto simp add: mod_star_conv)

lemma star_or_dist2:
  "C*(A ∨A B) = (C*A ∨A C*B)"
  apply (rule ent_iffI)
  unfolding entails_def
```

```
  by (auto simp add: mod_star_conv)

lemmas star_or_dist = star_or_dist1 star_or_dist2

lemma ent_disjI1': "A ⟹_A B ⟹ A ⟹_A B ∨_A C"
  by (auto simp: entails_def star_or_dist)

lemma ent_disjI2': "A ⟹_A C ⟹ A ⟹_A B ∨_A C"
  by (auto simp: entails_def star_or_dist)

lemma triv_exI[simp, intro!]: "Q x ⟹_A ∃_A x. Q x"
  by (meson ent_ex_postI ent_refl)
```

3.5.2 Weak Entails

Weakening of entails to allow arbitrary unspecified memory in conclusion

```
definition entailst :: "assn ⇒ assn ⇒ bool" (infix ‹⟹_t› 10)
  where "entailst A B ≡ A ⟹_A B * true"

lemma enttI: "A⟹_A B*true ⟹ A⟹_t B" unfolding entailst_def .
lemma enttD: "A⟹_t B ⟹ A⟹_A B*true" unfolding entailst_def .

lemma entt_trans:
  "entailst A B ⟹ entailst B C ⟹ entailst A C"
  unfolding entailst_def
  apply (erule ent_trans)
  by (metis assn_times_assoc ent_star_mono ent_true merge_true_star)

lemma entt_refl[simp, intro!]: "entailst A A"
  unfolding entailst_def
  by (simp add: entailsI mod_star_trueI)

lemma entt_true[simp, intro!]:
  "entailst A true"
  unfolding entailst_def by simp

lemma entt_emp[simp, intro!]:
  "entailst A emp"
  unfolding entailst_def by simp

lemma entt_star_true_simp[simp]:
  "entailst A (B*true) ⟷ entailst A B"
  "entailst (A*true) B ⟷ entailst A B"
  unfolding entailst_def
  subgoal by (auto simp: assn_times_assoc)
  subgoal
    apply (intro iffI)
    subgoal using entails_def mod_star_trueI by blast
    subgoal by (metis assn_times_assoc ent_refl ent_star_mono merge_true_star)
```

```
    done
  done

lemma entt_star_mono: "⟦entailst A B; entailst C D⟧ ⟹ entailst (A*C)
(B*D)"
  unfolding entailst_def
proof -
  assume a1: "A ⟹_A B * true"
  assume "C ⟹_A D * true"
  then have "A * C ⟹_A true * B * (true * D)"
    using a1 assn_times_comm ent_star_mono by force
  then show "A * C ⟹_A B * D * true"
    by (simp add: ab_semigroup_mult_class.mult.left_commute assn_times_comm)
qed

lemma entt_frame_fwd:
  assumes "entailst P Q"
  assumes "entailst A (P*F)"
  assumes "entailst (Q*F) B"
  shows "entailst A B"
  using assms
  by (metis entt_refl entt_star_mono entt_trans)

lemma enttI_true: "P*true ⟹_A Q*true ⟹ P⟹_tQ"
  by (drule enttI) simp

lemma entt_def_true: "(P⟹_tQ) ≡ (P*true ⟹_A Q*true)"
  unfolding entailst_def
  apply (rule eq_reflection)
  using entailst_def entt_star_true_simp(2) by auto

lemma ent_imp_entt: "P⟹_AQ ⟹ P⟹_tQ"
  apply (rule enttI)
  apply (erule ent_trans)
  by (simp add: entailsI mod_star_trueI)

lemma entt_disjI1_direct[simp]: "A ⟹_t A ∨_A B"
  by (rule ent_imp_entt[OF ent_disjI1_direct])

lemma entt_disjI2_direct[simp]: "B ⟹_t A ∨_A B"
  by (rule ent_imp_entt[OF ent_disjI2_direct])

lemma entt_disjI1': "A⟹_tB ⟹ A⟹_tB∨_AC"
  by (auto simp: entailst_def entails_def star_or_dist)

lemma entt_disjI2': "A⟹_tC ⟹ A⟹_tB∨_AC"
  by (auto simp: entailst_def entails_def star_or_dist)
```

```
lemma entt_disjE: "⟦ A⟹ₜM; B⟹ₜM ⟧ ⟹ A∨ₐB ⟹ₜ M"
  using ent_disjE enttD enttI by blast

lemma entt_disjD1: "A∨ₐB⟹ₜC ⟹ A⟹ₜC"
  using entt_disjI1_direct entt_trans by blast

lemma entt_disjD2: "A∨ₐB⟹ₜC ⟹ B⟹ₜC"
  using entt_disjI2_direct entt_trans by blast
```

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:

```
∀x y. (h⊨(P x * F1) ∧ₐ (P y * F2)) ⟶ x=y

definition "precise R ≡ ∀a a' h p F F'.
  h ⊨ R a p * F ∧ₐ R a' p * F' ⟶ a = a'"

lemma preciseI[intro?]:
  assumes "⋀a a' h p F F'. h ⊨ R a p * F ∧ₐ R a' p * F' ⟹ a = a'"
  shows "precise R"
  using assms unfolding precise_def by blast

lemma preciseD:
  assumes "precise R"
  assumes "h ⊨ R a p * F ∧ₐ R a' p * F'"
  shows "a=a'"
  using assms unfolding precise_def by blast

lemma preciseD':
  assumes "precise R"
  assumes "h ⊨ R a p * F"
  assumes "h ⊨ R a' p * F'"
  shows "a=a'"
  apply (rule preciseD)
  apply (rule assms)
  apply (simp only: mod_and_dist)
  apply (blast intro: assms)
  done

lemma precise_extr_pure[simp]:
  "precise (λx y. ↑P * R x y) ⟷ (P ⟶ precise R)"
  "precise (λx y. R x y * ↑P) ⟷ (P ⟶ precise R)"
  apply (cases P, (auto intro!: preciseI) [2])+
  done
```

```
lemma sngr_prec: "precise (λx p. p↦ᵣx)"
  apply rule
  apply (clarsimp simp: mod_and_dist)
  unfolding sngr_assn_def times_assn_def
  apply (simp add: Abs_assn_inverse)
  apply auto
  done

lemma snga_prec: "precise (λx p. p↦ₐx)"
  apply rule
  apply (clarsimp simp: mod_and_dist)
  unfolding snga_assn_def times_assn_def
  apply (simp add: Abs_assn_inverse)
  apply auto
  done

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 ⇒ addr set ⇒ heap ⇒ addr set" where
  "new_addrs h as h' = as ∪ {a. lim h ≤ a ∧ a < lim h'}"

lemma new_addr_refl[simp]: "new_addrs h as h = as"
  unfolding new_addrs_def by auto
```

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 ⇒ 'a Heap ⇒ ('a ⇒ assn) ⇒ bool" (‹<_>/ _/ <_>›)
  where
  "<P> c <Q> ≡ ∀h as σ r. (h,as)⊨P ∧ run c (Some h) σ r
  ⟶ (let h'=the_state σ; as'=new_addrs h as h' in
    ¬is_exn σ ∧ (h',as')⊨Q r ∧ relH ({a . a<lim h ∧ a∉as}) h h'
    ∧ lim h ≤ lim h')"

Sanity checking theorems for Hoare-Triples

lemma
  assumes *"<P> c <Q>"*
  assumes *"(h,as)⊨P"*
  shows *hoare_triple_success: "success c h"*
    and *hoare_triple_effect: "∃h' r. effect c h h' r ∧ (h',new_addrs h as h')⊨Q r"*
  using *assms*
  unfolding *hoare_triple_def success_def effect_def*
  apply *-*
  apply *(auto simp: Let_def run.simps)* **apply** *fastforce*
  by *(metis is_exn.simps(2) not_Some_eq2 the_state.simps)*

lemma *hoare_tripleD:*
  fixes *h h' as as' σ r*
  assumes *"<P> c <Q>"*
  assumes *"(h,as)⊨P"*
  assumes *"run c (Some h) σ r"*
  defines *"h'≡the_state σ"* **and** *"as'≡new_addrs h as h'"*
  shows *"¬is_exn σ"*
  and *"(h',as')⊨Q r"*
  and *"relH ({a . a<lim h ∧ a∉as}) h h'"*
  and *"lim h ≤ lim h'"*
  using *assms*
  unfolding *hoare_triple_def h'_def as'_def*
  by *(auto simp: Let_def)*

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 ⇒ 'r Heap ⇒ ('r ⇒ assn) ⇒ bool" (‹<_> _ <_>$_t$›)
  where *"<P> c <Q>$_t$ ≡ <P> c <λr. 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 "⋀h. h⊨P ⟹ <P> c <Q>"
  shows "<P> c <Q>"
  using assms
  unfolding hoare_triple_def
  by auto


lemma frame_rule:
  assumes A: "<P> c <Q>"
  shows "<P*R> c <λx. Q x * R>"
  unfolding hoare_triple_def Let_def
  apply (intro allI impI)
  apply (elim conjE)
  apply (intro conjI)
proof -
  fix h as
  assume "(h,as) ⊨ P * R"
  then obtain as1 as2 where [simp]: "as=as1∪as2" and DJ: "as1∩as2={}"
    and M1: "(h,as1)⊨P" and M2: "(h,as2)⊨R"
    unfolding times_assn_def
    by (auto simp: Abs_assn_inverse)

  fix σ r
  assume RUN: "run c (Some h) σ r"
  from hoare_tripleD(1)[OF A M1 RUN] show "¬ is_exn σ" .
  from hoare_tripleD(4)[OF A M1 RUN]
  show "lim h ≤ lim (the_state σ)" .

  from hoare_tripleD(3)[OF A M1 RUN] have
    RH1: "relH {a. a < lim h ∧ a ∉ as1} h (the_state σ)" .
  moreover have "{a. a < lim h ∧ a ∉ as} ⊆ {a. a < lim h ∧ a ∉ as1}"
    by auto
  ultimately show "relH {a. a < lim h ∧ a ∉ as} h (the_state σ)"
    by (blast intro: relH_subset)

  from hoare_tripleD(2)[OF A M1 RUN] have
    "(the_state σ, new_addrs h as1 (the_state σ)) ⊨ Q r" .
  moreover have DJN: "new_addrs h as1 (the_state σ) ∩ as2 = {}"
    using DJ models_in_range[OF M2]
    by (auto simp: in_range.simps new_addrs_def)
  moreover have "as2 ⊆ {a. a < lim h ∧ a ∉ as1}"
    using DJ models_in_range[OF M2]
    by (auto simp: in_range.simps)
  hence "relH as2 h (the_state σ)" using RH1
    by (blast intro: relH_subset)
  with M2 have "(the_state σ, as2)⊨R"
    by (metis mem_Collect_eq Rep_assn
```

```
      proper_iff relH_in_rangeI(2))
  moreover have "new_addrs h as (the_state σ)
    = new_addrs h as1 (the_state σ) ∪ as2"
    by (auto simp: new_addrs_def)
  ultimately show
    "(the_state σ, new_addrs h as (the_state σ)) ⊨ Q r * R"
    unfolding times_assn_def
    apply (simp add: Abs_assn_inverse)
    apply blast
    done
qed


lemma false_rule[simp, intro!]: "<false> c <Q>"
  unfolding hoare_triple_def by simp


lemma cons_rule:
  assumes CPRE: "P ⟹_A P'"
  assumes CPOST: "⋀x. Q x ⟹_A Q' x"
  assumes R: "<P'> c <Q>"
  shows "<P> c <Q'>"
  unfolding hoare_triple_def Let_def
  using hoare_tripleD[OF R entailsD[OF CPRE]] entailsD[OF CPOST]
  by blast

lemmas cons_pre_rule = cons_rule[OF _ ent_refl]
lemmas cons_post_rule = cons_rule[OF ent_refl, rotated]

lemma cons_rulet: "⟦P⟹_tP'; ⋀x. Q x ⟹_t Q' x; <P'> c <Q>_t ⟧ ⟹ <P>
c <Q'>_t"
  unfolding entailst_def
  apply (rule cons_pre_rule)
  apply assumption
  apply (rule cons_post_rule)
  apply (erule frame_rule)
  by (simp add: enttD enttI)

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: "⋀x. <P x> f <Q>"
  shows "<∃_A x. P x> f <Q>"
  unfolding hoare_triple_def Let_def
  apply (intro allI impI, elim conjE mod_exE)
  using hoare_tripleD[OF A]
```

```
  by blast

lemma norm_pre_pure_iff[simp]:
  "<P*↑b> f <Q> ⟷ (b ⟶ <P> f <Q>)"
  unfolding hoare_triple_def Let_def
  by auto

lemma norm_pre_pure_iff_sng[simp]:
  "<↑b> f <Q> ⟷ (b ⟶ <emp> f <Q>)"
  using norm_pre_pure_iff[where P=emp]
  by simp

lemma norm_pre_pure_rule1:
  "⟦b ⟹ <P> f <Q>⟧ ⟹ <P*↑b> f <Q>" by simp

lemma norm_pre_pure_rule2:
  "⟦ b ⟹ <emp> f <Q> ⟧ ⟹ <↑b> f <Q>" by simp

lemmas norm_pre_pure_rule = norm_pre_pure_rule1 norm_pre_pure_rule2

lemma post_exI_rule: "<P> c <λr. Q r x> ⟹ <P> c <λr. ∃A x. Q r x>"
  by (blast intro: cons_post_rule ent_ex_postI ent_refl)
```

4.2.2 Rules for Atomic Commands

```
lemma ref_rule:
  "<emp> ref x <λr. r ↦r x>"
  unfolding one_assn_def sngr_assn_def hoare_triple_def
  apply (simp add: Let_def Abs_assn_inverse)
  apply (intro allI impI)
  apply (elim conjE run_elims)
  apply simp
  apply (auto
    simp: new_addrs_def Ref.alloc_def Let_def
    Ref.set_def Ref.get_def relH_def in_range.simps)
  done

lemma lookup_rule:
  "<p ↦r x> !p <λr. p ↦r x * ↑(r = x)>"
  unfolding hoare_triple_def sngr_assn_def
  apply (simp add: Let_def Abs_assn_inverse)
  apply (auto elim: run_elims simp add: relH_refl in_range.simps new_addrs_def)
  done

lemma update_rule:
  "<p ↦r y> p := x <λr. p ↦r x>"
  unfolding hoare_triple_def sngr_assn_def
  apply (auto elim!: run_update
    simp: Let_def Abs_assn_inverse new_addrs_def in_range.simps
```

```
    intro!: relH_set_ref)
  done

lemma update_wp_rule:
  "<r ↦r y * ((r ↦r x) -* (Q ()))> r := x <Q>"
  apply (rule cons_post_rule)
  apply (rule frame_rule[OF update_rule[where p=r and x=x],
    where R="((r ↦r x) -* (Q ()))"])
  apply (rule ent_trans)
  apply (rule ent_mp)
  by simp

lemma new_rule:
  "<emp> Array.new n x <λr. r ↦a replicate n x>"
  unfolding hoare_triple_def snga_assn_def one_assn_def
  apply (simp add: Let_def Abs_assn_inverse)
  apply (auto
    elim!: run_elims
    simp: Let_def new_addrs_def Array.get_def Array.set_def Array.alloc_def
      relH_def in_range.simps
  )
  done

lemma make_rule: "<emp> Array.make n f <λr. r ↦a (map f [0 ..< n])>"
  unfolding hoare_triple_def snga_assn_def one_assn_def
  apply (simp add: Let_def Abs_assn_inverse)
  apply (auto
    elim!: run_elims
    simp: Let_def new_addrs_def Array.get_def Array.set_def Array.alloc_def
      relH_def in_range.simps
  )
  done

lemma of_list_rule: "<emp> Array.of_list xs <λr. r ↦a xs>"
  unfolding hoare_triple_def snga_assn_def one_assn_def
  apply (simp add: Let_def Abs_assn_inverse)
  apply (auto
    elim!: run_elims
    simp: Let_def new_addrs_def Array.get_def Array.set_def Array.alloc_def
      relH_def in_range.simps
  )
  done

lemma length_rule:
  "<a ↦a xs> Array.len a <λr. a ↦a xs * ↑(r = length xs)>"
  unfolding hoare_triple_def snga_assn_def
  apply (simp add: Let_def Abs_assn_inverse)
  apply (auto
    elim!: run_elims
```

```
    simp: Let_def new_addrs_def Array.get_def Array.set_def Array.alloc_def
      relH_def in_range.simps Array.length_def
  )
  done
```

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⟧ ⟹ <a ↦a xs> Array.nth a i <λr. a ↦a xs * ↑(r =
xs ! i)>"
  unfolding hoare_triple_def snga_assn_def
  apply (simp add: Let_def Abs_assn_inverse)
  apply (auto
    elim!: run_elims
    simp: Let_def new_addrs_def Array.get_def Array.set_def Array.alloc_def
      relH_def in_range.simps Array.length_def
  )
  done

lemma upd_rule:
  "⟦i < length xs⟧ ⟹
  <a ↦a xs>
  Array.upd i x a
  <λr. (a ↦a (list_update xs i x)) * ↑(r = a)>"
  unfolding hoare_triple_def snga_assn_def
  apply (simp add: Let_def Abs_assn_inverse)
  apply (auto
    elim!: run_elims
    simp: Let_def new_addrs_def Array.get_def Array.set_def Array.alloc_def
      relH_def in_range.simps Array.length_def Array.update_def comp_def
  )
  done

lemma freeze_rule:
  "<a ↦a xs> Array.freeze a <λr. a ↦a xs * ↑(r = xs)>"
  unfolding hoare_triple_def snga_assn_def
  apply (simp add: Let_def Abs_assn_inverse)
  apply (auto
    elim!: run_elims
    simp: Let_def new_addrs_def Array.get_def Array.set_def Array.alloc_def
      relH_def in_range.simps Array.length_def Array.update_def
  )
  done

lemma return_wp_rule:
  "<Q x> return x <Q>"
  unfolding hoare_triple_def Let_def
  apply (auto elim!: run_elims)
  apply (rule relH_refl)
```

```
  apply (simp add: in_range.simps)
  done

lemma return_sp_rule:
  "<P> return x <λr. P * ↑(r = x)>"
  unfolding hoare_triple_def Let_def
  apply (simp add: Abs_assn_inverse)
  apply (auto elim!: run_elims intro!: relH_refl intro: models_in_range)
  apply (simp add: in_range.simps)
  done

lemma raise_iff:
  "<P> raise s <Q> ⟷ P = false"
  unfolding hoare_triple_def Let_def
  apply (rule iffI)
  apply (unfold bot_assn_def) []
  apply rule
  apply (auto simp add: run_raise_iff) []

  apply (auto simp add: run_raise_iff) []
  done

lemma raise_rule: "<false> raise s <Q>"
  by (simp add: raise_iff)
```

4.2.3 Rules for Composed Commands

```
lemma bind_rule:
  assumes T1: "<P> f <R>"
  assumes T2: "⋀x. <R x> g x <Q>"
  shows "<P> bind f g <Q>"
  unfolding hoare_triple_def Let_def
  apply (intro allI impI)
  apply (elim conjE run_elims)
  apply (intro conjI)
proof -
  fix h as σ'' r'' σ' r'
  assume M: "(h,as) ⊨ P"
    and R1: "run f (Some h) σ' r'"
    and R2: "run (g r') σ' σ'' r''"

  from hoare_tripleD[OF T1 M R1] have NO_E: "¬ is_exn σ'"
    and M': "(the_state σ', new_addrs h as (the_state σ')) ⊨ R r'"
    and RH': "relH {a. a < lim h ∧ a ∉ as} h (the_state σ')"
    and LIM: "lim h ≤ lim (the_state σ')"
    by auto

  from NO_E have [simp]: "Some (the_state σ') = σ'" by (cases σ') auto
```

```
  from hoare_tripleD[OF T2 M', simplified, OF R2] have
    NO_E'': "¬ is_exn σ''"
    and M'': "(the_state σ'',
      new_addrs (the_state σ')
        (new_addrs h as (the_state σ')) (the_state σ''))
      ⊨ Q r''"
    and RH'':
    "relH
      {a. a < lim (the_state σ')
        ∧ a ∉ new_addrs h as (the_state σ')
      }
      (the_state σ') (the_state σ'')"
    and LIM': "lim (the_state σ') ≤ lim (the_state σ'')" by auto

  show "¬ is_exn σ''" by fact

  have
    "new_addrs
      (the_state σ')
      (new_addrs h as (the_state σ'))
      (the_state σ'')
    = new_addrs h as (the_state σ'')"
    using LIM LIM'
    by (auto simp add: new_addrs_def)
  with M'' show
    "(the_state σ'', new_addrs h as (the_state σ'')) ⊨ Q r''"
    by simp

  note RH'
  also have "relH {a. a < lim h ∧ a ∉ as} (the_state σ') (the_state σ'')"
    apply (rule relH_subset[OF RH''])
    using LIM LIM'
    by (auto simp: new_addrs_def)
  finally show "relH {a. a < lim h ∧ a ∉ as} h (the_state σ'')" .

  note LIM
  also note LIM'
  finally show "lim h ≤ lim (the_state σ'')" .
qed

lemma if_rule:
  assumes  "b ⟹ <P> f <Q>"
  assumes  "¬b ⟹ <P> g <Q>"
  shows "<P> if b then f else g <Q>"
  using assms by auto

lemma if_rule_split:
  assumes  B: "b ⟹ <P> f <Q1>"
  assumes  NB: "¬b ⟹ <P> g <Q2>"
```

assumes *M: "⋀x. (Q1 x * ↑b) ∨A (Q2 x * ↑(¬b)) ⟹A Q x"*
shows *"<P> if b then f else g <Q>"*
apply *(cases b)*
apply *simp_all*
apply *(rule cons_post_rule)*
apply *(erule B)*
apply *(rule ent_trans[OF _ ent_disjI1[OF M]])*
apply *simp*

apply *(rule cons_post_rule)*
apply *(erule NB)*
apply *(rule ent_trans[OF _ ent_disjI2[OF M]])*
apply *simp*
done

lemma *split_rule:*
assumes *P: "<P> c <R>"*
assumes *Q: "<Q> c <R>"*
shows *"<P ∨A Q> c <R>"*
unfolding *hoare_triple_def*
apply *(intro allI impI)*
apply *(elim conjE)*
apply *simp*
apply *(erule disjE)*
using *hoare_tripleD[OF P]* **apply** *simp*
using *hoare_tripleD[OF Q]* **apply** *simp*
done

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

lemma *case_prod_rule:*
"(⋀a b. x = (a, b) ⟹ <P> f a b <Q>) ⟹ <P> case x of (a, b) ⇒ f a b <Q>"
by *(auto split: prod.split)*

lemma *case_list_rule:*
"⟦ l=[] ⟹ <P> fn <Q>; ⋀x xs. l=x#xs ⟹ <P> fc x xs <Q> ⟧ ⟹ <P> case_list fn fc l <Q>"
by *(auto split: list.split)*

lemma *case_option_rule:*
"⟦ v=None ⟹ <P> fn <Q>; ⋀x. v=Some x ⟹ <P> fs x <Q> ⟧ ⟹ <P> case_option fn fs v <Q>"
by *(auto split: option.split)*

lemma *case_sum_rule:*
"⟦ ⋀x. v=Inl x ⟹ <P> fl x <Q>;
⋀x. v=Inr x ⟹ <P> fr x <Q> ⟧

```
  ⟹ <P> case_sum fl fr v <Q>"
  by (auto split: sum.split)

lemma let_rule: "(⋀x. x = t ⟹ <P> f x <Q>) ⟹ <P> Let t f <Q>"
  by (auto)

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"
  by (simp add: mult.left_ac)

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]:
  "⋀Q R. (∃A x. Q x) * R = (∃A x. (Q x * R))"
  "⋀Q R. R * (∃A x. Q x) = (∃A x. (R * Q x))"

  "⋀P Q. (∃A x. Q x) ∧A P = (∃A x. (Q x ∧A P)) "
```

```
  "⋀P Q. Q ∧A (∃Ax. P x) = (∃Ax. (Q ∧A P x))"

  "⋀P Q. (∃Ax. Q x) ∨A P = (∃Ax. (Q x ∨A P))"
  "⋀P Q. Q ∨A (∃Ax. P x) = (∃Ax. (Q ∨A P x))"
  apply -
  apply (simp add: ex_distrib_star)
  apply (subst mult.commute)
  apply (subst (2) mult.commute)
  apply (simp add: ex_distrib_star)

  apply (simp add: ex_distrib_and)
  apply (subst inf_commute)
  apply (subst (2) inf_commute)
  apply (simp add: ex_distrib_and)

  apply (simp add: ex_distrib_or)
  apply (subst sup_commute)
  apply (subst (2) sup_commute)
  apply (simp add: ex_distrib_or)
  done
```

Extract pure assertions from and-clauses

```
lemma and_extract_pure_left_iff[simp]: "↑b ∧A Q = (emp∧AQ)*↑b"
  by (cases b) auto

lemma and_extract_pure_left_ctx_iff[simp]: "P*↑b ∧A Q = (P∧AQ)*↑b"
  by (cases b) auto

lemma and_extract_pure_right_iff[simp]: "P ∧A ↑b = (emp∧AP)*↑b"
  by (cases b) (auto simp: assn_aci)

lemma and_extract_pure_right_ctx_iff[simp]: "P ∧A Q*↑b = (P∧AQ)*↑b"
  by (cases b) auto

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"
  by (metis Ref.unequal)
declare Ref.noteq_irrefl[dest!]

lemma not_same_noteqa[simp]: "¬ a=!!=a"
  by (metis Array.unequal)
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':
  "(⋀x. Z x ⟶ (P ⟹_A Q x)) ⟹ (∃x. Z x) ⟶ (P ⟹_A (∃_A x. Q x))"
  by (metis ent_ex_postI)
```

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> ⟹ <P> c <Q>"* .

Dummy rule to detect entailment goal

lemma *is_entails: "P⟹$_A$Q ⟹ P ⟹$_A$Q"* .

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 ≡ emp"*
lemma *SLN_left: "SLN * P = P"* **by** *simp*
lemma *SLN_right: "P * SLN = P"* **by** *simp*

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 ≡ P ⟹$_A$ Q*F"*

abbreviation *"fi_m_fst M ≡ foldr (*) (map fst M) emp"*
abbreviation *"fi_m_snd M ≡ foldr (*) (map snd M) emp"*
abbreviation *"fi_m_match M ≡ (∀(p,q)∈set M. p ⟹$_A$ 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 ≡*
*fi_m_match M ⟶ (fi_m_fst M * UP ⟹$_A$ 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 ≡*
*fi_m_match m ⟶ (fi_m_fst m * p * up ⟹$_A$ 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"*
using *assms* **by** *simp*

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"*
  using *assms* **by** *(simp add: assn_aci)*

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* $\Longrightarrow_A$ *fi_m_snd m"*
  using *assms* **apply** *(induct m)*
  apply *(simp_all split: prod.split_asm add: ent_star_mono)*
  done

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* $\Longrightarrow_A$ *q"*
  assumes *"FI ((p,q)#m) (ps*up) (qs*uq) SLN SLN f"*
  shows *"FI m (ps*p) (qs*q) up uq f"*
  using *assms* **unfolding** *FI_def*
  by *(simp add: assn_aci)*

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"*
  using *assms* **unfolding** *FI_def*
  by *(simp add: assn_aci)*

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"*
  using *assms* **unfolding** *FI_def*
  by *(simp add: assn_aci)*

5.4 Frame Inference

lemma *frame_inference_init:*
  assumes *"FI_QUERY P Q F"*
  shows *"P* $\Longrightarrow_A$ *Q * F"*
  using *assms* **by** *simp*

lemma *frame_inference_finalize:*
shows *"FI_RESULT M F emp F"*
apply *simp*
apply *rule*
apply *(drule fi_match_entails)*
apply *(rule ent_star_mono[OF _ ent_refl])*
apply *assumption*
done

5.5 Entailment Solver

lemma *entails_solve_init:*
*"FI_QUERY P Q true ⟹ P ⟹$_A$ Q * true"*
"FI_QUERY P Q emp ⟹ P ⟹$_A$ Q"
by *(simp_all add: assn_aci)*

lemma *entails_solve_finalize:*
"FI_RESULT M P emp true"
"FI_RESULT M emp emp emp"
by *(auto simp add: fi_match_entails intro: ent_star_mono)*

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> ⟹ <R * P> c <λx. R * Q x>"*
using *frame_rule* **by** *(simp add: assn_aci)*

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: "Ps ⟹_A P * F"
  shows "<Ps> c <λx. Q x * F>"
  apply (rule cons_pre_rule[rotated])
  apply (rule frame_rule)
  apply (rule CMD)
  apply (rule FRAME)
  done
```

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 ‹
infix 1 THEN_IGNORE_NEWGOALS

structure Seplogic_Auto =
struct

  (**********************************)
  (*             Tools              *)
  (**********************************)

  (* Repeat tac on subgoal. Determinize each step.
     Stop if tac fails or subgoal is solved. *)
  fun REPEAT_DETERM' tac i st = let
    val n = Thm.nprems_of st
  in
    REPEAT_DETERM (COND (has_fewer_prems n) no_tac (tac i)) st
  end


  (**********************************)
  (*             Debugging          *)
  (**********************************)
  fun tr_term t = Pretty.string_of (Syntax.pretty_term @{context} t);


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

```
(*        Custom Tacticals         *)
(************************************)

(* Apply tac1, and then tac2 with an offset such that anything left

   over by tac1 is skipped.

   The typical usage of this tactic is, if a theorem is instantiated
   with another theorem that produces additional goals that should
   be ignored first. Here, it is used in the vcg to ensure that
   frame inference is done before additional premises (that may
   depend on the frame) are discharged.
*)
fun (tac1 THEN_IGNORE_NEWGOALS tac2) i st = let
  val np = Thm.nprems_of st
in
  (tac1 i THEN (fn st' => let val np' = Thm.nprems_of st' in
    if np'<np then tac2 i st'
    else tac2 (i+(np'-np)+1) st'
  end)) st
end;


(************************************)
(*     Assertion Normalization      *)
(************************************)
(* Find two terms in a list whose key is equal *)
fun find_similar (key_of:term -> term) (ts:term list) = let
  fun frec _ [] = NONE
  | frec tab (t::ts) = let val k=key_of t in
    if Termtab.defined tab k then
      SOME (the (Termtab.lookup tab k),t)
    else frec (Termtab.update (k,t) tab) ts
  end
in
  frec Termtab.empty ts
end;

(* Perform DFS over term with binary operator opN, threading through
  a state. Atomic terms are transformed by tr. Supports omission of
  terms from the result structure by transforming them to NONE. *)
fun dfs_opr opN (tr:'state -> term -> ('state*term option))
  d (t as ((op_t as Const (fN,_))$t1$t2)) =
  if fN = opN then let
      val (d1,t1') = dfs_opr opN tr d t1;
      val (d2,t2') = dfs_opr opN tr d1 t2;
    in
      case (t1',t2') of
        (NONE,NONE) => (d2,NONE)
```

```
        | (SOME t1',NONE) => (d2,SOME t1')
        | (NONE,SOME t2') => (d2,SOME t2')
        | (SOME t1',SOME t2') => (d2,SOME (op_t$t1'$t2'))
      end
    else tr d t
  | dfs_opr _ tr d t = tr d t;

  (* Replace single occurrence of (atomic) ot in t by nt.
    Returns new term or NONE if nothing was removed. *)
  fun dfs_replace_atomic opN ot nt t = let
    fun tr d t = if not d andalso t=ot then (true,SOME nt) else (d,SOME
t);
    val (success,SOME t') = dfs_opr opN tr false t;
  in
    if success then SOME t' else NONE
  end;

  fun assn_simproc_fun ctxt credex = try (let
    val ([redex],ctxt') = Variable.import_terms true [Thm.term_of credex]
ctxt;
    (*val _ = tracing (tr_term redex);*)
    val export = singleton (Variable.export ctxt' ctxt)

    fun mk_star t1 t2 = @{term "(*)::assn ⇒ _ ⇒ _"}$t2$t1;

    fun mk_star' NONE NONE = NONE
    | mk_star' (SOME t1) NONE  = SOME t1
    | mk_star' NONE (SOME t2) = SOME t2
    | mk_star' (SOME t1) (SOME t2) = SOME (mk_star t1 t2);

    fun ptrs_key (_$k$_) = k;

    fun remove_term pt t = case
      dfs_replace_atomic @{const_name "Groups.times_class.times"} pt
        @{term emp} t
    of
      SOME t' => t';

    fun normalize t = let

      fun ep_tr (has_true,ps,ptrs) t = case t of
        Const (@{const_name "Assertions.pure_assn"},_)$_
        => ((has_true,t::ps,ptrs),NONE)
      | Const (@{const_name "Assertions.sngr_assn"},_)$_$_
        => ((has_true,ps,t::ptrs),SOME t)
      | Const (@{const_name "Assertions.snga_assn"},_)$_$_
        => ((has_true,ps,t::ptrs),SOME t)
      | Const (@{const_name "Orderings.top_class.top"},_)
        => ((true,ps,ptrs),NONE)
```

```
      | (inf_op as Const (@{const_name "Lattices.inf_class.inf"},_))$t1$t2
        => ((has_true,ps,ptrs),SOME (inf_op$normalize t1$normalize t2))
      | _ => ((has_true,ps,ptrs),SOME t);

      fun normalizer t = case dfs_opr @{const_name "Groups.times_class.times"}
        ep_tr (false,[],[]) t
      of
        ((has_true,ps,ptrs),rt) => ((has_true,rev ps,ptrs),rt);

      fun normalize_core t = let
        val ((has_true,pures,ptrs),rt) = normalizer t;
        val similar = find_similar ptrs_key ptrs;
        val true_t = if has_true then SOME @{term "Assertions.top_assn"}

          else NONE;
        val pures' = case pures of
            [] => NONE
          | p::ps => SOME (fold mk_star ps p);
      in
        case similar of NONE => the (mk_star' pures' (mk_star' true_t
rt))
        | SOME (t1,t2) => let
            val t_stripped = remove_term t1 (remove_term t2 t);
          in mk_star t_stripped (mk_star t1 t2) end
      end;

      fun skip_ex ((exq as Const (@{const_name "ex_assn"},_))$(Abs (n,ty,t)))
=
        exq$Abs (n,ty,skip_ex t)
      | skip_ex t = normalize_core t;

      val (bs,t') = strip_abs t;
      val ty = fastype_of1 (map #2 bs,t');
    in
      if ty = @{typ assn} then
        Logic.rlist_abs (bs,skip_ex t')
      else t
    end;

    (*val _ = tracing (tr_term redex);*)
    val (f,terms) = strip_comb redex;
    val nterms = map (fn t => let
        (*val _ = tracing (tr_term t); *)
        val t'=normalize t;
        (*val _ = tracing (tr_term t');*)
      in t' end) terms;
    val new_form = list_comb (f,nterms);

    val res_ss = (put_simpset HOL_basic_ss ctxt addsimps @{thms star_aci});
```

```
    val result = Option.map (export o mk_meta_eq) (Arith_Data.prove_conv_nohyps
      [simp_tac res_ss 1] ctxt' (redex,new_form)
    );

  in
    result
  end catch exc =>
    (tracing ("assn_simproc failed with exception\n:" ^ Runtime.exn_message
exc);
      NONE) (* Fail silently *)
  ›

  val assn_simproc =
    simproc_setup ‹passive assn
      ("h ⊨ P" | "P ⟹A Q" | "P ⟹t Q" | "Hoare_Triple.hoare_triple
P c R" | "P = Q") =
      ‹K assn_simproc_fun››;


  (***********************************)
  (*     Default Simplifications     *)
  (***********************************)

  (* Default simplification. MUST contain assertion normalization!
    Tactic must not fail! *)
  fun dflt_tac ctxt = asm_full_simp_tac
    (put_simpset HOL_ss ctxt
      addsimprocs [assn_simproc]
      addsimps @{thms norm_assertion_simps}
      addsimps (Named_Theorems.get ctxt @{named_theorems sep_dflt_simps})
      |> fold Splitter.del_split @{thms if_split}
    );

  (***********************************)
  (*         Frame Matcher           *)
  (***********************************)

  (* Do frame matching
    imp_solve_tac - tactic used to discharge first assumption of match-rule
      cf. lemma FI_match.
  *)
  fun match_frame_tac imp_solve_tac ctxt = let
    (* Normalize star-lists *)
    val norm_tac = simp_tac (
      put_simpset HOL_basic_ss ctxt addsimps @{thms SLN_normalize});

    (* Strip star-lists *)
    val strip_tac =
      simp_tac (put_simpset HOL_basic_ss ctxt addsimps @{thms SLN_strip})
```

```
THEN'
      simp_tac (put_simpset HOL_basic_ss ctxt addsimps @{thms SLN_def});

    (* Do a match step *)
    val match_tac = resolve_tac ctxt @{thms FI_match} (* Separate p,q*)
      THEN' SOLVED' imp_solve_tac (* Solve implication *)
      THEN' norm_tac;

    (* Do a no-match step *)
    val nomatch_tac = resolve_tac ctxt @{thms FI_p_nomatch} ORELSE'
      (resolve_tac ctxt @{thms FI_q_nomatch} THEN' norm_tac);
  in
    resolve_tac ctxt @{thms FI_init} THEN' norm_tac
    THEN' REPEAT_DETERM' (FIRST' [
      CHANGED o dflt_tac ctxt,
      (match_tac ORELSE' nomatch_tac)])
    THEN' resolve_tac ctxt @{thms FI_finalize} THEN' strip_tac
  end;


  (***********************************)
  (*         Frame Inference         *)
  (***********************************)

  fun frame_inference_tac ctxt =
    resolve_tac ctxt @{thms frame_inference_init}
    THEN' match_frame_tac (resolve_tac ctxt @{thms ent_refl}) ctxt
    THEN' resolve_tac ctxt @{thms frame_inference_finalize};


  (***********************************)
  (*       Entailment Solver         *)
  (***********************************)

  (* Extract existential quantifiers from entailment goal *)
  fun extract_ex_tac ctxt i st = let
    fun count_ex (Const (@{const_name Assertions.entails},_)$_$c) =
      count_ex c RS @{thm HOL.mp}
    | count_ex (Const (@{const_name Assertions.ex_assn},_)$Abs (_,_,t))
      = count_ex t RS @{thm enorm_exI'}
    | count_ex _ = @{thm imp_refl};

    val concl = Logic.concl_of_goal (Thm.prop_of st) i |> HOLogic.dest_Trueprop;
    val thm = count_ex concl;
  in
    (TRY o REPEAT_ALL_NEW (match_tac ctxt @{thms ent_ex_preI}) THEN'
     resolve_tac ctxt [thm]) i st
  end;
```

```
  (* Solve Entailment *)
  fun solve_entails_tac ctxt = let
    val preprocess_entails_tac =
      dflt_tac ctxt
      THEN' extract_ex_tac ctxt
      THEN' simp_tac
        (put_simpset HOL_ss ctxt addsimps @{thms solve_ent_preprocess_simps});

    val match_entails_tac =
      resolve_tac ctxt @{thms entails_solve_init}
      THEN' match_frame_tac (resolve_tac ctxt @{thms ent_refl}) ctxt
      THEN' resolve_tac ctxt @{thms entails_solve_finalize};
  in
    preprocess_entails_tac
    THEN' (TRY o
      REPEAT_ALL_NEW (match_tac ctxt (rev (Named_Theorems.get ctxt @{named_theorems
sep_eintros}))))
    THEN_ALL_NEW (dflt_tac ctxt THEN'
      TRY o (match_tac ctxt @{thms ent_triv}
        ORELSE' resolve_tac ctxt @{thms ent_refl}
        ORELSE' match_entails_tac))
  end;


  (***********************************)
  (* Verification Condition Generator*)
  (***********************************)

  fun heap_rule_tac ctxt h_thms =
    resolve_tac ctxt h_thms ORELSE' (
    resolve_tac ctxt @{thms fi_rule} THEN' (resolve_tac ctxt h_thms THEN_IGNORE_NEWGOALS
    frame_inference_tac ctxt));

  fun vcg_step_tac ctxt = let
    val h_thms = rev (Named_Theorems.get ctxt @{named_theorems sep_heap_rules});
    val d_thms = rev (Named_Theorems.get ctxt @{named_theorems sep_decon_rules});
    val heap_rule_tac = heap_rule_tac ctxt h_thms

    (* Apply consequence rule if postcondition is not a schematic var
*)
    fun app_post_cons_tac i st =
      case Logic.concl_of_goal (Thm.prop_of st) i |> HOLogic.dest_Trueprop
of
        Const (@{const_name Hoare_Triple.hoare_triple},_)$_$_$qt =>
          if is_Var (head_of qt) then no_tac st
          else resolve_tac ctxt @{thms cons_post_rule} i st
      | _ => no_tac st;
```

```
  in
    CSUBGOAL (snd #> (FIRST' [
      CHANGED o dflt_tac ctxt,
      REPEAT_ALL_NEW (resolve_tac ctxt @{thms normalize_rules}),
      CHANGED o (FIRST' [resolve_tac ctxt d_thms, heap_rule_tac]
        ORELSE' (app_post_cons_tac THEN'
          FIRST' [resolve_tac ctxt d_thms, heap_rule_tac]))
    ]))
  end;

  fun vcg_tac ctxt = REPEAT_DETERM' (vcg_step_tac ctxt)

  (***********************************)
  (*        Automatic Solver         *)
  (***********************************)

  fun sep_autosolve_tac do_pre do_post ctxt = let
    val pre_tacs = [
      CHANGED o clarsimp_tac ctxt,
      CHANGED o REPEAT_ALL_NEW (match_tac ctxt @{thms ballI allI impI
conjI})
    ];
    val main_tacs = [
      match_tac ctxt @{thms is_hoare_triple} THEN' CHANGED o vcg_tac ctxt,
      match_tac ctxt @{thms is_entails} THEN' CHANGED o solve_entails_tac
ctxt
    ];
    val post_tacs = [SELECT_GOAL (auto_tac ctxt)];
    val tacs = (if do_pre then pre_tacs else [])
      @ main_tacs
      @ (if do_post then post_tacs else []);
  in
    REPEAT_DETERM' (CHANGED o FIRST' tacs)
  end;


  (***********************************)
  (*          Method Setup           *)
  (***********************************)

  val dflt_simps_modifiers = [
    Args.$$$ "dflt_simps" -- Scan.option Args.add -- Args.colon
      >> K (Method.modifier (Named_Theorems.add @{named_theorems sep_dflt_simps})
here),
    Args.$$$ "dflt_simps" -- Scan.option Args.del -- Args.colon
      >> K (Method.modifier (Named_Theorems.del @{named_theorems sep_dflt_simps})
here)
  ];
  val heap_modifiers = [
```

```
    Args.$$$ "heap" -- Scan.option Args.add -- Args.colon
      >> K (Method.modifier (Named_Theorems.add @{named_theorems sep_heap_rules})
here),
    Args.$$$ "heap" -- Scan.option Args.del -- Args.colon
      >> K (Method.modifier (Named_Theorems.del @{named_theorems sep_heap_rules})
here)
  ];
  val decon_modifiers = [
    Args.$$$ "decon" -- Scan.option Args.add -- Args.colon
      >> K (Method.modifier (Named_Theorems.add @{named_theorems sep_decon_rules})
here),
    Args.$$$ "decon" -- Scan.option Args.del -- Args.colon
      >> K (Method.modifier (Named_Theorems.del @{named_theorems sep_decon_rules})
here)
  ];

  val eintros_modifiers = [
    Args.$$$ "eintros" -- Scan.option Args.add -- Args.colon
      >> K (Method.modifier (Named_Theorems.add @{named_theorems sep_eintros})
here),
    Args.$$$ "eintros" -- Scan.option Args.del -- Args.colon
      >> K (Method.modifier (Named_Theorems.del @{named_theorems sep_eintros})
here)
  ];


  val solve_entails_modifiers = dflt_simps_modifiers @ eintros_modifiers;

  val vcg_modifiers =
    heap_modifiers @ decon_modifiers @ dflt_simps_modifiers;

  val sep_auto_modifiers =
    clasimp_modifiers @ vcg_modifiers @ eintros_modifiers;

end;
›

simproc_setup assn_simproc
  ("h⊨P" | "P⟹_AQ" | "P⟹_tQ" | "<P> c <R>" | "(P::assn) = Q")
  = ‹K Seplogic_Auto.assn_simproc_fun›

method_setup assn_simp =‹Scan.succeed (fn ctxt => (SIMPLE_METHOD' (
  CHANGED o Seplogic_Auto.dflt_tac ctxt
)))› "Seplogic: Simplification of assertions"

method_setup frame_inference = ‹Scan.succeed (fn ctxt => (SIMPLE_METHOD'
(
  CHANGED o Seplogic_Auto.frame_inference_tac ctxt
)))› "Seplogic: Frame inference"
```

```
method_setup solve_entails = <
  Method.sections Seplogic_Auto.solve_entails_modifiers >>
  (fn _ => fn ctxt => SIMPLE_METHOD' (
  CHANGED o Seplogic_Auto.solve_entails_tac ctxt
))> "Seplogic: Entailment Solver"

method_setup heap_rule = <
  Attrib.thms >>
  (fn thms => fn ctxt => SIMPLE_METHOD' (
    let
      val thms = case thms of [] => rev (Named_Theorems.get ctxt @{named_theorems
sep_heap_rules})
        | _ => thms
    in
      CHANGED o Seplogic_Auto.heap_rule_tac ctxt thms
    end
))> "Seplogic: Apply rule with frame inference"


method_setup vcg = <
  Scan.lift (Args.mode "ss") --
  Method.sections Seplogic_Auto.vcg_modifiers >>
  (fn (ss,_) => fn ctxt => SIMPLE_METHOD' (
  CHANGED o (
    if ss then Seplogic_Auto.vcg_step_tac ctxt
    else Seplogic_Auto.vcg_tac ctxt
  )
))> "Seplogic: Verification Condition Generator"

method_setup sep_auto =
  <Scan.lift (Args.mode "nopre" -- Args.mode "nopost" -- Args.mode "plain")

      --| Method.sections Seplogic_Auto.sep_auto_modifiers >>
  (fn ((nopre,nopost),plain) => fn ctxt => SIMPLE_METHOD' (
    CHANGED o Seplogic_Auto.sep_autosolve_tac
      ((not nopre) andalso (not plain))
      ((not nopost) andalso (not plain)) ctxt
  ))> "Seplogic: Automatic solver"

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 ⟹A R"
  assumes F: "Ps ⟹A P*F"
  assumes I: "R*F ⟹A Q"
  shows "Ps ⟹A Q"
  using assms
  by (metis ent_refl ent_star_mono ent_trans)

lemma mod_frame_fwd:
  assumes M: "h⊨Ps"
  assumes R: "P⟹AR"
  assumes F: "Ps ⟹A P*F"
  shows "h⊨R*F"
  using assms
  by (metis ent_star_mono entails_def)
```

Apply precision rule with frame inference.

```
lemma prec_frame:
  assumes PREC: "precise P"
  assumes M1: "h⊨(R1 ∧A R2)"
  assumes F1: "R1 ⟹A P x p * F1"
  assumes F2: "R2 ⟹A P y p * F2"
  shows "x=y"
  using preciseD[OF PREC] M1 F1 F2
  by (metis entailsD mod_and_dist)

lemma prec_frame_expl:
  assumes PREC: "∀x y. (h⊨(P x * F1) ∧A (P y * F2)) ⟶ x=y"
  assumes M1: "h⊨(R1 ∧A R2)"
  assumes F1: "R1 ⟹A P x * F1"
  assumes F2: "R2 ⟹A P y * F2"
  shows "x=y"
  using assms
  by (metis entailsD mod_and_dist)
```

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

```
lemma prec_frame':
  assumes PREC: "(h⊨(P x * F1) ∧A (P y * F2)) ⟶ x=y"
  assumes M1: "h⊨(R1 ∧A R2)"
  assumes F1: "R1 ⟹A P x * F1"
  assumes F2: "R2 ⟹A P y * F2"
  shows "x=y"
```

```
  using assms
  by (metis entailsD mod_and_dist)


lemma ent_wand_frameI:
  assumes "(Q -* R) * F ⟹A S"
  assumes "P ⟹A F * X"
  assumes "Q*X ⟹A R"
  shows "P ⟹A S"
  using assms
  by (metis ent_frame_fwd ent_wandI mult.commute)
```

5.8.1 Manual Frame Inference

```
lemma ent_true_drop:
  "P⟹AQ*true ⟹ P*R⟹AQ*true"
  "P⟹AQ ⟹ P⟹AQ*true"
  apply (metis assn_times_comm ent_star_mono ent_true merge_true_star_ctx)
  apply (metis assn_one_left ent_star_mono ent_true star_aci(2))
  done

lemma fr_refl: "A⟹AB ⟹ A*C ⟹AB*C"
  by (blast intro: ent_star_mono ent_refl)

lemma fr_rot: "(A*B ⟹A C) ⟹ (B*A ⟹A C)"
  by (simp add: assn_aci)

lemma fr_rot_rhs: "(A ⟹A B*C) ⟹ (A ⟹A C*B)"
  by (simp add: assn_aci)

lemma ent_star_mono_true:
  assumes "A ⟹A A' * true"
  assumes "B ⟹A B' * true"
  shows "A*B*true ⟹A A'*B'*true"
  using ent_star_mono[OF assms] apply simp
  using ent_true_drop(1) by blast

lemma ent_refl_true: "A ⟹A A * true"
  by (simp add: ent_true_drop(2))

lemma entt_fr_refl: "F⟹tF' ⟹ F*A ⟹t F'*A" by (rule entt_star_mono)
auto
lemma entt_fr_drop: "F⟹tF' ⟹ F*A ⟹t F'"
  using ent_true_drop(1) enttD enttI by blast


method_setup fr_rot = <
  let
    fun rot_tac ctxt =
```

```
      resolve_tac ctxt @{thms fr_rot} THEN'
      simp_tac (put_simpset HOL_basic_ss ctxt
        addsimps @{thms star_assoc[symmetric]})

  in
    Scan.lift Parse.nat >>
      (fn n => fn ctxt => SIMPLE_METHOD' (
        fn i => REPEAT_DETERM_N n (rot_tac ctxt i)))

  end
>

method_setup fr_rot_rhs = <
  let
    fun rot_tac ctxt =
      resolve_tac ctxt @{thms fr_rot_rhs} THEN'
      simp_tac (put_simpset HOL_basic_ss ctxt
        addsimps @{thms star_assoc[symmetric]})

  in
    Scan.lift Parse.nat >>
      (fn n => fn ctxt => SIMPLE_METHOD' (
        fn i => REPEAT_DETERM_N n (rot_tac ctxt i)))

  end
>
```

5.9 Quick Overview of Proof Methods

In this section, we give a quick overview of the available proof methods and options. The most versatile proof method that we provide is *sep_auto*. It tries to solve the first subgoal, invoking appropriate proof methods as required. If it cannot solve the subgoal completely, it stops at the intermediate state that it could not handle any more.

sep_auto can be configured by section-arguments for the simplifier, the classical reasoner, and all section-arguments for the verification condition generator and entailment solver. Moreover, it takes an optional mode argument (mode), where valid modes are:

(nopre) No preprocessing of goal. The preprocessor tries to clarify and simplify the goal before the main method is invoked.

(nopost) No postprocessing of goal. The postprocessor tries to solve or 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 $P \Longrightarrow_A Q$. 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 $X_1 * \ldots * X_n$. 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 $P \Longrightarrow Q * ?F$, 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 *<P>c<Q>*. 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 ⇒ 'l ⇒ assn"*
  assumes *precise: "precise is_list"*

locale *imp_list_empty = imp_list +*
  constrains *is_list :: "'a list ⇒ 'l ⇒ 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 ⇒ 'l ⇒ assn"*
  fixes *is_empty :: "'l ⇒ bool Heap"*
  assumes *is_empty_rule[sep_heap_rules]:*
    *"<is_list l p> is_empty p <λr. is_list l p * ↑(r ⟷ l=[])>$_t$"*

locale *imp_list_append = imp_list +*
  constrains *is_list :: "'a list ⇒ 'l ⇒ assn"*
  fixes *append :: "'a ⇒ 'l ⇒ '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 ⇒ 'l ⇒ assn"*
  fixes *prepend :: "'a ⇒ 'l ⇒ '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 ⇒ 'l ⇒ 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.

```
setup ‹Sign.add_const_constraint
  (@{const_name Ref}, SOME @{typ "nat ⇒ 'a::type ref"})›

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

setup ‹Sign.add_const_constraint
  (@{const_name Ref}, SOME @{typ "nat ⇒ 'a::heap ref"})›

setup ‹Sign.add_const_constraint (@{const_name Node},
  SOME @{typ "'a::heap ⇒ 'a node ref option ⇒ 'a node"})›
```

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
  apply (rule heap_class.intro)
  apply (rule countable_classI [of "node_encode"])
  apply (case_tac x, simp_all, case_tac y, simp_all)
  ..
```

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 = (∃_A q. 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=[])"
  apply (cases l, simp_all)
  done

lemma lseg_if_splitf2[simp, sep_dflt_simps]:
  "lseg (x#xs) p q
    = (∃_A pp n. pp ↦_r (Node x n) * lseg xs n q * ↑(p=Some pp))"
  apply (cases p, simp_all)

  apply (rule ent_iffI)
  apply solve_entails
  apply solve_entails
  done
```

8.3 Lemmas

8.3.1 Concatenation

```
lemma lseg_prepend:
  "p↦_rNode x q * lseg l q s ⟹_A lseg (x#l) (Some p) s"
  by sep_auto

lemma lseg_append:
  "lseg l p (Some s) * s↦_rNode x q ⟹_A lseg (l@[x]) p q"
proof (induction l arbitrary: p)
  case Nil thus ?case by sep_auto
next
  case (Cons y l)
  show ?case
    apply (cases p)
    apply simp
    apply (sep_auto)
    apply (rule ent_frame_fwd[OF Cons.IH])
    apply frame_inference
    apply solve_entails
    done
qed

lemma lseg_conc: "lseg l1 p q * lseg l2 q r ⟹_A lseg (l1@l2) p r"
proof (induct l1 arbitrary: p)
  case Nil thus ?case by simp
next
  case (Cons x l1)
  show ?case
```

```
    apply simp
    apply sep_auto
    apply (rule ent_frame_fwd[OF Cons.hyps])
    apply frame_inference
    apply solve_entails
    done
qed
```

8.3.2 Splitting

```
lemma lseg_split:
  "lseg (l1@l2) p r ⟹A ∃Aq. lseg l1 p q * lseg l2 q r"
proof (induct l1 arbitrary: p)
  case Nil thus ?case by sep_auto
next
  case (Cons x l1)

  have "lseg ((x # l1) @ l2) p r
    ⟹A ∃App n. pp ↦r Node x n * lseg (l1 @ l2) n r * ↑ (p = Some pp)"
    by simp
  also have "... ⟹A
    ∃App n q. pp ↦r Node x n
      * lseg l1 n q
      * lseg l2 q r
      * ↑(p = Some pp)"
    apply (intro ent_ex_preI)
    apply clarsimp
    apply (rule ent_frame_fwd[OF Cons.hyps])
    apply frame_inference
    apply sep_auto
    done
  also have "... ⟹A ∃Aq. lseg (x#l1) p q * lseg l2 q r"
    by sep_auto
  finally show ?case .
qed
```

8.3.3 Precision

```
lemma lseg_prec1:
  "∀l l'. (h⊨
      (lseg l p (Some q) * q ↦r x * F1)
       ∧A (lseg l' p (Some q) * q ↦r x * F2))
    ⟶ l=l'"
  apply (intro allI)
  subgoal for l l'
  proof (induct l arbitrary: p l' F1 F2)
    case Nil thus ?case
      apply simp_all
      apply (cases l')
      apply simp
```

```
    apply auto
    done
  next
    case (Cons y l)
    from Cons.prems show ?case
    apply (cases l')
    apply auto []
    apply (cases p)
    apply simp

    apply (clarsimp)

    apply (subgoal_tac "y=a ∧ na=n", simp)

    using Cons.hyps apply (erule prec_frame')
    apply frame_inference
    apply frame_inference

    apply (drule_tac p=aa in prec_frame[OF sngr_prec])
    apply frame_inference
    apply frame_inference
    apply simp
    done
  qed
  done

lemma lseg_prec2:
  "∀l l'. (h⊨
     (lseg l p None * F1) ∧_A (lseg l' p None * F2))
    ⟶ l=l'"
  apply (intro allI)
  subgoal for l l'
  proof (induct l arbitrary: p l' F1 F2)
    case Nil thus ?case
    apply simp_all
    apply (cases l')
    apply simp
    apply (cases p)
    apply auto
    done
  next
    case (Cons y l)
    from Cons.prems show ?case
    apply (cases p)
    apply simp
    apply (cases l')
    apply (auto) []

    apply (clarsimp)
```

```
    apply (subgoal_tac "y=aa ∧ na=n", simp)

    using Cons.hyps apply (erule prec_frame')
    apply frame_inference
    apply frame_inference

    apply (drule_tac p=a in prec_frame[OF sngr_prec])
    apply frame_inference
    apply frame_inference
    apply simp
    done
  qed
  done

lemma lseg_prec3:
  "∀q q'. h ⊨ (lseg l p q * F1) ∧A (lseg l p q' * F2) ⟶ q=q'"
  apply (intro allI)
proof (induct l arbitrary: p F1 F2)
  case Nil thus ?case by auto
next
  case (Cons x l)
  show ?case
    apply auto

    apply (subgoal_tac "na=n")
    using Cons.hyps apply (erule prec_frame')
    apply frame_inference
    apply frame_inference

    apply (drule prec_frame[OF sngr_prec])
    apply frame_inference
    apply frame_inference
    apply simp
    done
qed

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 ⇒ ('a::heap) os_list ⇒ assn"* **where**
"os_list l p ≡ lseg l p None"

9.2 Precision

lemma *os_prec:*
"precise os_list"
by *rule (simp add: lseg_prec2)*

lemma *os_imp_list_impl: "imp_list os_list"*
apply *unfold_locales*
apply *(rule os_prec)*
done
interpretation *os: imp_list os_list* **by** *(rule os_imp_list_impl)*

9.3 Operations

9.3.1 Allocate Empty List

definition *os_empty :: "'a::heap os_list Heap"* **where**
"os_empty ≡ return None"

lemma *os_empty_rule: "<emp> os_empty <os_list []>"*
unfolding *os_empty_def*
apply *sep_auto*
done

lemma *os_empty_impl: "imp_list_empty os_list os_empty"*
apply *unfold_locales*
apply *(sep_auto heap add: os_empty_rule)*
done
interpretation *os: imp_list_empty os_list os_empty* **by** *(rule os_empty_impl)*

9.3.2 Emptiness check

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

definition *os_is_empty :: "'a::heap os_list ⇒ bool Heap"* **where**
"os_is_empty b ≡ return (b = None)"

lemma *os_is_empty_rule:*
*"<os_list xs b> os_is_empty b <λr. os_list xs b * ↑(r ⟷ xs = [])>"*
unfolding *os_is_empty_def*
apply *sep_auto*
done

lemma *os_is_empty_impl: "imp_list_is_empty os_list os_is_empty"*
apply *unfold_locales*
apply *(sep_auto heap add: os_is_empty_rule)*
done

```
interpretation os: imp_list_is_empty os_list os_is_empty
  by (rule os_is_empty_impl)
```

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)>"
  unfolding os_prepend_def
  apply sep_auto
  done

lemma os_prepend_impl: "imp_list_prepend os_list os_prepend"
  apply unfold_locales
  apply (sep_auto heap add: os_prepend_rule)
  done
interpretation os: imp_list_prepend os_list os_prepend
  by (rule os_prepend_impl)
```

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)>"

  apply (cases r, simp_all)
  apply (cases xs, simp_all)

  apply (sep_auto simp: os_pop.simps)
  done

lemma os_pop_impl: "imp_list_pop os_list os_pop"
  apply unfold_locales
  apply (sep_auto heap add: os_pop_rule)
```

```
  done
interpretation os: imp_list_pop os_list os_pop by (rule os_pop_impl)
```

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) }"
  apply (subst os_reverse_aux.simps)
  apply simp
  apply (subst os_reverse_aux.simps)
  apply simp
  done

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 (induct xs arbitrary: p q ys)
  case Nil thus ?case
    apply sep_auto
    done
next
  case (Cons x xs)
  show ?case
    apply (cases p, simp_all)
    apply (sep_auto heap add: cons_pre_rule[OF _ Cons.hyps])
    done
qed

lemma os_reverse_rule: "<os_list xs p> os_reverse p <os_list (rev xs)>"
  unfolding os_reverse_def
```

```
  apply (auto simp: os_reverse_aux_rule[where ys="[]", simplified, rule_format])
  done

lemma os_reverse_impl: "imp_list_reverse os_list os_reverse"
  apply unfold_locales
  apply (sep_auto heap add: os_reverse_rule)
  done
interpretation os: imp_list_reverse os_list os_reverse
  by (rule os_reverse_impl)
```

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) }) }"
  apply (subst os_rem.simps, simp)+
  done

lemma os_rem_rule[sep_heap_rules]:
  "<os_list xs b> os_rem x b <λr. os_list (removeAll x xs) r * true>"
proof (induct xs arbitrary: b x)
  case Nil show ?case
    apply sep_auto
    done
next
  case (Cons y xs)
  show ?case by (sep_auto heap add: Cons.hyps)
```

```
qed

lemma os_rem_rule_alt_proof:
  "<os_list xs b> os_rem x b <λr. os_list (removeAll x xs) r * true>"
proof (induct xs arbitrary: b)
  case Nil show ?case
    apply sep_auto
    done
next
  case (Cons y xs)
  show ?case
    by (sep_auto (nopre) heap add: Cons.hyps)
qed
```

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"
  apply unfold_locales
  unfolding os_it_init_def os_is_it_def[abs_def]
  apply sep_auto

  apply (case_tac it, simp)
  apply (case_tac l', simp)
  apply sep_auto
  apply (rule ent_frame_fwd[OF lseg_append])
    apply frame_inference
    apply simp
  apply (sep_auto)

  unfolding os_it_has_next_def
  apply (sep_auto elim!: neq_NilE)
```

```
  apply solve_entails
  apply (rule ent_frame_fwd[OF lseg_conc])
    apply frame_inference
    apply solve_entails
  done
interpretation os:
  imp_list_iterate os_list os_is_it os_it_init os_it_has_next os_it_next
  by (rule os_iterate_impl)
```

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
  <λr. os_list l p * ↑(r = s + sum_list l')>t"
proof (induct l' arbitrary: it s)
  case Nil thus ?case
    apply (subst os_sum'.simps)
    apply (sep_auto intro: os.quit_iteration ent_true_drop(1))
    done
next
  case (Cons x l')
  show ?case
    apply (subst os_sum'.simps)
    apply (sep_auto heap: Cons.hyps)
    done
qed

definition "os_sum p ≡ do {
  it ← os_it_init p;
  os_sum' it 0}"

lemma os_sum_rule[sep_heap_rules]:
  "<os_list l p> os_sum p <λr. os_list l p * ↑(r=sum_list l)>t"
  unfolding os_sum_def
  by sep_auto

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 ⇒ 'a node ref option ⇒ 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 = ↑(l=[])"
  by (cases l) auto

lemma [simp]:
  "cs_list l (Some p)
  = (∃_A x ls. ↑(l=x#ls) * lseg (x#ls) (Some p) (Some p))"
  apply (rule ent_iffI)
  apply (cases l)
  apply simp
  apply sep_auto
  apply (cases l)
  apply simp
  apply sep_auto
  done
```

10.2 Precision

```
lemma cs_prec:
  "precise cs_list"
  apply rule
  apply (case_tac p)
  apply clarsimp

  apply clarsimp
  apply (subgoal_tac "x=xa ∧ n=na", simp)

  apply (erule prec_frame_expl[OF lseg_prec1])
  apply frame_inference
  apply frame_inference
```

```
  apply (drule prec_frame[OF sngr_prec])
  apply frame_inference
  apply frame_inference
  apply simp
  done

lemma cs_imp_list_impl: "imp_list cs_list"
  apply unfold_locales
  apply (rule cs_prec)
  done
interpretation cs: imp_list cs_list by (rule cs_imp_list_impl)
```

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 []>"
  unfolding cs_empty_def
  by sep_auto

lemma cs_empty_impl: "imp_list_empty cs_list cs_empty"
  by unfold_locales (sep_auto heap: cs_empty_rule)
interpretation cs: imp_list_empty cs_list cs_empty by (rule cs_empty_impl)
```

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)>"
  apply (cases p)
  apply simp_all
  apply (sep_auto simp: cs_prepend.simps)
```

```
  apply (sep_auto simp: cs_prepend.simps)
  done

lemma cs_prepend_impl: "imp_list_prepend cs_list cs_prepend"
  by unfold_locales (sep_auto heap: cs_prepend_rule)
interpretation cs: imp_list_prepend cs_list cs_prepend
  by (rule cs_prepend_impl)
```

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])>"
  apply (cases p)
  apply simp_all
  apply (sep_auto simp: cs_append.simps)

  apply (sep_auto simp: cs_append.simps)
  apply (rule ent_frame_fwd)
  apply (rule_tac s=a in lseg_append)
  apply frame_inference
  apply (sep_auto)
  done

lemma cs_append_impl: "imp_list_append cs_list cs_append"
  by unfold_locales (sep_auto heap: cs_append_rule)
interpretation cs: imp_list_append cs_list cs_append
  by (rule cs_append_impl)
```

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)>"
  apply (cases p)
  apply (sep_auto simp: cs_pop.simps)

  apply (cases l)
  apply (sep_auto simp: cs_pop.simps dflt_simps: option.sel)

  apply (sep_auto
    simp: cs_pop.simps
    dflt_simps: option.sel
    eintros del: exI)

  apply (rule_tac x=aa in exI)
  apply (rule_tac x=list in exI)
  apply (rule_tac x=a in exI)
  apply clarsimp
  apply (rule exI)
  apply sep_auto
  done

lemma cs_pop_impl: "imp_list_pop cs_list cs_pop"
  apply unfold_locales
  apply (sep_auto heap: cs_pop_rule elim!: neq_NilE)
  done
interpretation cs: imp_list_pop cs_list cs_pop by (rule cs_pop_impl)
```

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)>"
  apply (cases p)
  apply (sep_auto simp: cs_rotate.simps)

  apply (cases l)
  apply simp

  apply (case_tac list)
  apply simp
  apply (sep_auto simp: cs_rotate.simps)

  apply (sep_auto simp: cs_rotate.simps)
  apply (rule ent_frame_fwd)
  apply (rule_tac s="a" in lseg_append)
  apply frame_inference
  apply sep_auto
  done

lemma cs_rotate_impl: "imp_list_rotate cs_list cs_rotate"
  apply unfold_locales
  apply (sep_auto heap: cs_rotate_rule)
  done
interpretation cs: imp_list_rotate cs_list cs_rotate by (rule cs_rotate_impl)
```

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>"
  unfolding test_def test_result_def
  apply (sep_auto)
  done

export_code test checking SML_imp
```

```
ML_val ‹
  val res = @{code test} ();
  if res = @{code test_result} then () else raise Match;
›

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 ⇀ 'v) ⇒ 'm ⇒ assn"
  assumes precise: "precise is_map"

locale imp_map_empty = imp_map +
  constrains is_map :: "('k ⇀ 'v) ⇒ 'm ⇒ 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 ⇀ 'v) ⇒ 'm ⇒ assn"
  fixes is_empty :: "'m ⇒ bool Heap"
  assumes is_empty_rule[sep_heap_rules]:
    "<is_map m p> is_empty p <λr. is_map m p * ↑(r ⟷ m=Map.empty)>ₜ"

locale imp_map_lookup = imp_map +
  constrains is_map :: "('k ⇀ 'v) ⇒ 'm ⇒ assn"
  fixes lookup :: "'k ⇒ 'm ⇒ ('v option) Heap"
  assumes lookup_rule[sep_heap_rules]:
    "<is_map m p> lookup k p <λr. is_map m p * ↑(r = m k)>ₜ"

locale imp_map_update = imp_map +
  constrains is_map :: "('k ⇀ 'v) ⇒ 'm ⇒ assn"
  fixes update :: "'k ⇒ 'v ⇒ 'm ⇒ 'm Heap"
  assumes update_rule[sep_heap_rules]:
    "<is_map m p> update k v p <is_map (m(k ↦ v))>ₜ"

locale imp_map_delete = imp_map +
  constrains is_map :: "('k ⇀ 'v) ⇒ 'm ⇒ assn"
```

```
  fixes delete :: "'k ⇒ 'm ⇒ '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 ⇀ 'v) ⇒ 'm ⇒ assn"
  fixes add :: "'m ⇒ 'm ⇒ 'm Heap"
  assumes add_rule[sep_heap_rules]:
    "<is_map m p * is_map m' p'> add p p'
     <λr. is_map m p * is_map m' p' * is_map (m ++ m') r>t"


locale imp_map_size = imp_map +
  constrains is_map :: "('k ⇀ 'v) ⇒ 'm ⇒ assn"
  fixes size :: "'m ⇒ nat Heap"
  assumes size_rule[sep_heap_rules]:
    "<is_map m p> size p <λr. is_map m p * ↑(r = card (dom m))>t"

locale imp_map_iterate = imp_map +
  constrains is_map :: "('k ⇀ 'v) ⇒ 'm ⇒ assn"
  fixes is_it :: "('k ⇀ 'v) ⇒ 'm ⇒ ('k ⇀ 'v) ⇒ 'it ⇒ assn"
  fixes it_init :: "'m ⇒ ('it) Heap"
  fixes it_has_next :: "'it ⇒ bool Heap"
  fixes it_next :: "'it ⇒ (('k×'v)×'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'≠Map.empty ⟹
    <is_it m p m' it>
      it_next it
    <λ((k,v),it'). is_it m p (m' |` (-{k})) it' * ↑(m' k = Some v)>t"
  assumes it_has_next_rule[sep_heap_rules]:
    "<is_it m p m' it> it_has_next it <λr. is_it m p m' it * ↑(r⟷m'≠Map.empty)>t"
  assumes quit_iteration:
    "is_it m p m' it ⟹A is_map m p * true"

locale imp_map_iterate' = imp_map +
  constrains is_map :: "('k ⇀ 'v) ⇒ 'm ⇒ assn"
  fixes is_it :: "('k ⇀ 'v) ⇒ 'm ⇒ ('k×'v) list ⇒ 'it ⇒ assn"
  fixes it_init :: "'m ⇒ ('it) Heap"
  fixes it_has_next :: "'it ⇒ bool Heap"
  fixes it_next :: "'it ⇒ (('k×'v)×'it) Heap"
  assumes it_init_rule[sep_heap_rules]:
    "<is_map s p> it_init p <λr. ∃A l. ↑(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
    <λ(kv',it'). is_it m p l it' * ↑(kv'=kv)>"
  assumes it_has_next_rule[sep_heap_rules]:
    "<is_it m p l it> it_has_next it <λr. is_it m p l it * ↑(r⟷l≠[])>"
```

assumes *quit_iteration:*
"is_it m p l it $\Longrightarrow_A$ *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 *(rule countable_classI[of "hs_encode"])*
fix *x y :: "('a, 'b) hashtable"*
assume *"hs_encode x = hs_encode y"*
then show *"x = y"* **by** *(cases x, cases y) auto*
qed

instance *hashtable :: (heap, heap) heap* **..**

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)* $\mapsto_a$ *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"
  apply (rule preciseI)
  unfolding is_hashtable_def
  apply (auto simp add: preciseD[OF snga_prec])
  done
```

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)>"
  unfolding is_hashtable_def
  by sep_auto

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)>"
  unfolding is_hashtable_def
  by sep_auto
```

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 [])"*
apply *(induct n)*
apply *(auto simp add: ht_hash_def)*
apply *(case_tac i)*
apply *auto*
done

lemma *ht_distinct_replicate[simp, intro!]: "ht_distinct (replicate n [])"*
apply *(induct n)*
apply *(auto simp add: ht_distinct_def)*
apply *(case_tac i)*
apply *auto*
done

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

— We can't create hash tables with a size of zero
lemma *complete_ht_new_sz: "1 < n ⟹ <emp> ht_new_sz n <is_hashtable (replicate n [])>"*
apply *(unfold ht_new_sz_def)*
apply *(simp del: replicate.simps)*
apply *(rule bind_rule)*
apply *(rule of_list_rule)*
apply *(rule return_cons_rule)*
apply *(simp add: is_hashtable_def)*
done

lemma *complete_ht_new:*
"<emp>
ht_new::('k::{heap,hashable}, 'v::heap) hashtable Heap
<is_hashtable (replicate (def_hashmap_size TYPE('k)) [])>"

```
  unfolding ht_new_def
  by (simp add: complete_ht_new_sz[OF def_hashmap_size])
```

12.4 Lookup

12.4.1 Definition

```
fun ls_lookup :: "'k ⇒ ('k × 'v) list ⇒ '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 ⇒ ('k::{heap,hashable}, 'v::heap) hashtable
⇒ 'v option Heap"
where
  "ht_lookup x ht = do {
    m ← Array.len (the_array ht);
    let i = bounded_hashcode_nat m x;
    l ← 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
     <λr. is_hashtable l ht *
        ↑(r = ls_lookup x (l!(bounded_hashcode_nat (length l) x))) >"
  apply (cases ht)
  apply (clarsimp simp: is_hashtable_def)
  apply (simp add: ht_lookup_def)
  apply (rule bind_rule)
  apply (rule length_rule)
  apply (rule norm_pre_pure_rule)
  apply (rule bind_rule)
  apply (rule nth_rule)
  apply (simp add: bounded_hashcode_nat_bounds)
  apply (rule norm_pre_pure_rule)
  apply (rule return_cons_rule)
  apply simp
  done
```

Alternative, more automatic proof

```
lemma complete_ht_lookup_alt_proof:
  "<is_hashtable l ht> ht_lookup x ht
    <λr. is_hashtable l ht *
      ↑(r = ls_lookup x (l!(bounded_hashcode_nat (length l) x)))>"
  unfolding is_hashtable_def ht_lookup_def
  apply (cases ht)
  apply (sep_auto simp: bounded_hashcode_nat_bounds)
```

done

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) ⟷ k ∈ set (map fst
l)"
  by (induct l rule: ls_update.induct) simp_all

lemma ls_update_fst_set: "set (fst (ls_update k v l)) ⊆ insert (k, v)
(set l)"
  apply (induct l rule: ls_update.induct)
  apply simp
  apply (auto simp add: Let_def)
  done

lemma ls_update_fst_map_set: "set (map fst (fst (ls_update k v l))) =
insert k (set (map fst l))"
  apply (induct l rule: ls_update.induct)
  apply simp
  apply (auto simp add: Let_def)
  done

lemma ls_update_distinct: "distinct (map fst l) ⟹ distinct (map fst
(fst (ls_update k v l)))"
proof (induct l rule: ls_update.induct)
  case 1 thus ?case by simp
next
  case (2 k v l w ls) show ?case
  proof (cases "k = l")
    case True
    with 2 show ?thesis by simp
```

```
  next
    case False
    with 2 have d: "distinct (map fst (fst (ls_update k v ls)))"
      by simp
    from 2(2) have "l ∉ set (map fst ls)" by simp
    with False have "l ∉ set (map fst (fst (ls_update k v ls)))"
      by (simp only: ls_update_fst_map_set) simp
    with d False show ?thesis by (simp add: Let_def)
  qed
qed

lemma ls_update_length: "length (fst (ls_update k v l))
  = (if (k ∈ set (map fst l)) then length l else Suc (length l))"
  by (induct l rule: ls_update.induct) (auto simp add: Let_def)

lemma ls_update_length_snd_True:
  "snd (ls_update k v l) ⟹ length (fst (ls_update k v l)) = length l"
  by (simp add: ls_update_length ls_update_snd_set)

lemma ls_update_length_snd_False:
  "¬ snd (ls_update k v l) ⟹ length (fst (ls_update k v l)) = Suc (length
l)"
  by (simp add: ls_update_length ls_update_snd_set)


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)"
  unfolding ht_hash_def abs_update_def
  apply (intro allI ballI impI)
  apply simp
  subgoal premises prems for i x
```

```
  proof (cases "i = bounded_hashcode_nat (length ls) k")
    case True
    note i = True
    show ?thesis
    proof (cases "fst x = k")
      case True
      with i show ?thesis by simp
    next
      case False
      with prems i
      have
        "x ∈ set (fst (ls_update k v
                        (ls ! bounded_hashcode_nat (length ls) k)))"
        by auto
      with
        ls_update_fst_set[
          of k v "ls ! bounded_hashcode_nat (length ls) k"]
        False
      have "x ∈ insert (k, v)
                 (set (ls ! bounded_hashcode_nat (length ls) k))"
        by auto
      with False have "x ∈ set (ls ! bounded_hashcode_nat (length ls)
k)"
        by auto
      with i prems assms[unfolded ht_hash_def] show ?thesis by simp
    qed
  next
    case False
    with prems have "x ∈ set (ls ! i)" by simp
    with prems assms[unfolded ht_hash_def] show ?thesis by simp
  qed
  done


lemma ht_distinct_update:
  assumes "ht_distinct l"
  shows "ht_distinct (abs_update k v l)"
  unfolding ht_distinct_def abs_update_def
  apply (intro allI impI)
  apply simp
  subgoal premises prems for i
  proof (cases "i = bounded_hashcode_nat (length l) k")
    case True
    with prems assms[unfolded ht_distinct_def]
    have "distinct (map fst (l ! bounded_hashcode_nat (length l) k))"
      by simp
    from ls_update_distinct[OF this, of k v] True prems
    show ?thesis by simp
  next
```

```
    case False
    with prems assms[unfolded ht_distinct_def] show ?thesis by simp
  qed
  done


lemma length_update:
  assumes "1 < length l"
  shows "1 < length (abs_update k v l)"
  using assms
  by (simp add: abs_update_def)

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 -
  have "(map length (l[i := fst (ls_update k v (l ! i))]))
    = (map length l)[i := length (fst (ls_update k v (l ! i)))]"
    by (simp add: map_update) also
  from sum_list_update[of i "map length l", simplified, OF i,
    of "length (fst (ls_update k v (l ! i)))"]
    ls_update_length_snd_True[OF snd]
  have
    "sum_list ((map length l)[i := length (fst (ls_update k v (l ! i)))])
    = sum_list (map length l)" by simp
  finally show ?thesis using assms by (simp add: ht_size_def assms)
qed

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 -
  have "(map length (l[i := fst (ls_update k v (l ! i))]))
    = (map length l)[i := length (fst (ls_update k v (l ! i)))]"
    by (simp add: map_update) also
  from sum_list_update[of i "map length l", simplified, OF i,
    of "length (fst (ls_update k v (l ! i)))"]
    ls_update_length_snd_False[OF snd]
  have
    "sum_list ((map length l)[i := length (fst (ls_update k v (l ! i)))])
    = Suc (sum_list (map length l))" by simp
  finally show ?thesis using assms by (simp add: ht_size_def assms)
qed
```

```
lemma complete_ht_upd: "<is_hashtable l ht> ht_upd k v ht
  <is_hashtable (abs_update k v l)>"
  unfolding ht_upd_def is_hashtable_def
  apply (rule norm_pre_pure_rule)
  apply (rule bind_rule)
  apply (rule length_rule)
  apply (rule norm_pre_pure_rule)
  apply (simp add: Let_def)
  apply (rule bind_rule)
  apply (rule nth_rule)
  apply (simp add: bounded_hashcode_nat_bounds)
  apply (rule norm_pre_pure_rule)
  apply (rule bind_rule)
  apply (rule upd_rule)
  apply (simp add: bounded_hashcode_nat_bounds)
  apply (rule return_cons_rule)
  apply (auto
    simp add: ht_size_update1 ht_size_update2 bounded_hashcode_nat_bounds
              is_hashtable_def ht_hash_update[unfolded abs_update_def]
              ht_distinct_update[unfolded abs_update_def] abs_update_def)
  done
```

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)>"
  unfolding ht_upd_def is_hashtable_def Let_def

  apply (sep_auto
    simp: ht_size_update1 ht_size_update2 bounded_hashcode_nat_bounds
              is_hashtable_def ht_hash_update[unfolded abs_update_def]
              ht_distinct_update[unfolded abs_update_def] abs_update_def)
  done
```

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)"
  by (induct l rule: ls_delete.induct) simp_all

lemma ls_delete_fst_set: "set (fst (ls_delete k l)) ⊆ set l"
  apply (induct l rule: ls_delete.induct)
```

```
  apply simp
  apply (auto simp add: Let_def)
  done

lemma ls_delete_fst_map_set:
  "distinct (map fst l) ⟹
  set (map fst (fst (ls_delete k l))) = (set (map fst l)) - {k}"
  apply (induct l rule: ls_delete.induct)
  apply simp
  apply (auto simp add: Let_def)
  done

lemma ls_delete_distinct: "distinct (map fst l) ⟹ distinct (map fst
(fst (ls_delete k l)))"
proof (induct l rule: ls_delete.induct)
  case 1 thus ?case by simp
next
  case (2 k l w ls) show ?case
  proof (cases "k = l")
    case True
    with 2 show ?thesis by simp
  next
    case False
    with 2 have d: "distinct (map fst (fst (ls_delete k ls)))"
      by simp
    from 2 have d2: "distinct (map fst ls)" by simp
    from 2(2) have "l ∉ set (map fst ls)" by simp
    with False 2 ls_delete_fst_map_set[OF d2, of k]
    have "l ∉ set (map fst (fst (ls_delete k ls)))"
      by simp
    with d False show ?thesis by (simp add: Let_def)
  qed
qed

lemma ls_delete_length:
  "length (fst (ls_delete k l)) = (
    if (k ∈ set (map fst l)) then
      (length l - 1)
    else
      length l)"
proof (induct l rule: ls_delete.induct)
  case 1
  then show ?case by simp
next
  case (2 k l w ls)
  then show ?case by (cases ls) (auto simp add: Let_def)
qed

lemma ls_delete_length_snd_True:
```

```
  "snd (ls_delete k l) ⟹ length (fst (ls_delete k l)) = length l - 1"
  by (simp add: ls_delete_length ls_delete_snd_set)

lemma ls_delete_length_snd_False:
  "¬ snd (ls_delete k l) ⟹ length (fst (ls_delete k l)) = length l"
  by (simp add: ls_delete_length ls_delete_snd_set)


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)
              )
      ]
  )"
  unfolding ht_hash_def
  apply (intro allI ballI impI)
  apply simp
  subgoal premises prems for i x
  proof (cases "i = bounded_hashcode_nat (length ls) k")
    case i: True
    show ?thesis
    proof (cases "fst x = k")
      case True
      with i show ?thesis by simp
    next
      case False
      with prems i
      have
        "x ∈ set (fst (ls_delete k
```

```
                    (ls ! bounded_hashcode_nat (length ls) k)))"
      by auto
    with
      ls_delete_fst_set[
        of k "ls ! bounded_hashcode_nat (length ls) k"]
      False
    have "x ∈ (set (ls ! bounded_hashcode_nat (length ls) k))" by auto
    with i prems assms[unfolded ht_hash_def] show ?thesis by simp
  qed
next
  case False
  with prems have "x ∈ set (ls ! i)" by simp
  with prems assms[unfolded ht_hash_def] show ?thesis by simp
qed
done

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))])"
  unfolding ht_distinct_def
  apply (intro allI impI)
  apply simp
  subgoal premises prems for i
  proof (cases "i = bounded_hashcode_nat (length l) k")
    case True
    with prems assms[unfolded ht_distinct_def]
    have "distinct (map fst (l ! bounded_hashcode_nat (length l) k))"
      by simp
    from ls_delete_distinct[OF this, of k] True prems
    show ?thesis by simp
  next
    case False
    with prems assms[unfolded ht_distinct_def] show ?thesis by simp
  qed
  done

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 -
  have "(map length (l[i := fst (ls_delete k (l ! i))]))
    = (map length l)[i := length (fst (ls_delete k (l ! i)))]"
    by (simp add: map_update) also
  from sum_list_update[of i "map length l", simplified, OF i,
    of "length (fst (ls_delete k (l ! i)))"]
```

```
    ls_delete_length_snd_True[OF snd] snd
  have "sum_list ((map length l)[i := length (fst (ls_delete k (l ! i)))])
    = sum_list (map length l) - 1"
    by (cases "length (l ! i)") (simp_all add: ls_delete_snd_set)
  finally show ?thesis using assms by (simp add: ht_size_def assms)
qed

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 -
  have "(map length (l[i := fst (ls_delete k (l ! i))]))
    = (map length l)[i := length (fst (ls_delete k (l ! i)))]"
    by (simp add: map_update) also
  from sum_list_update[of i "map length l", simplified, OF i,
    of "length (fst (ls_delete k (l ! i)))"]
    ls_delete_length_snd_False[OF snd]
  have "sum_list ((map length l)[i := length (fst (ls_delete k (l ! i)))])
    = sum_list (map length l)" by simp
  finally show ?thesis using assms by (simp add: ht_size_def assms)
qed


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))])>"
  unfolding ht_delete_def is_hashtable_def
  apply (rule norm_pre_pure_rule)
  apply (rule bind_rule)
  apply (rule length_rule)
  apply (rule norm_pre_pure_rule)
  apply (simp add: Let_def)
  apply (rule bind_rule)
  apply (rule nth_rule)
  apply (simp add: bounded_hashcode_nat_bounds)
  apply (rule norm_pre_pure_rule)
  apply (rule bind_rule)
  apply (rule upd_rule)
  apply (simp add: bounded_hashcode_nat_bounds)
  apply (rule return_cons_rule)
  apply (auto
    simp add: ht_size_delete1 ht_size_delete2 bounded_hashcode_nat_bounds
              is_hashtable_def ht_hash_delete ht_distinct_delete)
  using ht_size_delete1[OF _ bounded_hashcode_nat_bounds[of "length l"
k], of "the_size ht"]
  apply simp
  done
```

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))])>"
  unfolding ht_delete_def is_hashtable_def Let_def
  using ht_size_delete1[OF _ bounded_hashcode_nat_bounds[of "length l"
k],
    of "the_size ht"]
  apply (sep_auto simp:
    ht_size_delete1 ht_size_delete2 bounded_hashcode_nat_bounds
    is_hashtable_def ht_hash_delete ht_distinct_delete)
  done
```

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)"
  using assms
  apply (induct l rule: ls_insls.induct)
  apply simp
  apply (simp add: ht_hash_update)
  done

lemma ht_distinct_ls_insls:
  assumes "ht_distinct l"
  shows "ht_distinct (ls_insls ls l)"
  using assms
  apply (induct l rule: ls_insls.induct)
  apply simp
  apply (simp add: ht_distinct_update)
```

```
  done

lemma length_ls_insls:
  assumes "1 < length l"
  shows "1 < length (ls_insls ls l)"
  using assms
proof (induct l rule: ls_insls.induct)
  case 1
  then show ?case by simp
next
  case (2 k v ls l)
  from 2(1)[OF length_update[OF 2(2), of k v]] show ?case
    by simp
qed

lemma complete_ht_insls:
  "<is_hashtable ls ht> ht_insls xs ht <is_hashtable (ls_insls xs ls)>"
proof (induct xs arbitrary: ls ht)
  case Nil
  show ?case by (auto intro: return_cons_rule)
next
  case (Cons x xs)
  show ?case
  proof (cases x)
    case (Pair k v)
    then show ?thesis
      apply simp
      apply (rule bind_rule)
      apply (rule complete_ht_upd)
      apply (simp add: Cons)
      done
  qed
qed




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

Abstract version

```
fun ls_copy :: "nat ⇒ ('k::hashable × 'v) list list
  ⇒ ('k × 'v) list list ⇒ ('k × '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)"
  using assms
  apply (induct n arbitrary: l)
  apply simp
  apply (simp add: ht_hash_ls_insls)
  done

lemma ht_distinct_ls_copy:
  assumes "ht_distinct l"
  shows "ht_distinct (ls_copy n ss l)"
  using assms
  apply (induct n arbitrary: l)
  apply simp
  apply (simp add: ht_distinct_ls_insls)
  done

lemma length_ls_copy:
  assumes "1 < length l"
  shows "1 < length (ls_copy n ss l)"
  using assms
proof (induct n arbitrary: l)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from Suc(1)[OF length_ls_insls[OF Suc(2)]] show ?case by simp
qed

lemma complete_ht_copy: "n ≤ List.length ss ⟹
  <is_hashtable ss src * is_hashtable ds dst>
  ht_copy n src dst
  <λr. is_hashtable ss src * is_hashtable (ls_copy n ss ds) r>"
proof (induct n arbitrary: ds dst)
  case 0
  show ?case by (auto intro!: return_cons_rule)
next
  case (Suc n)
  then have n: "n < length ss" by simp
  then have "n ≤ length ss" by simp
  note IH = Suc(1)[OF this]
  show ?case
    apply simp
    apply (rule bind_rule)
```

```
    apply (rule frame_rule)
    apply (subgoal_tac "<is_hashtable ss src>
      Array.nth (the_array src) n
      <λr. is_hashtable ss src * ↑(r = ss ! n)>")
    apply simp
    apply (simp add: is_hashtable_def)
    apply (auto intro!: nth_rule simp add: n)[]
    apply clarsimp
    apply (rule bind_rule)
    apply (rule frame_rule_left)
    apply (rule complete_ht_insls)
    apply (simp add: IH)
    done
qed
```

Alternative, more automatic proof

```
lemma complete_ht_copy_alt_proof: "n ≤ List.length ss ⟹
  <is_hashtable ss src * is_hashtable ds dst>
  ht_copy n src dst
  <λr. is_hashtable ss src * is_hashtable (ls_copy n ss ds) r>"
proof (induct n arbitrary: ds dst)
  case 0
  show ?case by (sep_auto)
next
  case (Suc n)
  then have N_LESS: "n < length ss" by simp
  then have N_LE: "n ≤ length ss" by simp
  note IH = Suc(1)[OF this]
  show ?case
    by (sep_auto simp: N_LESS N_LE heap: complete_ht_insls IH)
qed

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

Operation on Abstraction

```
definition ls_rehash :: "('k::hashable × 'v) list list ⇒ ('k × '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)"
```

```
  by (simp add: ht_hash_ls_copy ls_rehash_def)

lemma ht_distinct_ls_rehash: "ht_distinct (ls_rehash l)"
  by (simp add: ht_distinct_ls_copy ls_rehash_def)

lemma length_ls_rehash:
  assumes "1 < length l"
  shows "1 < length (ls_rehash l)"
proof -
  from assms have "1 < length (replicate (2 * length l) [])" by simp
  from length_ls_copy[OF this, of "length l" l] show ?thesis
    by (simp add: ls_rehash_def)
qed

lemma ht_imp_len: "is_hashtable l ht ⟹_A is_hashtable l ht * ↑(length
l > 0)"
  unfolding is_hashtable_def
  by sep_auto

lemma complete_ht_rehash:
  "<is_hashtable l ht> ht_rehash ht
  <λr. is_hashtable l ht * is_hashtable (ls_rehash l) r>"
  apply (rule cons_pre_rule[OF ht_imp_len])
  unfolding ht_rehash_def
  apply (sep_auto heap: complete_ht_new_sz)
  apply (cases l; simp)
  apply (sep_auto heap: complete_ht_copy simp: ls_rehash_def)
  done

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

definition ht_update
  :: "'k::{heap,hashable} ⇒ 'v::heap ⇒ ('k, 'v) hashtable
  ⇒ ('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)>"
  unfolding ht_update_def
  apply (sep_auto simp: is_hashtable_def)
  apply (rule cons_pre_rule[where P' = "is_hashtable l ht"])
  apply (simp add: is_hashtable_def)
  apply (simp add: complete_ht_upd)
  done

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>"
  unfolding ht_update_def
  by (sep_auto heap: complete_ht_rehash complete_ht_upd)
```

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)>"
  unfolding ht_to_list_def is_hashtable_def
  by sep_auto

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)"
  apply (induct l rule: ls_update.induct)
  by (auto simp add: ext Let_def)

lemma map_of_concat:
  "k ∈ dom (map_of(concat l))
```

```
  ⟹ ∃i. k ∈ dom (map_of(l!i)) ∧ i < length l"
  apply (induct l)
  apply simp
  apply auto
  apply (rule_tac x = 0 in exI)
  apply auto
  by (metis Suc_mono domI nth_Cons_Suc)

lemma map_of_concat':
  "k ∈ dom (map_of(l!i)) ∧ i < length l ⟹ k ∈ dom (map_of(concat l))"
  apply (induct l arbitrary: i)
  apply simp
  apply auto
  apply (case_tac i)
  apply auto
  done

lemma map_of_concat''':
  assumes "∃i. k ∈ dom (map_of(l!i)) ∧ i < length l"
  shows "k ∈ dom (map_of(concat l))"
proof -
  from assms obtain i where "k ∈ dom (map_of (l ! i)) ∧ i < length l"
by blast
  from map_of_concat'[OF this] show ?thesis .
qed

lemma map_of_concat'':
  "(k ∈ dom (map_of(concat l)))
  ⟷ (∃i. k ∈ dom (map_of(l!i)) ∧ i < length l)"
  apply rule
  using map_of_concat[of k l]
  apply simp
  using map_of_concat'[of k l]
  by blast

lemma abs_update_length: "length (abs_update k v l) = length l"
  by (simp add: abs_update_def)

lemma ls_update_map_of_eq:
  "map_of (fst (ls_update k v ls)) k = Some v"
  apply (induct ls rule: ls_update.induct)
  by (simp_all add: Let_def)

lemma ls_update_map_of_neq:
  "x ≠ k ⟹ map_of (fst (ls_update k v ls)) x = map_of ls x"
  apply (induct ls rule: ls_update.induct)
  by (auto simp add: Let_def)
```

13.2 Main Definitions and Lemmas

definition *is_hashmap'*
```
  :: "('k, 'v) map
    ⇒ ('k × 'v) list list
    ⇒ ('k::{heap,hashable}, 'v::heap) hashtable
    ⇒ assn"
  where
  "is_hashmap' m l ht = is_hashtable l ht * ↑ (map_of (concat l) = m)"
```

definition *is_hashmap*
```
  :: "('k, 'v) map ⇒ ('k::{heap,hashable}, 'v::heap) hashtable ⇒ assn"

  where
  "is_hashmap m ht = (∃A l. 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'"
  unfolding is_hashmap'_def
  apply (auto simp add: preciseD[OF is_hashtable_prec])
  apply (subgoal_tac "l = l'")
  by (auto simp add: preciseD[OF is_hashtable_prec])
```

lemma *is_hashmap_prec: "precise is_hashmap"*
```
  unfolding is_hashmap_def[abs_def]
  apply rule
  by (auto simp add: is_hashmap'_prec)
```

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)) [])>"
  apply (rule cons_post_rule)
  using complete_ht_new
  apply simp
  apply (simp add: is_hashmap'_def)
  done
```

lemma *hm_new_rule:*
```
  "<emp> hm_new <is_hashmap Map.empty>"
  apply (rule cons_post_rule)
  using complete_ht_new
  apply simp
  apply (simp add: is_hashmap_def is_hashmap'_def)
  apply sep_auto
  done
```

```
lemma ht_hash_distinct:
  "ht_hash l
  ⟹ ∀i j . i≠j ∧ i < length l ∧ j < length l
    ⟶ set (l!i) ∩ set (l!j) = {}"
  apply (auto simp add: ht_hash_def)
  apply metis
  done

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 -
  from map_of_concat[OF assms(2)] obtain i
    where i: "k ∈ dom (map_of (l ! i)) ∧ i < length l"
    by blast
  thm ht_hash_def
  hence "∃v. (k,v)∈set(l!i)" by (auto dest: map_of_SomeD)
  from this obtain v where v: "(k,v)∈set(l!i)" by blast
  from assms(1)[unfolded ht_hash_def] i v bounded_hashcode_nat_bounds

  have "bounded_hashcode_nat (length l) k = i"
    by (metis fst_conv)
  with i show ?thesis by simp
qed

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))"
  using map_of_concat'[of k l "bounded_hashcode_nat (length l) k"]
    assms(2,3) bounded_hashcode_nat_bounds[of "length l" k]
  by simp


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)))"
  apply rule
  using ht_hash_in_dom_in_dom_bounded_hashcode_nat[OF assms(1)]
    ht_hash_in_dom_bounded_hashcode_nat_in_dom[OF assms]
  by simp_all


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"
  using assms
  using bounded_hashcode_nat_bounds
  by (auto simp add: ht_hash_def ht_distinct_def dom_map_of_conv_image_fst)

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))"
  using assms
  using bounded_hashcode_nat_bounds
  by (auto simp add: ht_hash_def ht_distinct_def dom_map_of_conv_image_fst)

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 -
  from assms(4) have ex: "∃v. (k,v) ∈ set (concat l)"
    by (auto dest!: map_of_SomeD)
  have "v = w" if kv: "(k,v) ∈ set (concat l)" and kw: "(k,w) ∈ set (concat
l)" for v w
  proof -
    from ht_hash_in_dom_in_dom_bounded_hashcode_nat[OF assms(1,4)]
    have a: "k ∈ dom (map_of (l ! bounded_hashcode_nat (length l) k))"
.
    have "k ∉ dom(map_of(l!i))"
      if "i < length l" and "i ≠ bounded_hashcode_nat (length l) k" for
i
    proof -
      from ht_hash_in_bounded_hashcode_nat_not_i_not_in_dom_i[OF assms(1,3)
that]
      show ?thesis .
    qed
    have v: "(k,v) ∈ set (l ! bounded_hashcode_nat (length l) k)"
    proof -
      from kv have "∃i. i < length l ∧ (k, v) ∈ set (l!i)"
        by auto (metis in_set_conv_nth)
      from this obtain i where i: "i < length l ∧ (k, v) ∈ set (l!i)"
        by blast
      hence "k ∈ dom (map_of (l!i))"
        by (metis (no_types) prod.exhaust a assms(1) fst_conv ht_hash_def)
```

```
      from i ht_hash_in_dom_i_bounded_hashcode_nat_i[OF assms(1,3) _ this]

      have "i = bounded_hashcode_nat (length l) k" by simp
      with i show ?thesis by simp
    qed
    have w: "(k,w) ∈ set (l ! bounded_hashcode_nat (length l) k)"
    proof -
      from kw have "∃i. i < length l ∧ (k, w) ∈ set (l!i)"
        by auto (metis in_set_conv_nth)
      from this obtain i where i: "i < length l ∧ (k, w) ∈ set (l!i)"
        by blast
      hence "k ∈ dom (map_of (l!i))"
        by (metis (no_types) prod.exhaust a assms(1) fst_conv ht_hash_def)

      from i ht_hash_in_dom_i_bounded_hashcode_nat_i[OF assms(1,3) _ this]

      have "i = bounded_hashcode_nat (length l) k" by simp
      with i show ?thesis by simp
    qed
    from assms(2,3) have
      "distinct (map fst (l ! bounded_hashcode_nat (length l) k))"
      by (simp add: ht_distinct_def bounded_hashcode_nat_bounds)
    from Map.map_of_is_SomeI[OF this v] Map.map_of_is_SomeI[OF this w]
    show "v = w" by simp
  qed
  with ex show ?thesis by blast
qed


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 (cases "k ∈ dom (map_of(concat l))")
  case False
  hence a: "map_of (concat l) k = None" by auto
  from ht_hash_in_dom_in_dom_bounded_hashcode_nat_eq[OF assms(1,3)] False

  have "k ∉ dom (map_of (l ! bounded_hashcode_nat (length l) k))" by
simp
  hence "map_of(l!bounded_hashcode_nat (length l) k) k = None" by auto
  with a show ?thesis by simp
next
  case True
  from True obtain y where y: "map_of (concat l) k = Some y" by auto
  hence a: "(k,y) ∈ set (concat l)" by (metis map_of_SomeD)
```

```
  from ht_hash_in_dom_in_dom_bounded_hashcode_nat_eq[OF assms(1,3)] True

  have "k ∈ dom (map_of (l ! bounded_hashcode_nat (length l) k))" by
simp
  from this obtain z where
    z: "map_of(l!bounded_hashcode_nat (length l) k) k = Some z" by auto
  hence "(k,z) ∈ set (l ! bounded_hashcode_nat (length l) k)"
    by (metis map_of_SomeD)
  with bounded_hashcode_nat_bounds[OF assms(3), of k]
  have b: "(k,z) ∈ set (concat l)" by auto
  from ht_hash_ht_distinct_in_dom_unique_value[OF assms True] a b
  have "y = z" by auto
  with y z show ?thesis by simp
qed

lemma ls_lookup_map_of_pre:
  "distinct (map fst l) ⟹ ls_lookup k l = map_of l k"
  apply (induct l)
  apply simp
  apply (case_tac a)
  by simp

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 -
  from assms(2,3)
  have "distinct (map fst (l ! bounded_hashcode_nat (length l) k))"
    by (simp add: ht_distinct_def bounded_hashcode_nat_bounds)
  from ls_lookup_map_of_pre[OF this]
  have "ls_lookup k (l ! bounded_hashcode_nat (length l) k)
    = map_of (l ! bounded_hashcode_nat (length l) k) k" .
  also from ht_hash_ht_distinct_map_of[OF assms]
  have "map_of (l ! bounded_hashcode_nat (length l) k) k
    = map_of (concat l) k"
    by simp
  finally show ?thesis .
qed

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)>"
  unfolding is_hashmap'_def
  apply sep_auto
```

```
  apply (rule cons_post_rule)
  using complete_ht_lookup[of l ht k]
  apply simp
  apply sep_auto
  by (simp add: ls_lookup_map_of is_hashtable_def)

lemma hm_lookup_rule:
  "<is_hashmap m ht> hm_lookup k ht
    <λr. is_hashmap m ht *
      ↑(r = m k)>"
  unfolding is_hashmap_def
  apply sep_auto
  apply (rule cons_post_rule[OF hm_lookup_rule'])
  by sep_auto

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 -
  from ht_hash_ht_distinct_map_of[
    OF ht_hash_update[OF assms(1)]
       ht_distinct_update[OF assms(2)]
       length_update[OF assms(3)],
    of k v k]
  have "map_of (concat (abs_update k v l)) k
    = map_of ((abs_update k v l) ! bounded_hashcode_nat (length l) k)
k"
    by (simp add: abs_update_length)
  also have
    "... = map_of (fst (ls_update k v
                        (l ! bounded_hashcode_nat (length l) k))) k"
    by (simp add: abs_update_def bounded_hashcode_nat_bounds[OF assms(3)])
  also have "... = Some v" by (simp add: ls_update_map_of_eq)
  finally show ?thesis .
qed

lemma abs_update_map_of_hceq:
  assumes "ht_hash l"
  assumes "ht_distinct l"
  assumes "1 < length l"
  assumes "x ≠ 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 -
  from ht_hash_ht_distinct_map_of[
    OF ht_hash_update[OF assms(1)]
```

```
      ht_distinct_update[OF assms(2)]
      length_update[OF assms(3)],
    of k v x]
  have "map_of (concat (abs_update k v l)) x
    = map_of ((abs_update k v l) ! bounded_hashcode_nat (length l) x)
x"
    by (simp add: abs_update_length)
  also from assms(5) have
    "... = map_of (fst (ls_update k v
                        (l ! bounded_hashcode_nat (length l) k))) x"
    by (simp add: abs_update_def bounded_hashcode_nat_bounds[OF assms(3)])
  also have
    "... = map_of (l ! bounded_hashcode_nat (length l) x) x"
    by (simp add: ls_update_map_of_neq[OF assms(4)] assms(5))
  also from ht_hash_ht_distinct_map_of[OF assms(1-3)] have
    "... = map_of (concat l) x"
    by simp
  finally show ?thesis .
qed

lemma abs_update_map_of_hcneq:
  assumes "ht_hash l"
  assumes "ht_distinct l"
  assumes "1 < length l"
  assumes "x ≠ 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 -
  from ht_hash_ht_distinct_map_of[
    OF ht_hash_update[OF assms(1)]
       ht_distinct_update[OF assms(2)]
       length_update[OF assms(3)],
    of k v x]
  have "map_of (concat (abs_update k v l)) x
    = map_of ((abs_update k v l) ! bounded_hashcode_nat (length l) x)
x"
    by (simp add: abs_update_length)
  also from assms(5)
  have "... = map_of (l ! bounded_hashcode_nat (length l) x) x"
    by (simp add: abs_update_def bounded_hashcode_nat_bounds[OF assms(3)])
  also from ht_hash_ht_distinct_map_of[OF assms(1-3)]
  have "... = map_of (concat l) x"
    by simp
  finally show ?thesis .
qed


lemma abs_update_map_of''':
```

```
  assumes "ht_hash l"
  assumes "ht_distinct l"
  assumes "1 < length l"
  assumes "x ≠ k"
  shows "map_of (concat (abs_update k v l)) x = map_of (concat l) x"
  apply (cases
    "bounded_hashcode_nat (length l) x = bounded_hashcode_nat (length
l) k")
  by (auto simp add: abs_update_map_of_hceq[OF assms]
    abs_update_map_of_hcneq[OF assms])

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 ↦ v)) x"
  apply (cases "x = k")
  apply (simp add: abs_update_map_of''[OF assms])
  by (simp add: abs_update_map_of'''[OF assms])

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) "
  apply (rule ext)
  by (simp add: abs_update_map_of'[OF assms])


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"
  using assms
  apply (induct xs arbitrary: ld)
  apply simp
  apply (case_tac a)
  apply (simp only: ls_insls.simps)
  subgoal premises prems
  proof -
    from prems(5) prems(1)[OF ht_hash_update[OF prems(2)]
      ht_distinct_update[OF prems(3)]
      length_update[OF prems(4)]]
      abs_update_map_of[OF prems(2-4)]
```

```
    show ?thesis
      apply simp
      apply (rule map_add_upd_left)
      apply (metis dom_map_of_conv_image_fst)
      done
  qed
  done

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 -
  from assms(2,6) have "distinct (map fst (ls ! n))"
    by (simp add: ht_distinct_def)
  from ls_insls_map_of[OF assms(3-5) this] assms(6) show ?thesis
    by (simp add: List.take_Suc_conv_app_nth)
qed

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))"
  using assms
  apply (induct n arbitrary: ld)
  apply simp
  subgoal premises prems for n ld
  proof -
    note a = ht_hash_ls_insls[OF prems(4), of "ls ! n"]
    note b = ht_distinct_ls_insls[OF prems(5), of "ls ! n"]
    note c = length_ls_insls[OF prems(6), of "ls ! n"]
    from prems have "n < length ls" by simp
    with
      ls_insls_map_of'[OF prems(2-6) this]
      prems(1)[OF assms(1,2) a b c]
    show ?thesis by simp
  qed
  done
```

```
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)"
  using assms(3) ls_copy_map_of[OF assms(1,2)
    ht_hash_replicate ht_distinct_replicate]
  by (simp add: ls_rehash_def)


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 -
  note a = ht_hash_ls_rehash[of l]
  note b = ht_distinct_ls_rehash[of l]
  note c = length_ls_rehash[OF assms(3)]
  from abs_update_map_of[OF a b c] ls_rehash_map_of[OF assms]
  show ?thesis by simp
qed

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 (cases "length l * load_factor ≤ the_size ht * 100")
  case True
  show ?thesis
    unfolding is_hashmap'_def
    apply sep_auto
    apply (rule cons_post_rule[OF complete_ht_update_rehash[OF True]])
    unfolding is_hashmap_def is_hashmap'_def
    apply sep_auto
    apply (simp add: abs_update_rehash_map_of is_hashtable_def)
    done
next
  case False
  show ?thesis
    unfolding is_hashmap'_def is_hashtable_def
    apply sep_auto
    apply (rule cons_post_rule)
    using complete_ht_update_normal[OF False, simplified is_hashtable_def,
```

```
    simplified, of k v]
  apply auto
  unfolding is_hashmap_def is_hashmap'_def
  apply sep_auto
  by (simp add: abs_update_map_of is_hashtable_def)
qed

lemma hm_update_rule:
  "<is_hashmap m ht>
    hm_update k v ht
  <λr. is_hashmap (m(k ↦ v)) r * true>"
  unfolding is_hashmap_def[of m]
  by (sep_auto heap add: hm_update_rule')

lemma ls_delete_map_of:
  assumes "distinct (map fst l)"
  shows "map_of (fst (ls_delete k l)) x = ((map_of l) |` (- {k})) x"
  using assms
  apply (induct l rule: ls_delete.induct)
  apply simp
  apply (auto simp add: map_of_eq_None_iff Let_def)
  by (metis ComplD ComplI Compl_insert option.set(2)
    insertE insertI2 map_upd_eq_restrict restrict_map_def)

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 -
  from assms(2) bounded_hashcode_nat_bounds[OF assms(5)] have
    distinct: "distinct (map fst (l ! bounded_hashcode_nat (length l)
k))"
    by (auto simp add: ht_distinct_def)
  note id1 = ht_hash_ht_distinct_map_of[OF assms(3,4), simplified,
    OF assms(5)[simplified], of x]
  note id2 = ht_hash_ht_distinct_map_of[OF assms(1,2,5), of x]
  show ?thesis
  proof (cases
      "bounded_hashcode_nat (length l) x = bounded_hashcode_nat (length
l) k")
    case True
    with id1
```

```
    have "map_of (concat (l[bounded_hashcode_nat (length l) k
      := fst (ls_delete k (l ! bounded_hashcode_nat (length l) k))]))
x
      =
      map_of (l[bounded_hashcode_nat (length l) k
        := fst (ls_delete k (l ! bounded_hashcode_nat (length l) k))]

      ! bounded_hashcode_nat (length l) k) x"
      by simp
    also have
      "... = map_of (fst (ls_delete k
                          (l ! bounded_hashcode_nat (length l) k))) x"
      by (simp add: bounded_hashcode_nat_bounds[OF assms(5)])
    also from ls_delete_map_of[OF distinct] have
      "... = (map_of (l ! bounded_hashcode_nat (length l) k) |` (- {k}))
x"
      by simp
    finally show ?thesis
      by (cases "x = k") (simp_all add: id2 True)
  next
    case False
    with bounded_hashcode_nat_bounds[OF assms(5)] id1 id2[symmetric]
    show ?thesis
      by (cases "x = k") simp_all
  qed
qed

abbreviation "hm_delete ≡ ht_delete"
lemma hm_delete_rule':
  "<is_hashmap' m l ht> hm_delete k ht <is_hashmap (m |` (-{k}))>"
  apply (simp only: is_hashmap'_def[of m] is_hashtable_def)
  apply sep_auto
  apply (rule cons_post_rule)
  using complete_ht_delete[simplified is_hashtable_def]
  apply sep_auto
  apply (simp add: is_hashmap_def is_hashmap'_def)
  apply (sep_auto)
  apply (simp add: is_hashtable_def)
  apply (sep_auto)
  by (auto simp add: update_ls_delete_map_of)

lemma hm_delete_rule:
  "<is_hashmap m ht> hm_delete k ht <is_hashmap (m |` (-{k}))>"
  unfolding is_hashmap_def[of m]
  by (sep_auto heap add: hm_delete_rule')

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)>"
  unfolding hm_isEmpty_def
  unfolding is_hashmap_def is_hashmap'_def is_hashtable_def ht_size_def
  apply (cases ht, simp)
  apply sep_auto
  done

lemma hm_isEmpty_rule:
  "<is_hashmap m ht> hm_isEmpty ht <λr. is_hashmap m ht * ↑(r ⟷ m=Map.empty)>"
  unfolding is_hashmap_def
  apply (sep_auto heap: hm_isEmpty_rule')
  done

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))"
  using assms
  apply (induct l)
  apply simp
  apply simp
  apply (case_tac a)
  apply (auto intro!: fst_conv map_of_SomeD)
  apply (subgoal_tac "aa ∉ dom (map_of l)")
  apply simp
  by (metis dom_map_of_conv_image_fst)

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)) = {}"
  using assms
  unfolding ht_hash_def
  apply auto
  by (metis fst_conv map_of_SomeD)


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)))
= {}"
```

```
  apply auto
  subgoal premises prems for x y z
  proof -
    from prems(2) have "x ∈ dom (map_of (concat (drop (Suc i) l)))"
      by auto
    hence "∃j. j < length (drop (Suc i) l)
      ∧ x ∈ dom (map_of ((drop (Suc i) l)!j))"
      by (metis Hash_Map.map_of_concat
        ‹x ∈ dom (map_of (concat (drop (Suc i) l)))› length_drop)
    from this obtain j where
      j: "j < length (drop (Suc i) l)
        ∧ x ∈ dom (map_of ((drop (Suc i) l)!j))"
      by blast
    hence length: "(Suc i + j) < length l" by auto
    from j have neq: "i ≠ (Suc i + j)" by simp
    from j have in_dom: "x ∈ dom (map_of (l!(Suc i + j)))" by auto
    from prems(1) have in_dom2: "x ∈ dom (map_of (l ! i))" by auto
    from ht_hash_dom_map_of_disj[OF assms length neq] in_dom in_dom2
    show ?thesis by auto
  qed
  done

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)))"
  using assms
proof -
  from ht_hash_dom_map_of_disj_drop[OF assms(1)]
  have "∀i. i < length l
    ⟶ dom (map_of (l ! i)) ∩ dom (map_of (concat (drop (Suc i) l)))
      = {}"
    by auto
  with assms(2) show ?thesis
  proof (induct l)
    case Nil
    thus ?case by simp
  next
    case (Cons l ls)
    from Cons(2) have a: "ht_distinct ls" by (auto simp add: ht_distinct_def)
    from Cons(3) have b: "∀i < length ls. dom (map_of (ls ! i))
      ∩ dom (map_of (concat (drop (Suc i) ls))) = {}"
      apply simp
      apply (rule allI)
      apply (rule_tac x="Suc i" and P="(λi. i<Suc (length ls) ⟶
            dom (map_of ((l # ls) ! i)) ∩ dom (map_of (concat (drop i
ls))) =
            {})" in allE)
      by simp_all
```

```
    from Cons(2) have "distinct (map fst l)" by (auto simp add: ht_distinct_def)
    note l = length_card_dom_map_of[OF this]
    from Cons(3) have c: "dom (map_of l) ∩ dom (map_of (concat ls)) =
{}"
      apply (rule_tac x="0" and P="(λi. i<Suc (length ls) ⟶
             dom (map_of ((l # ls) ! i))
               ∩ dom (map_of (concat (drop i ls)))
             = {})" in allE)
      by simp_all
    from Cons(1)[OF a b] l c show ?case by (simp add: card_Un_disjoint)
  qed
qed


lemma hm_size_rule':
  "<is_hashmap' m l ht>
  hm_size ht
  <λr. is_hashmap' m l ht * ↑(r = card (dom m))>"
  unfolding hm_size_def is_hashmap_def is_hashmap'_def is_hashtable_def
  apply sep_auto
  apply (cases ht)
  apply (simp add: ht_size_def)
  apply sep_auto
  by (simp add: sum_list_length_card_dom_map_of_concat)

lemma hm_size_rule:
  "<is_hashmap m ht>
    hm_size ht
  <λr. is_hashmap m ht * ↑(r = card (dom m))>"
  unfolding is_hashmap_def
  by (sep_auto heap: hm_size_rule')
```

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)"
  apply (induct n arbitrary: l)
```

```
  apply (case_tac l)
  apply auto [2]
  apply (case_tac l)
  apply auto [2]
  done

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"
  using assms
proof (induct ls arbitrary: i thesis)
  case Nil thus ?case by auto
next
  case (Cons l ls)
  show ?case proof (cases)
    assume L: "i < length l"
    hence "concat (l#ls) ! i = (l#ls)!0!i" by (auto simp: nth_append)
    thus ?thesis
      apply (rule_tac Cons.prems(1)[of 0 i])
      apply (simp_all add: L)
      done
  next
    assume L: "¬(i < length l)"
    hence 1: "concat (l#ls)!i = concat ls ! (i - length l)"
      by (auto simp: nth_append)
    obtain j k where
      "j < length ls" and "k < length (ls!j)"
      and "concat ls ! (i - length l) = ls!j!k"
      and "i - length l = length (concat (take j ls)) + k"
      apply (rule Cons.hyps[of "i - length l"])
      using Cons.prems L
      by auto
    thus ?case using L
      apply (rule_tac Cons.prems(1)[of "Suc j" k])
      apply (auto simp: nth_append)
      done
  qed
qed

lemma is_hashmap'_distinct:
  "is_hashtable l ht
    ⟹_A is_hashtable l ht * ↑(distinct (map fst (concat l)))"
  apply (simp add: distinct_conv_nth)
proof (intro allI impI, elim exE)
  fix i j a b
```

```
  assume 1: "i < length (concat l)"
  assume 2: "j < length (concat l)"
  assume 3: "i≠j"

  assume HM: "(a,b) ⊨ is_hashtable l ht"

  from 1 obtain ji ki where
    IFMT: "i = length (concat (take ji l)) + ki"
    and JI_LEN: "ji < length l"
    and KI_LEN: "ki < length (l!ji)"
    and [simp]: "concat l ! i = l!ji!ki"
    by (blast elim: nth_concat_splitE)

  from 2 obtain jj kj where
    JFMT: "j = length (concat (take jj l)) + kj"
    and JJ_LEN: "jj < length l"
    and KJ_LEN: "kj < length (l!jj)"
    and [simp]: "concat l ! j = l!jj!kj"
    by (blast elim: nth_concat_splitE)

  show "fst (concat l ! i) ≠ fst (concat l ! j)"
  proof cases
    assume [simp]: "ji=jj"
    with IFMT JFMT ‹i≠j› have "ki≠kj" by auto
    moreover from HM JJ_LEN have "distinct (map fst (l!jj))"
      unfolding is_hashmap'_def is_hashtable_def ht_distinct_def
      by auto
    ultimately show ?thesis using KI_LEN KJ_LEN
      by (simp add: distinct_conv_nth)
  next
    assume NE: "ji≠jj"
    from HM have
      "∀x∈set (l!ji). bounded_hashcode_nat (length l) (fst x) = ji"
      "∀x∈set (l!jj). bounded_hashcode_nat (length l) (fst x) = jj"
      unfolding is_hashmap'_def is_hashtable_def ht_hash_def
      using JI_LEN JJ_LEN
      by auto
    with KI_LEN KJ_LEN NE show ?thesis
      apply (auto) by (metis nth_mem)
  qed
qed

lemma take_set: "set (take n l) = { l!i | i. i<n ∧ i<length l }"
  apply (auto simp add: set_conv_nth)
  apply (rule_tac x=i in exI)
  apply auto
  done

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 -
  have "take (Suc n) l = take (Suc x + (n - x)) l"
    by simp
  also have "... = take (Suc x) l @ take (n - x) (drop (Suc x) l)"
    by (simp only: take_add)
  finally  have
    "concat (take (Suc x) l) =
      concat (take (Suc x) l) @ concat (take (n - x) (drop (Suc x) l))"
    using A by simp
  hence 1: "∀l∈set (take (n - x) (drop (Suc x) l)). l=[]" by simp
  show ?thesis
  proof safe
    fix i
    assume "x<i" and "i≤n"
    hence "l!i ∈ set (take (n - x) (drop (Suc x) l))"
      using L[simp del]
      apply (auto simp: take_set)
      apply (rule_tac x="i - Suc x" in exI)
      apply auto
      done
    with 1 show "l!i=[]" by blast
  qed
qed

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]"
  by (cases l, auto)+


lemma concat_take_Suc_app_nth:
  assumes "x < length l"
  shows "concat (take (Suc x) l) = concat (take x l) @ l ! x"
  using assms
  by (auto simp: take_Suc_conv_app_nth)

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"
  using assms
  unfolding is_hashtable_def ht_hash_def
  apply (cases "l!i!j")
  apply (force simp: set_conv_nth)
```

```
  done

lemma distinct_imp_distinct_take:
  "distinct (map fst (concat l))
  ⟹ distinct (map fst (concat (take x l)))"
  apply (subst (asm) append_take_drop_id[of x l,symmetric])
  apply (simp del: append_take_drop_id)
  done

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 (induct i)
  case 0 thus ?case by sep_auto
next
  case (Suc n)
  show ?case using Suc.prems
    by (sep_auto
      heap add: Suc.hyps
      simp: concat_take_Suc_empty
      split: list.split)
qed

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') )>"
  unfolding hm_it_next_def hm_is_it'_def is_hashmap'_def
  using [[hypsubst_thin = true]]
  apply (sep_auto (plain)
    split: nat.split list.split
    heap: hm_it_adjust_rule
    simp: take_Suc0)
  apply (simp split: prod.split nat.split list.split)
  apply (intro allI impI conjI)
  apply auto []
  apply auto []
  apply sep_auto []

  apply (sep_auto (plain)
    heap: hm_it_adjust_rule)
```

```
  apply auto []
  apply sep_auto

  apply (cases l, auto) []
  apply (metis SUP_upper fst_image_mp image_mono set_concat)

  apply (drule skip_empty_aux, simp_all) []
  defer
  apply (auto simp: concat_take_Suc_app_nth) []

  apply auto []
  apply sep_auto

  apply (auto simp: butlast_append) []
  apply (auto simp: butlast_append) []

  apply sep_auto
  apply (auto simp: butlast_append) []
  apply (auto simp: butlast_append) []
  by (metis Ex_list_of_length Suc_leD concat_take_Suc_app_nth le_neq_implies_less
le_trans nat.inject not_less_eq_eq)
```

13.3.3 Main Lemmas

```
lemma hm_it_next_rule: "m'≠Map.empty ⟹
    <hm_is_it m ht m' it>
      hm_it_next it
    <λ((k,v),it'). hm_is_it m ht (m' |` (-{k})) it' * ↑(m' k = Some v)>"
proof -
  { fix ys a
    have aux3: "
      ⟦distinct (map fst ys); a ∉ fst ` set ys⟧ ⟹ map_of ys a = None"
      by (induct ys) auto
  } note aux3 = this

  assume "m'≠Map.empty"
  thus ?thesis
    unfolding hm_is_it_def
    apply (sep_auto heap: hm_it_next_rule')
    apply (case_tac l' rule: rev_cases,
      auto simp: restrict_map_def aux3 intro!: ext) []
    apply (case_tac l' rule: rev_cases, auto)
    done
qed

lemma hm_it_init_rule:
  fixes ht :: "('k::{heap,hashable},'v::heap) hashtable"
  shows "<is_hashmap m ht> hm_it_init ht <hm_is_it m ht m>_t"
  unfolding hm_it_init_def is_hashmap_def is_hashmap'_def
```

```
    hm_is_it_def hm_is_it'_def
  apply (sep_auto simp del: map_of_append heap add: hm_it_adjust_rule)
  apply (case_tac l, auto) []
  apply (sep_auto simp del: concat_eq_Nil_conv map_of_append)
  apply (auto simp: distinct_imp_distinct_take
    dest: ent_fwd[OF _ is_hashmap'_distinct]) []

  apply (drule sym)
  apply (auto
    simp: is_hashtable_def ht_distinct_def rev_map[symmetric]) []

  apply (auto simp: set_conv_nth) []
  apply (hypsubst_thin)
  apply (drule_tac j=ia in hm_hashcode_eq, simp_all) []
  apply (drule_tac j=ib in hm_hashcode_eq, simp_all) []

  apply (auto
    simp: is_hashmap'_def is_hashtable_def ht_distinct_def) []

  apply (clarsimp)
  apply (drule ent_fwd[OF _ is_hashmap'_distinct])
  apply clarsimp
  apply (subst concat_take_Suc_app_nth)
  apply (case_tac l,auto) []
  apply (simp)
  apply (hypsubst_thin)
  apply (subst (asm) (2) concat_take_Suc_app_nth)
  apply (case_tac l,auto) []
  apply (subst map_of_rev_distinct)
  apply auto
  done

lemma hm_it_has_next_rule:
  "<hm_is_it m ht m' it> hm_it_has_next it
    <λr. hm_is_it m ht m' it * ↑(r⟷m'≠Map.empty)>"
  unfolding is_hashmap'_def hm_is_it_def hm_is_it'_def hm_it_has_next_def
  by (sep_auto split: nat.split list.split)

lemma hm_it_finish: "hm_is_it m p m' it ⟹_A is_hashmap m p"
  unfolding hm_is_it_def hm_is_it'_def is_hashmap_def is_hashmap'_def
  by sep_auto

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"
  apply unfold_locales
  apply (rule is_hashmap_prec)
  done
interpretation hm: imp_map is_hashmap by (rule hm_map_impl)

lemma hm_empty_impl: "imp_map_empty is_hashmap hm_new"
  apply unfold_locales
  apply (sep_auto heap: hm_new_rule)
  done
interpretation hm: imp_map_empty is_hashmap hm_new by (rule hm_empty_impl)

lemma hm_lookup_impl: "imp_map_lookup is_hashmap hm_lookup"
  apply unfold_locales
  apply (sep_auto heap: hm_lookup_rule)
  done
interpretation hm: imp_map_lookup is_hashmap hm_lookup by (rule hm_lookup_impl)

lemma hm_update_impl: "imp_map_update is_hashmap hm_update"
  apply unfold_locales
  apply (sep_auto heap: hm_update_rule)
  done
interpretation hm: imp_map_update is_hashmap hm_update by (rule hm_update_impl)

lemma hm_delete_impl: "imp_map_delete is_hashmap hm_delete"
  apply unfold_locales
  apply (sep_auto heap: hm_delete_rule)
  done
interpretation hm: imp_map_delete is_hashmap hm_delete by (rule hm_delete_impl)

lemma hm_is_empty_impl: "imp_map_is_empty is_hashmap hm_isEmpty"
  apply unfold_locales
  apply (sep_auto heap: hm_isEmpty_rule)
  done
interpretation hm: imp_map_is_empty is_hashmap hm_isEmpty
  by (rule hm_is_empty_impl)

lemma hm_size_impl: "imp_map_size is_hashmap hm_size"
  apply unfold_locales
  apply (sep_auto heap: hm_size_rule)
  done
interpretation hm: imp_map_size is_hashmap hm_size by (rule hm_size_impl)

lemma hm_iterate_impl:
  "imp_map_iterate is_hashmap hm_is_it hm_it_init hm_it_has_next hm_it_next"
  apply unfold_locales
  apply (rule hm_it_init_rule)
  apply (sep_auto heap add: hm_it_next_rule)
```

```
  apply (sep_auto heap add: hm_it_has_next_rule)
  apply (rule ent_frame_fwd[OF hm_it_finish])
  apply (frame_inference)
  apply solve_entails
  done
interpretation hm:
  imp_map_iterate is_hashmap hm_is_it hm_it_init hm_it_has_next hm_it_next
  by (rule hm_iterate_impl)



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 <λr. 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 <λr. 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)>"



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. ∃A s'. is_set s' r *  is_set b q  * true * ↑ (s' = a ∪ it) >"
```
proof -
```
  interpret imp_set_iterate is_set is_it it_init it_has_next it_next
      + imp_set_ins is_set set_ins
    by fact+

  from FIN show ?thesis
  proof (induction  arbitrary: a p iti rule: finite_psubset_induct)
    case (psubset it)
    show ?case
      apply (subst set_it_union.simps)
      using imp_set_iterate_axioms
      apply (sep_auto heap: psubset.IH)
      by (metis ent_refl_true ent_star_mono_true quit_iteration star_aci(2))
  qed
```
qed

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.  ∃A s'. is_set s' r * true * is_set b q * ↑ (s' = a ∪ b)>"
```
proof -

```
    interpret
      imp_set_iterate is_set is_it it_init it_has_next it_next
        + imp_set_ins is_set set_ins
      by fact+

    note it_aux[sep_heap_rules] = set_it_union_rule[OF IT INS finb]
    show ?thesis
      unfolding union_loop_ins_def
       apply (sep_auto)
      done
  qed


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 ⇒ 'a⇀unit"
  where "map_of_set S x ≡ if x∈S then Some () else None"

lemma ne_some_unit_eq: "x≠Some () ⟷ x=None"
  by (cases x) auto

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 ⟷ x∉s"
  "map_of_set s x = Some u ⟷ x∈s"
  "(map_of_set s) (x↦()) = map_of_set (insert x s)"
  "(map_of_set s) |` (-{x}) = map_of_set (s -{x})"
  apply (auto simp: map_of_set_def
    dom_def ne_some_unit_eq restrict_map_def
    intro!: ext)
  done

lemma map_of_set_eq':
  "map_of_set a = map_of_set b ⟷ a=b"
  apply (auto simp: map_of_set_def[abs_def])
  apply (metis option.simps(3))+
  done

lemma map_of_set_eq[simp]:
```

```
  "map_of_set s = m ⟷ dom m=s"
  apply (auto
    simp: dom_def map_of_set_def[abs_def] ne_some_unit_eq
    intro!: ext)
  apply (metis option.simps(3))
  done
```

16.2 Main Definitions

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

lemma hs_set_impl: "imp_set is_hashset"
  apply unfold_locales
  apply rule
  unfolding is_hashset_def
  apply (subst map_of_set_eq'[symmetric])
  by (metis preciseD[OF is_hashmap_prec])
interpretation hs: imp_set is_hashset by (rule hs_set_impl)

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

lemma hs_new_impl: "imp_set_empty is_hashset hs_new"
  apply unfold_locales
  apply (sep_auto heap: hm_new_rule simp: is_hashset_def hs_new_def)
  done
interpretation hs: imp_set_empty is_hashset hs_new by (rule hs_new_impl)

definition hs_memb:: "'a::{heap,hashable} ⇒ 'a hashset ⇒ bool Heap"
  where "hs_memb x s ≡ do {
  r←hm_lookup x s;
  return (case r of Some _ ⇒ True | None ⇒ False)
}"

lemma hs_memb_impl: "imp_set_memb is_hashset hs_memb"
  apply unfold_locales
  unfolding hs_memb_def
  apply (sep_auto
    heap: hm_lookup_rule
    simp: is_hashset_def split: option.split)
  done
interpretation hs: imp_set_memb is_hashset hs_memb by (rule hs_memb_impl)

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"
  apply unfold_locales
```

```
  apply (sep_auto heap: hm_update_rule simp: hs_ins_def is_hashset_def)
  done
interpretation hs: imp_set_ins is_hashset hs_ins by (rule hs_ins_impl)

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"
  apply unfold_locales
  apply (sep_auto heap: hm_delete_rule simp: is_hashset_def hs_delete_def)
  done
interpretation hs: imp_set_delete is_hashset hs_delete
  by (rule hs_delete_impl)

definition "hs_isEmpty == hm_isEmpty"

lemma hs_is_empty_impl: "imp_set_is_empty is_hashset hs_isEmpty"
  apply unfold_locales
  apply (sep_auto heap: hm_isEmpty_rule simp: is_hashset_def hs_isEmpty_def)
  done
interpretation hs: imp_set_is_empty is_hashset hs_isEmpty
  by (rule hs_is_empty_impl)

definition "hs_size == hm_size"

lemma hs_size_impl: "imp_set_size is_hashset hs_size"
  apply unfold_locales
  apply (sep_auto heap: hm_size_rule simp: is_hashset_def hs_size_def)
  done
interpretation hs: imp_set_size is_hashset hs_size by (rule hs_size_impl)

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"
  apply unfold_locales
  unfolding hs_it_init_def hs_it_next_def hs_it_has_next_def
    hs_is_it_def is_hashset_def
  apply sep_auto
  apply sep_auto
  apply sep_auto
  apply (sep_auto eintros: hm.quit_iteration)
  done

interpretation hs: imp_set_iterate
  is_hashset hs_is_it hs_it_init hs_it_has_next hs_it_next
  by (rule hs_iterate_impl)

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"
  apply (unfold_locales)
  by (sep_auto)

interpretation hs: imp_set_union
  is_hashset hs_is_it hs_it_init hs_it_has_next hs_it_next hs_union
  by (rule hs_union_impl)




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. ∃_A l'. is_set s si
    * is_list l' r
    * ↑(set l' = set l ∪ it) >_t"
proof -
  interpret imp_set_iterate is_set is_it it_init it_has_next it_next
    + imp_list_prepend is_list l_prepend
    by fact+

  from FIN show ?thesis
  proof (induction arbitrary: l li iti rule: finite_psubset_induct)
    case (psubset it)
    show ?case
      apply (subst to_list_ga_rec.simps)
      apply (sep_auto heap: psubset.IH)
      apply (rule ent_frame_fwd[OF quit_iteration])
      apply frame_inference
      apply solve_entails
      done
  qed
qed
```

```
  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. ∃A l. is_set s si * is_list l r * true * ↑(set l = s)>"
  proof -
    interpret imp_list_empty is_list l_empty +
      imp_set_iterate is_set is_it it_init it_has_next it_next
      by fact+

    note [sep_heap_rules] = to_list_ga_rec_rule[OF IT PREP]
    show ?thesis
      unfolding to_list_ga_def
      by (sep_auto simp: FIN)
  qed
```

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_Numeral"
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))"*
  by *pat_completeness auto*

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"
  unfolding *ufa_invar_def* **by** *auto*

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

lemma *rep_of_refl: "l!i=i ⟹ rep_of l i = i"*
  apply *(subst rep_of.psimps)*
  apply *(rule rep_of.domintros)*
  apply *(auto)*
  done

lemma *rep_of_step:*
  "⟦ufa_invar l; i<length l; l!i≠i⟧ ⟹ rep_of l i = rep_of l (l!i)"
  apply *(subst rep_of.psimps)*

```
  apply (auto dest: ufa_invarD)
  done

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))"
  by (simp add: rep_of_simps)
```

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 -
  from ufa_invarD[OF I L] have "ufa_invar l ∧ i<length l ⟶ P l i"
    apply (induct l≡l i rule: rep_of.pinduct)
    apply (auto intro: STEP BASE dest: ufa_invarD)
    done
  thus ?thesis using I L by simp
qed
```

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 -
  have "⟦rep_of_dom (l,i) ⟧ ⟹ l!(rep_of l i) = rep_of l i"
    apply (induct arbitrary:  rule: rep_of.pinduct)
    apply (subst rep_of.psimps, assumption)
    apply (subst (2) rep_of.psimps, assumption)
    apply auto
    done
  thus "⟦ ufa_invar l; i<length l ⟧ ⟹ l!(rep_of l i) = rep_of l i"
    by (metis ufa_invarD(1))
qed

lemma rep_of_bound:
  "⟦ ufa_invar l; i<length l ⟧ ⟹ rep_of l i < length l"
  apply (induct rule: rep_of_induct)
  apply (auto simp: rep_of_iff)
  done

lemma rep_of_idem:
  "⟦ ufa_invar l; i<length l ⟧ ⟹ rep_of l (rep_of l i) = rep_of l i"
  by (auto simp: rep_of_min rep_of_refl)
```

```
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"
  by (metis list_update_id rep_of_min)

lemma rep_of_idx:
  "⟦ufa_invar l; i<length l⟧ ⟹ rep_of l (l!i) = rep_of l i"
  by (metis rep_of_step)
```

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)"
  apply rule
  unfolding ufa_α_def
  apply (rule symI)
  apply auto
  apply (rule transI)
  apply auto
  done

lemma ufa_α_lenD:
  "(x,y)∈ufa_α l ⟹ x<length l"
  "(x,y)∈ufa_α l ⟹ y<length l"
  unfolding ufa_α_def by auto

lemma ufa_α_dom[simp]: "Domain (ufa_α l) = {0..<length l}"
  unfolding ufa_α_def by auto

lemma ufa_α_refl[simp]: "(i,i)∈ufa_α l ⟷ i<length l"
  unfolding ufa_α_def
  by simp

lemma ufa_α_len_eq:
  assumes "ufa_α l = ufa_α l'"
  shows "length l = length l'"
  by (metis assms le_antisym less_not_refl linorder_le_less_linear ufa_α_refl)
```

18.1.3 Operations

```
lemma ufa_init_invar: "ufa_invar [0..<n]"
  unfolding ufa_invar_def
  by (auto intro: rep_of.domintros)

lemma ufa_init_correct: "ufa_α [0..<n] = {(x,x) | x. x<n}"
  unfolding ufa_α_def
  using ufa_init_invar[of n]
  apply (auto simp: rep_of_refl)
```

```
  done

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"
  unfolding ufa_α_def
  by auto

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)"
  unfolding ufa_invar_def
proof (intro allI impI, simp only: length_list_update)
  fix i
  assume A: "i<length l"
  with I have "rep_of_dom (l,i)" by (auto dest: ufa_invarD)

  have "ufa_union l x y ! i < length l" using I L A
    apply (cases "i=rep_of l x")
    apply (auto simp: rep_of_bound dest: ufa_invarD)
    done
  moreover have "rep_of_dom (ufa_union l x y, i)" using I A L
  proof (induct rule: rep_of_induct)
    case (base i)
    thus ?case
      apply -
      apply (rule rep_of.domintros)
      apply (cases "i=rep_of l x")
      apply auto
      apply (rule rep_of.domintros)
      apply (auto simp: rep_of_min)
      done
  next
    case (step i)

    from step.prems ‹ufa_invar l› ‹i<length l› ‹l!i≠i›
    have [simp]: "ufa_union l x y ! i = l!i"
      apply (auto simp: rep_of_min rep_of_bound nth_list_update)
      done

    from step show ?case
      apply -
      apply (rule rep_of.domintros)
      apply simp
      done
  qed
  ultimately show
```

```
    "rep_of_dom (ufa_union l x y, i) ∧ ufa_union l x y ! i < length l"
    by blast

qed

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)"
  using I IL
proof (induct rule: rep_of_induct)
  case (base i)
  have [simp]: "rep_of l i = i" using ‹l!i=i› by (simp add: rep_of_refl)
  note [simp] = ‹ufa_invar l› ‹i<length l›
  show ?case proof (cases)
    assume A[simp]: "rep_of l x = i"
    have [simp]: "l[i := rep_of l y] ! i = rep_of l y"
      by (auto simp: rep_of_bound)

    show ?thesis proof (cases)
      assume [simp]: "rep_of l y = i"
      show ?thesis by (simp add: rep_of_refl)
    next
      assume A: "rep_of l y ≠ i"
      have [simp]: "rep_of (l[i := rep_of l y]) i = rep_of l y"
        apply (subst rep_of_step[OF ufa_union_invar[OF I L], simplified])
        using A apply simp_all
        apply (subst rep_of_refl[where i="rep_of l y"])
        using I L
        apply (simp_all add: rep_of_min)
        done
      show ?thesis by (simp add: rep_of_refl)
    qed
  next
    assume A: "rep_of l x ≠ i"
    hence "ufa_union l x y ! i = l!i" by (auto)
    also note ‹l!i=i›
    finally have "rep_of (ufa_union l x y) i = i" by (simp add: rep_of_refl)
    thus ?thesis using A by auto
  qed
next
  case (step i)

  note [simp] = I L ‹i<length l›

  have "rep_of l x ≠ i" by (metis I L(1) rep_of_min ‹l!i≠i›)
  hence [simp]: "ufa_union l x y ! i = l!i"
```

```
    by (auto simp add: nth_list_update rep_of_bound ‹l!i≠i›) []

  have "rep_of (ufa_union l x y) i = rep_of (ufa_union l x y) (l!i)"
    by (auto simp add: rep_of_iff[OF ufa_union_invar[OF I L]])
  also note step.hyps(4)
  finally show ?case
    by (auto simp: rep_of_idx)
qed

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"
  unfolding ufa_α_def per_union_def
  by (auto simp: ufa_union_aux
    split: if_split_asm
  )

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 -
  {
    fix i
    assume "i<length (l[x := rep_of l x])"
    hence IL: "i<length l" by simp

    have G1: "l[x := rep_of l x] ! i < length (l[x := rep_of l x])"
      using I IL
      by (auto dest: ufa_invarD[OF I] simp: nth_list_update rep_of_bound)
    from I IL have G2: "rep_of (l[x := rep_of l x]) i = rep_of l i
      ∧ rep_of_dom (l[x := rep_of l x], i)"
    proof (induct rule: rep_of_induct)
      case (base i)
      thus ?case
        apply (cases "x=i")
        apply (auto intro: rep_of.domintros simp: rep_of_refl)
        done
    next
      case (step i)
      hence D: "rep_of_dom (l[x := rep_of l x], i)"
        apply -
        apply (rule rep_of.domintros)
        apply (cases "x=i")
        apply (auto intro: rep_of.domintros simp: rep_of_min)
        done

      thus ?case apply simp using step
        apply -
```

```
        apply (subst rep_of.psimps[OF D])
        apply (cases "x=i")
        apply (auto simp: rep_of_min rep_of_idx)
        apply (subst rep_of.psimps[where i="rep_of l i"])
        apply (auto intro: rep_of.domintros simp: rep_of_min)
        done
    qed
    note G1 G2
  } note G=this

  thus "∀i<length l. rep_of (l[x := rep_of l x]) i = rep_of l i"
    by auto

  from G show "ufa_invar (l[x := rep_of l x])"
    by (auto simp: ufa_invar_def)
qed

lemma ufa_compress_invar:
  assumes I: "ufa_invar l"
  assumes L[simp]: "x<length l"
  shows "ufa_invar (l[x := rep_of l x])"
  using assms by (rule ufa_compress_aux)

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"
  by (auto simp: ufa_α_def ufa_compress_aux[OF I])
```

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}>"
  unfolding uf_init_def is_uf_def[abs_def]
  by (sep_auto simp: ufa_init_correct ufa_init_invar)

partial_function (heap) uf_rep_of :: "nat array ⇒ nat ⇒ nat Heap"
  where [code]:
  "uf_rep_of p i = do {
    n ← 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⟧ ⟹
  <p↦ₐl> uf_rep_of p i <λr. p↦ₐl * ↑(r=rep_of l i)>"
  apply (induct rule: rep_of_induct)
  apply (subst uf_rep_of.simps)
  apply (sep_auto simp: rep_of_refl)

  apply (subst uf_rep_of.simps)
  apply (sep_auto simp: rep_of_step)
  done
```

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

```
partial_function (heap) uf_compress :: "nat ⇒ nat ⇒ nat array ⇒ unit
Heap"
  where [code]:
  "uf_compress i ci p = (
    if i=ci then return ()
    else do {
      ni←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 ⟧ ⟹
  <p↦ₐl> uf_compress i ci p
  <λ_. ∃ₐl'. p↦ₐl' * ↑(ufa_invar l' ∧ length l' = length l
     ∧ (∀i<length l. rep_of l' i = rep_of l i))>"
proof (induction rule: rep_of_induct)
  case (base i) thus ?case
    apply (subst uf_compress.simps)
    apply (sep_auto simp: rep_of_refl)
    done
next
  case (step i)
```

```
  note SS = <ufa_invar l> <i<length l> <l!i≠i> <ci = rep_of l i>

  from step.IH
  have IH':
    "<p ↦a l>
       uf_compress (l ! i) (rep_of l i) p
     <λ_. ∃A l'. p ↦a l' *
        ↑ (ufa_invar l' ∧ length l = length l'
           ∧ (∀i<length l'. rep_of l i = rep_of l' i))
     >"
    apply (simp add: rep_of_idx SS)
    apply (erule
      back_subst[OF _ cong[OF cong[OF arg_cong[where f=hoare_triple]]]])
    apply (auto) [2]
    apply (rule ext)
    apply (rule ent_iffI)
    apply sep_auto+
    done

  show ?case
    apply (subst uf_compress.simps)
    apply (sep_auto simp: SS)

    apply (rule IH')

    using SS apply (sep_auto (plain))
    using ufa_compress_invar apply fastforce []
    apply simp
    using ufa_compress_aux(2) apply fastforce []
    done
qed

definition uf_rep_of_c :: "nat array ⇒ nat ⇒ nat Heap"
  where "uf_rep_of_c p i ≡ do {
    ci←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⟧ ⟹
  <p↦al> uf_rep_of_c p i <λr. ∃A l'. p↦al'
    * ↑(r=rep_of l i ∧ ufa_invar l'
       ∧ length l' = length l
       ∧ (∀i<length l. rep_of l' i = rep_of l i))>"
  unfolding uf_rep_of_c_def
  by (sep_auto heap: uf_compress_rule)

definition uf_cmp :: "uf ⇒ nat ⇒ nat ⇒ 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:
  "⟦(∀i<length l. rep_of l' i = rep_of l i); length l = length l'⟧
  ⟹ (ufa_α l = ufa_α l')"
  unfolding ufa_α_def by auto

lemma uf_cmp_rule[sep_heap_rules]:
  "<is_uf R u> uf_cmp u i j <λr. is_uf R u * ↑(r⟷(i,j)∈R)>"
  unfolding uf_cmp_def is_uf_def
  apply (sep_auto dest: ufa_α_lenD simp: not_le split: prod.split)
  apply (drule cnv_to_ufa_α_eq, simp_all)
  apply (drule cnv_to_ufa_α_eq, simp_all)
  apply (drule cnv_to_ufa_α_eq, simp_all)
  apply (drule cnv_to_ufa_α_eq, simp_all)
  apply (drule cnv_to_ufa_α_eq, simp_all)
  apply (drule cnv_to_ufa_α_eq, simp_all)
  apply (subst ufa_find_correct)
  apply (auto simp add: )
  done


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]: "⟦i∈Domain R; j∈ Domain R⟧
  ⟹ <is_uf R u> uf_union u i j <is_uf (per_union R i j)>"
  unfolding uf_union_def
  apply (cases u)
  apply (simp add: is_uf_def[abs_def])
  apply (sep_auto
    simp: per_union_cmp ufa_α_lenD ufa_find_correct
    rep_of_bound
    ufa_union_invar
    ufa_union_correct
  )
  done


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"
  unfolding test_def
  apply (sep_auto)
  done
```

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
  <λr. is_hashtable l ht * is_hashtable (ls_rehash l) r>"
proof -
  have LEN: " l ≠ [] ⟹ Suc 0 < 2 * length l" by (cases l) auto
  show ?thesis
    apply (rule cons_pre_rule[OF ht_imp_len])
    unfolding ht_rehash_def
    apply (sep_auto
      heap: complete_ht_new_sz complete_ht_copy
      simp: ls_rehash_def LEN
    ) — Here we add a heap-rule, and some simp-rules
    done
qed
```

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)>"
```

unfolding *os_prepend_def*

The rules to deconstruct compound statements are contained in the *sep_decon_rules* collection

thm *sep_decon_rules*
apply *(rule sep_decon_rules)*

The rules for statement that deend on the heap are contained in the *sep_heap_rules* collection. The *fi_rule*-lemma prepares frame inference for them

apply *(rule sep_heap_rules[THEN fi_rule])*
apply *frame_inference* — This method does the frame-inference

The consequence rule comes in three versions, *const_rule*, *cons_pre_rule*, and *cons_post_rule*

apply *(rule cons_post_rule)*
apply *(rule sep_decon_rules)*

A simplification unfolds *os_list* and extract the pure part of the assumption

apply *(clarsimp)*

We can use *ent_ex_postI* to manually introduce existentials in entailsments

apply *(rule_tac x=xa* **in** *ent_ex_postI)*
apply *(rule_tac x=n* **in** *ent_ex_postI)*

The simplifier has a setup for assertions, so it will do the rest

apply *simp*
done

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)>"
unfolding *os_prepend_def* **by** *sep_auto*

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])>"
apply *(cases p)*
apply *(sep_auto simp: cs_append.simps)*

apply *(sep_auto simp: cs_append.simps heap: lseg_append)*

At this point, we are left with an entailment subgoal that sep-auto cannot solve. A closer look reveals that we could use the rule *lseg_append*.

With the *ent_frame_fwd*-rule, we can manually apply a rule to solve an entailment, involving frame inference. In this case, we have the additional problem that frame-inference guesses a wrong instantiation, and is not able to infer the frame. So we have to pre-instantiate the rule, as done below.

apply *(rule_tac s1=a* **in** *ent_frame_fwd[OF lseg_append])*
apply *frame_inference* — Now frame-inference is able to infer the frame

Now we are left with a trivial entailment, modulo commutativity of star. This can be handled by the entailment solver:

apply *solve_entails*
done

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 ⇒ 'x hashset ⇒ 'x hashset Heap"
  where
  "ins_from_list [] hs = return hs" |
  "ins_from_list (x # l) hs = do { hs ← 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 (induction l arbitrary: hs s)
  case (Cons x l)
```

In the induction step, the induction hypothesis has to be declared as a heap-rule, as *sep_auto* currently does not look for potential heap-rules among the premises of the subgoal

```
  show ?case by (sep_auto heap: Cons.IH)
qed sep_auto
```

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 <...> ... <...> 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 (induct l' arbitrary: it s)
  case Nil thus ?case
```

To unfold the definition of a partial function, we have to use *subst*. Note that *simp* would loop, unfolding the function arbitrarily deep

```
    apply (subst os_sum'.simps)
```

sep_auto accepts all the section parameters that *auto* does, eg. *intro:*

```
    apply (sep_auto intro: os.quit_iteration ent_true_drop)
    done
next
  case (Cons x l')
  show ?case
    apply (subst os_sum'.simps)
```

Additionally, *sep_auto* accepts some more section parameters. The most common one, *heap:*, declares rules to be used with frame inference. See Section 5.9 for a complete overview.

```
    apply (sep_auto heap: Cons.hyps)
    done
qed
```

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'"
  apply (intro allI)
```

```
  proof (induct l arbitrary: p F1 F2)
    case Nil thus ?case
      apply simp
```
— A precision solver for references and arrays is included in the standard simplifier setup. Building a general precision solver remains future work.
```
      by metis
```
— Unfortunately, the simplifier cannot cope with arbitrarily directed equations, so we have to use some more powerful tool
```
  next
    case (Cons x l)
    show ?case
      apply clarsimp

      apply (subgoal_tac "na=n")
```

The *prec_frame* and *prec_frame'* rules are useful to do precision proofs

```
      apply (erule prec_frame'[OF Cons.hyps])
      apply frame_inference
      apply frame_inference

      apply (drule prec_frame[OF sngr_prec])
      apply frame_inference
      apply frame_inference
      apply simp
      done
  qed
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.