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"
  ⟨*proof*⟩

lemma *run_effectI:*
  assumes *"run c (Some h) (Some h') r"*
  shows *"effect c h h' r"*
  ⟨*proof*⟩

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

⟨*proof*⟩

lemma *success_run:*
assumes *"success f h"*
obtains *h' r* **where** *"run f (Some h) (Some h') r"*
⟨*proof*⟩

run always yields a result

lemma *run_complete:*
obtains *σ' r* **where** *"run c σ σ' r"*
⟨*proof*⟩

lemma *run_detE:*
assumes *"run c σ σ' r" "run c σ τ s"*
"¬is_exn σ"
obtains *"is_exn σ'" "σ' = τ" | "¬ is_exn σ'" "σ' = τ" "r = s"*
⟨*proof*⟩

lemma *run_detI:*
assumes *"run c (Some h) (Some h') r" "run c (Some h) σ s"*
shows *"σ = Some h' ∧ r = s"*
⟨*proof*⟩

lemma *run_exn:*
assumes *"run f σ σ' r"*
"is_exn σ"
obtains *"σ'=σ"*
⟨*proof*⟩

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"
⟨*proof*⟩

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"
⟨*proof*⟩

lemma *run_return[run_elims]:*

assumes *"run (return x) σ σ' r"*
obtains *"r = x" "σ' = σ" "¬ is_exn σ" | "σ = None"*
⟨*proof*⟩

lemma *run_raise_iff: "run (raise s) σ σ' r ⟷ (σ'=None)"*
⟨*proof*⟩

lemma *run_raise[run_elims]:*
assumes *"run (raise s) σ σ' r"*
obtains *"σ' = None"*
⟨*proof*⟩

lemma *run_raiseI:*
"run (raise s) σ None r" ⟨*proof*⟩

lemma *run_if[run_elims]:*
assumes *"run (if c then t else e) h h' r"*
obtains *"c" "run t h h' r"*
| *"¬c" "run e h h' r"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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)))"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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"
  ⟨proof⟩

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])"
  ⟨proof⟩

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"
  ⟨proof⟩


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"
  ⟨proof⟩


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"
  ⟨proof⟩

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"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

lemma *"fst (Ref.alloc x h) = Ref (lim h)"*
⟨*proof*⟩

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 = ()"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

end

3 Assertions

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

3.1 Partial Heaps

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

type_synonym *pheap = "heap × addr set"*

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

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

declare *in_range.simps[simp del]*

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

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

lemma *in_range_subset:*
  "⟦as ⊆ as'; in_range (h,as')⟧ ⟹ in_range (h,as)"
  ⟨*proof*⟩

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

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

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

Reflexivity

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

Symmetry

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

Transitivity

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

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

⟨*proof*⟩

lemma *relH_subset:*
assumes *"relH bs h h'"*
assumes *"as ⊆ bs"*
shows *"relH as h h'"*
⟨*proof*⟩

lemma *relH_ref:*
assumes *"relH as h h'"*
assumes *"addr_of_ref r ∈ as"*
shows *"Ref.get h r = Ref.get h' r"*
⟨*proof*⟩

lemma *relH_array:*
assumes *"relH as h h'"*
assumes *"addr_of_array r ∈ as"*
shows *"Array.get h r = Array.get h' r"*
⟨*proof*⟩

lemma *relH_set_ref: "⟦ addr_of_ref r ∉ as; in_range (h,as)⟧*
⟹ relH as h (Ref.set r x h)"
⟨*proof*⟩

lemma *relH_set_array: "⟦addr_of_array r ∉ as; in_range (h,as)⟧*
⟹ relH as h (Array.set r x h)"
⟨*proof*⟩

3.2 Assertions

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

type_synonym *assn_raw = "pheap ⇒ 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"*
⟨*proof*⟩

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

shows *"in_range (h,as)"*
⟨*proof*⟩

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

lemmas *properD = properD1 properD2*

lemma *proper_iff:*
assumes *"proper P"*
assumes *"relH as h h'"*
assumes *"in_range (h',as)"*
shows *"P (h,as) ⟷ P (h',as)"*
⟨*proof*⟩

We encapsulate assertions in their own type

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

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

lemma *Abs_assn_eqI[intro?]:*
"(⋀h. P h = Rep_assn Pr h) ⟹ Abs_assn P = Pr"
"(⋀h. P h = Rep_assn Pr h) ⟹ Pr = Abs_assn P"
⟨*proof*⟩

abbreviation *models :: "pheap ⇒ assn ⇒ bool"* (**infix** *<⊨> 50*)
where *"h⊨P ≡ Rep_assn P h"*

lemma *models_in_range: "h⊨P ⟹ in_range h"*
⟨*proof*⟩

3.2.1 Empty Partial Heap

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

abbreviation *h_bot (<h⊥>)* **where** *"h⊥ ≡ (undefined,{})"*
lemma *mod_h_bot_indep: "(h,{})⊨P ⟷ (h',{})⊨P"*
⟨*proof*⟩

lemma *mod_h_bot_normalize[simp]:*

```
"syntax_fo_nomatch undefined h ⟹ (h,{})⊨P ⟷ h⊥ ⊨ P"
```
⟨*proof*⟩

Properness, lifted to the assertion type.

lemma `mod_relH: "relH as h h' ⟹ (h,as)⊨P ⟷ (h',as)⊨P"`
⟨*proof*⟩

3.3 Connectives

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

3.3.1 Empty Heap and Separation Conjunction

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

instantiation `assn :: one` **begin**
fun `one_assn_raw :: "pheap ⇒ bool"`
where `"one_assn_raw (h,as) ⟷ as={}"`

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

definition `one_assn :: assn` **where** `"1 ≡ Abs_assn one_assn_raw"`
instance ⟨*proof*⟩
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)"`
⟨*proof*⟩

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

instance ⟨*proof*⟩
end

lemma `mod_star_conv: "h⊨A*B`
`⟷ (∃hr as1 as2. h=(hr,as1∪as2) ∧ as1∩as2={} ∧ (hr,as1)⊨A ∧ (hr,as2)⊨B)"`
⟨*proof*⟩

lemma *mod_starD: "h⊨A*B ⟹ ∃h1 h2. h1⊨A ∧ h2⊨B"*
⟨*proof*⟩

lemma *star_assnI:*
assumes *"(h,as)⊨P"* **and** *"(h,as')⊨Q"* **and** *"as∩as'={}"*
shows *"(h,as∪as')⊨P*Q"*
⟨*proof*⟩

instantiation *assn :: comm_monoid_mult* **begin**
lemma *assn_one_left: "1*P = (P::assn)"*
⟨*proof*⟩

lemma *assn_times_comm: "P*Q = Q*(P::assn)"*
⟨*proof*⟩

lemma *assn_times_assoc: "(P*Q)*R = P*(Q*(R::assn))"*
⟨*proof*⟩

instance
⟨*proof*⟩

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)"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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)"
⟨*proof*⟩

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

definition *less_eq_assn* **where**
[simp]: "(a::assn) ≤ 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
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *star_false_right[simp]: "P * false = false"*
⟨*proof*⟩

lemmas *star_false = star_false_left star_false_right*

lemma *assn_basic_inequalities[simp, intro!]:*
"true ≠ emp" "emp ≠ true"
"false ≠ emp" "emp ≠ false"
"true ≠ false" "false ≠ true"
⟨*proof*⟩

3.3.4 Existential Quantification

definition *ex_assn :: "('a ⇒ assn) ⇒ assn"* (**binder** ‹∃$_A$› *11*)
where *"(∃$_A$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)"
⟨*proof*⟩

lemma *ex_assn_const[simp]: "(∃$_A$x. c) = c"*
⟨*proof*⟩

lemma *ex_one_point_gen:*
"⟦⋀h x. h⊨P x ⟹ x=v⟧ ⟹ (∃$_A$x. P x) = (P v)"
⟨*proof*⟩

lemma *ex_distrib_star: "(∃$_A$x. P x * Q) = (∃$_A$x. P x) * Q"*
⟨*proof*⟩

lemma *ex_distrib_and: "(∃$_A$x. P x ∧$_A$ Q) = (∃$_A$x. P x) ∧$_A$ Q"*
⟨*proof*⟩

lemma *ex_distrib_or: "(∃$_A$x. P x ∨$_A$ Q) = (∃$_A$x. P x) ∨$_A$ Q"*
⟨*proof*⟩

lemma *ex_join_or: "(∃$_A$x. P x ∨$_A$ (∃$_A$x. Q x)) = (∃$_A$x. P x ∨$_A$ Q x)"*
⟨*proof*⟩

3.3.5 Pure Assertions

Pure assertions do not depend on any heap content.

fun *pure_assn_raw* **where** *"pure_assn_raw b (h,as) ⟷ as={} ∧ b"*
definition *pure_assn :: "bool ⇒ assn"* (‹↑›) **where**
"↑b ≡ Abs_assn (pure_assn_raw b)"

lemma *pure_assn_proper[simp, intro!]: "proper (pure_assn_raw b)"*
⟨*proof*⟩

lemma *pure_true[simp]: "↑True = emp"*
⟨*proof*⟩

lemma *pure_false[simp]: "↑False = false"*
⟨*proof*⟩

lemma *pure_assn_eq_false_iff[simp]: "↑P = false ⟷ ¬P"* ⟨*proof*⟩

lemma *pure_assn_eq_emp_iff[simp]: "↑P = emp ⟷ P"* ⟨*proof*⟩

lemma *merge_pure_star[simp]:*
*"↑a * ↑b = ↑(a∧b)"*
⟨*proof*⟩

lemma *merge_true_star[simp]: "true*true = true"*
⟨*proof*⟩

lemma *merge_pure_and[simp]:*
"↑a ∧A ↑b = ↑(a∧b)"
⟨*proof*⟩

lemma *merge_pure_or[simp]:*
"↑a ∨A ↑b = ↑(a∨b)"
⟨*proof*⟩

lemma *pure_assn_eq_conv[simp]: "↑P = ↑Q ⟷ P=Q"* ⟨*proof*⟩

definition *"is_pure_assn a ≡ ∃P. a=↑P"*
lemma *is_pure_assnE:* **assumes** *"is_pure_assn a"* **obtains** *P* **where** *"a=↑P"*
⟨*proof*⟩

lemma *is_pure_assn_pure[simp, intro!]: "is_pure_assn (↑P)"*
⟨*proof*⟩

lemma *is_pure_assn_basic_simps[simp]:*
"is_pure_assn false"
"is_pure_assn emp"
⟨*proof*⟩

lemma *is_pure_assn_starI[simp,intro!]:*
*"⟦is_pure_assn a; is_pure_assn b⟧ ⟹ is_pure_assn (a*b)"*
⟨*proof*⟩

3.3.6 Pointers

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

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

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

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

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

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

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

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

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

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

3.4 Properties of the Models-Predicate

lemma *mod_true[simp]: "h⊨true ⟷ in_range h"*
⟨*proof*⟩
lemma *mod_false[simp]: "¬ h⊨false"*
⟨*proof*⟩

lemma *mod_emp: "h⊨emp ⟷ snd h = {}"*
⟨*proof*⟩

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

lemma *mod_pure[simp]: "h⊨⇑b ⟷ snd h = {} ∧ b"*
⟨*proof*⟩

lemma *mod_ex_dist[simp]: "h⊨(∃*$_A$*x. P x) ⟷ (∃x. h⊨P x)"*
⟨*proof*⟩

lemma *mod_exI: "∃x. h⊨P x ⟹ h⊨(∃*$_A$*x. P x)"*
⟨*proof*⟩
lemma *mod_exE:* **assumes** *"h⊨(∃*$_A$*x. P x)"* **obtains** *x* **where** *"h⊨P x"*
⟨*proof*⟩

lemma *mod_and_dist: "h⊨P∧*$_A$*Q ⟷ h⊨P ∧ h⊨Q"*
⟨*proof*⟩

lemma *mod_or_dist[simp]: "h⊨P∨*$_A$*Q ⟷ h⊨P ∨ h⊨Q"*
⟨*proof*⟩

lemma *mod_not_dist[simp]: "h⊨(¬*$_A$*P) ⟷ in_range h ∧ ¬ h⊨P"*
⟨*proof*⟩

lemma *mod_pure_star_dist[simp]: "h⊨P*⇑b ⟷ h⊨P ∧ b"*
⟨*proof*⟩

lemmas *mod_dist = mod_pure mod_pure_star_dist mod_ex_dist mod_and_dist mod_or_dist mod_not_dist*

lemma *mod_star_trueI: "h⊨P ⟹ h⊨P*true"*
⟨*proof*⟩

lemma *mod_star_trueE':* **assumes** *"h⊨P*true"* **obtains** *h'* **where** *"fst h' = fst h"* **and** *"snd h' ⊆ snd h"* **and** *"h'⊨P"*
⟨*proof*⟩

lemma *mod_star_trueE:* **assumes** *"h⊨P*true"* **obtains** *h'* **where** *"h'⊨P"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *entailsD:*
assumes *"P ⟹$_A$ Q"*
assumes *"h⊨P"*
shows *"h⊨Q"*
⟨*proof*⟩

3.5.1 Properties

lemma *ent_fwd:*
assumes *"h⊨P"*
assumes *"P ⟹$_A$ Q"*
shows *"h⊨Q"* ⟨*proof*⟩

lemma *ent_refl[simp]: "P ⟹$_A$ P"*
⟨*proof*⟩

lemma *ent_trans[trans]: "⟦ P ⟹$_A$ Q; Q ⟹$_A$R ⟧ ⟹ P ⟹$_A$ R"*
⟨*proof*⟩

lemma *ent_iffI:*
assumes *"A⟹$_A$B"*
assumes *"B⟹$_A$A"*
shows *"A=B"*
⟨*proof*⟩

lemma *ent_false[simp]: "false ⟹$_A$ P"*
⟨*proof*⟩
lemma *ent_true[simp]: "P ⟹$_A$ true"*
⟨*proof*⟩

lemma *ent_false_iff[simp]: "(P ⟹$_A$ false) ⟷ (∀h. ¬h⊨P)"*
⟨*proof*⟩

lemma *ent_pure_pre_iff[simp]: "(P*↑b ⟹$_A$ Q) ⟷ (b ⟶ (P ⟹$_A$ Q))"*
⟨*proof*⟩

lemma *ent_pure_pre_iff_sng[simp]:*
"(↑b ⟹$_A$ Q) ⟷ (b ⟶ (emp ⟹$_A$ Q))"
⟨*proof*⟩

lemma *ent_pure_post_iff[simp]:*
"(P $\Longrightarrow_A$ Q*↑b) $\longleftrightarrow$ ((∀h. h⊨P $\longrightarrow$ b) ∧ (P $\Longrightarrow_A$ Q))"
⟨*proof*⟩

lemma *ent_pure_post_iff_sng[simp]:*
"(P $\Longrightarrow_A$ ↑b) $\longleftrightarrow$ ((∀h. h⊨P $\longrightarrow$ b) ∧ (P $\Longrightarrow_A$ emp))"
⟨*proof*⟩

lemma *ent_ex_preI:* "(⋀x. P x $\Longrightarrow_A$ Q) $\Longrightarrow$ $\exists_A$x. P x $\Longrightarrow_A$ Q"
⟨*proof*⟩

lemma *ent_ex_postI:* "(P $\Longrightarrow_A$ Q x) $\Longrightarrow$ P $\Longrightarrow_A$ $\exists_A$x. Q x"
⟨*proof*⟩

lemma *ent_mp:* "(P * (P -* Q)) $\Longrightarrow_A$ Q"
⟨*proof*⟩

lemma *ent_star_mono:* "⟦ P $\Longrightarrow_A$ P'; Q $\Longrightarrow_A$ Q'⟧ $\Longrightarrow$ P*Q $\Longrightarrow_A$ P'*Q'"
⟨*proof*⟩

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

lemma *ent_disjI1:*
assumes "P $\vee_A$ Q $\Longrightarrow_A$ R"
shows "P $\Longrightarrow_A$ R" ⟨*proof*⟩

lemma *ent_disjI2:*
assumes "P $\vee_A$ Q $\Longrightarrow_A$ R"
shows "Q $\Longrightarrow_A$ R" ⟨*proof*⟩

lemma *ent_disjI1_direct[simp]:* "A $\Longrightarrow_A$ A $\vee_A$ B"
⟨*proof*⟩

lemma *ent_disjI2_direct[simp]:* "B $\Longrightarrow_A$ A $\vee_A$ B"
⟨*proof*⟩

lemma *ent_disjE:* "⟦ A$\Longrightarrow_A$C; B$\Longrightarrow_A$C ⟧ $\Longrightarrow$ A$\vee_A$B $\Longrightarrow_A$C"
⟨*proof*⟩

lemma *ent_conjI:* "⟦ A$\Longrightarrow_A$B; A$\Longrightarrow_A$C ⟧ $\Longrightarrow$ A $\Longrightarrow_A$ B $\wedge_A$ C"
⟨*proof*⟩

lemma *ent_conjE1:* "⟦A$\Longrightarrow_A$C⟧ $\Longrightarrow$ A$\wedge_A$B$\Longrightarrow_A$C"
⟨*proof*⟩
lemma *ent_conjE2:* "⟦B$\Longrightarrow_A$C⟧ $\Longrightarrow$ A$\wedge_A$B$\Longrightarrow_A$C"
⟨*proof*⟩

lemma *star_or_dist1:*
*"(A ∨$_A$ B)*C = (A*C ∨$_A$ B*C)"*
⟨*proof*⟩

lemma *star_or_dist2:*
"C(A ∨$_A$ B) = (C*A ∨$_A$ C*B)"*
⟨*proof*⟩

lemmas *star_or_dist = star_or_dist1 star_or_dist2*

lemma *ent_disjI1': "A⟹$_A$B ⟹ A⟹$_A$B∨$_A$C"*
⟨*proof*⟩

lemma *ent_disjI2': "A⟹$_A$C ⟹ A⟹$_A$B∨$_A$C"*
⟨*proof*⟩

lemma *triv_exI[simp, intro!]: "Q x ⟹$_A$ ∃$_A$x. Q x"*
⟨*proof*⟩

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"* ⟨*proof*⟩
lemma *enttD: "A⟹$_t$B ⟹ A⟹$_A$B*true"* ⟨*proof*⟩

lemma *entt_trans:*
"entailst A B ⟹ entailst B C ⟹ entailst A C"
⟨*proof*⟩

lemma *entt_refl[simp, intro!]: "entailst A A"*
⟨*proof*⟩

lemma *entt_true[simp, intro!]:*
"entailst A true"
⟨*proof*⟩

lemma *entt_emp[simp, intro!]:*
"entailst A emp"
⟨*proof*⟩

lemma *entt_star_true_simp[simp]:*
*"entailst A (B*true) ⟷ entailst A B"*
*"entailst (A*true) B ⟷ entailst A B"*

⟨*proof*⟩

lemma *entt_star_mono:* "⟦*entailst A B; entailst C D*⟧ ⟹ *entailst (A*C) (B*D)*"
⟨*proof*⟩

lemma *entt_frame_fwd:*
assumes "*entailst P Q*"
assumes "*entailst A (P*F)*"
assumes "*entailst (Q*F) B*"
shows "*entailst A B*"
⟨*proof*⟩

lemma *enttI_true:* "*P*true* $\Longrightarrow_A$ *Q*true* ⟹ *P*$\Longrightarrow_t$*Q*"
⟨*proof*⟩

lemma *entt_def_true:* "(*P*$\Longrightarrow_t$*Q*) ≡ (*P*true* $\Longrightarrow_A$ *Q*true)*"
⟨*proof*⟩

lemma *ent_imp_entt:* "*P*$\Longrightarrow_A$*Q* ⟹ *P*$\Longrightarrow_t$*Q*"
⟨*proof*⟩

lemma *entt_disjI1_direct[simp]:* "*A* $\Longrightarrow_t$ *A* $\vee_A$ *B*"
⟨*proof*⟩

lemma *entt_disjI2_direct[simp]:* "*B* $\Longrightarrow_t$ *A* $\vee_A$ *B*"
⟨*proof*⟩

lemma *entt_disjI1':* "*A*$\Longrightarrow_t$*B* ⟹ *A*$\Longrightarrow_t$*B*$\vee_A$*C*"
⟨*proof*⟩

lemma *entt_disjI2':* "*A*$\Longrightarrow_t$*C* ⟹ *A*$\Longrightarrow_t$*B*$\vee_A$*C*"
⟨*proof*⟩

lemma *entt_disjE:* "⟦ *A*$\Longrightarrow_t$*M*; *B*$\Longrightarrow_t$*M* ⟧ ⟹ *A*$\vee_A$*B* $\Longrightarrow_t$ *M*"
⟨*proof*⟩

lemma *entt_disjD1:* "*A*$\vee_A$*B*$\Longrightarrow_t$*C* ⟹ *A*$\Longrightarrow_t$*C*"
⟨*proof*⟩

lemma *entt_disjD2:* "*A*$\vee_A$*B*$\Longrightarrow_t$*C* ⟹ *B*$\Longrightarrow_t$*C*"
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *preciseD:*
assumes *"precise R"*
assumes *"h ⊨ R a p * F ∧A R a' p * F'"*
shows *"a=a'"*
⟨*proof*⟩

lemma *preciseD':*
assumes *"precise R"*
assumes *"h ⊨ R a p * F"*
assumes *"h ⊨ R a' p * F'"*
shows *"a=a'"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *sngr_prec: "precise (λx p. p↦rx)"*
⟨*proof*⟩

lemma *snga_prec: "precise (λx p. p↦ax)"*
⟨*proof*⟩

end

4 Hoare-Triples

theory *Hoare_Triple*
imports *Run Assertions*
begin

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

4.1 Definition

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

definition *new_addrs :: "heap ⇒ 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"*
⟨*proof*⟩

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

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

definition *hoare_triple*
:: "assn ⇒ '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"*
⟨*proof*⟩

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'"*

⟨*proof*⟩

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

abbreviation *hoare_triple'*
  :: "assn ⇒ '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>"*
  ⟨*proof*⟩

lemma *frame_rule:*
  assumes *A: "<P> c <Q>"*
  shows *"<P*R> c <λx. Q x * R>"*
  ⟨*proof*⟩

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

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'>"*
  ⟨*proof*⟩

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

lemma *cons_rulet: "⟦P⟹$_t$P'; ⋀x. Q x ⟹$_t$ Q' x; <P'> c <Q>$_t$ ⟧ ⟹ <P> c <Q'>$_t$"*
  ⟨*proof*⟩

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

lemma *norm_pre_ex_rule:*
assumes *A:* "⋀x. <P x> f <Q>"
shows "<∃ A x. P x> f <Q>"
⟨*proof*⟩

lemma *norm_pre_pure_iff[simp]:*
"<P*↑b> f <Q> ⟷ (b ⟶ <P> f <Q>)"
⟨*proof*⟩

lemma *norm_pre_pure_iff_sng[simp]:*
"<↑b> f <Q> ⟷ (b ⟶ <emp> f <Q>)"
⟨*proof*⟩

lemma *norm_pre_pure_rule1:*
"⟦b ⟹ <P> f <Q>⟧ ⟹ <P*↑b> f <Q>" ⟨*proof*⟩

lemma *norm_pre_pure_rule2:*
"⟦ b ⟹ <emp> f <Q> ⟧ ⟹ <↑b> f <Q>" ⟨*proof*⟩

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>"
⟨*proof*⟩

4.2.2 Rules for Atomic Commands

lemma *ref_rule:*
"<emp> ref x <λr. r ↦r x>"
⟨*proof*⟩

lemma *lookup_rule:*
"<p ↦r x> !p <λr. p ↦r x * ↑(r = x)>"
⟨*proof*⟩

lemma *update_rule:*
"<p ↦r y> p := x <λr. p ↦r x>"
⟨*proof*⟩

lemma *update_wp_rule:*
"<r ↦r y * ((r ↦r x) -* (Q ()))> r := x <Q>"
⟨*proof*⟩

lemma *new_rule:*
"<emp> Array.new n x <λr. r ↦a replicate n x>"
⟨*proof*⟩

lemma *make_rule:* "<emp> Array.make n f <λr. r ↦a (map f [0 ..< n])>"

⟨*proof*⟩

lemma *of_list_rule: "<emp> Array.of_list xs <λr. r ↦$_a$ xs>"*
⟨*proof*⟩

lemma *length_rule:*
*"<a ↦$_a$ xs> Array.len a <λr. a ↦$_a$ xs * ↑(r = length xs)>"*
⟨*proof*⟩

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

lemma *nth_rule:*
*"⟦i < length xs⟧ ⟹ <a ↦$_a$ xs> Array.nth a i <λr. a ↦$_a$ xs * ↑(r = xs ! i)>"*
⟨*proof*⟩

lemma *upd_rule:*
"⟦i < length xs⟧ ⟹
<a ↦$_a$ xs>
Array.upd i x a
*<λr. (a ↦$_a$ (list_update xs i x)) * ↑(r = a)>"*
⟨*proof*⟩

lemma *freeze_rule:*
*"<a ↦$_a$ xs> Array.freeze a <λr. a ↦$_a$ xs * ↑(r = xs)>"*
⟨*proof*⟩

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

lemma *return_sp_rule:*
*"<P> return x <λr. P * ↑(r = x)>"*
⟨*proof*⟩

lemma *raise_iff:*
"<P> raise s <Q> ⟷ P = false"
⟨*proof*⟩

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

4.2.3 Rules for Composed Commands

lemma *bind_rule:*
assumes *T1: "<P> f <R>"*
assumes *T2: "⋀x. <R x> g x <Q>"*
shows *"<P> bind f g <Q>"*
⟨*proof*⟩

lemma *if_rule:*
assumes *"b ⟹ <P> f <Q>"*
assumes *"¬b ⟹ <P> g <Q>"*
shows *"<P> if b then f else g <Q>"*
⟨*proof*⟩

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>"*
⟨*proof*⟩

lemma *split_rule:*
assumes *P: "<P> c <R>"*
assumes *Q: "<Q> c <R>"*
shows *"<P ∨A Q> c <R>"*
⟨*proof*⟩

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

lemma *case_prod_rule:*
"(⋀a b. x = (a, b) ⟹ <P> f a b <Q>) ⟹ <P> case x of (a, b) ⇒ f a b <Q>"
⟨*proof*⟩

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>"
⟨*proof*⟩

lemma *case_option_rule:*
"⟦ v=None ⟹ <P> fn <Q>; ⋀x. v=Some x ⟹ <P> fs x <Q> ⟧ ⟹ <P> case_option fn fs v <Q>"
⟨*proof*⟩

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>"
⟨*proof*⟩

lemma *let_rule: "(⋀x. x = t ⟹ <P> f x <Q>) ⟹ <P> Let t f <Q>"*
⟨*proof*⟩

end

5 Automation

theory *Automation*
imports *Hoare_Triple*
begin

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

5.1 Normalization of Assertions

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

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

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

lemma *merge_true_star_ctx: "true * (true * P) = true * P"*
  ⟨*proof*⟩

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

Move existential quantifiers to the front of assertions

lemma *ex_assn_move_out[simp]:*
  *"⋀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 (∃A x. P x) = (∃A x. (Q ∧A P x))"

  "⋀P Q. (∃A x. Q x) ∨A P = (∃A x. (Q x ∨A P))"
  "⋀P Q. Q ∨A (∃A x. P x) = (∃A x. (Q ∨A P x))"
  ⟨*proof*⟩

Extract pure assertions from and-clauses

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

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

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

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

lemmas *and_extract_pure_iff =*
and_extract_pure_left_iff and_extract_pure_left_ctx_iff
and_extract_pure_right_iff and_extract_pure_right_ctx_iff

lemmas *norm_assertion_simps =*

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

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

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

ex_assn_move_out ex_assn_const

and_extract_pure_iff

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

sngr_same_false snga_same_false

5.1.1 Simplifier Setup Fine-Tuning

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

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

⟨*proof*⟩
declare *Ref.noteq_irefl[dest!]*

lemma *not_same_noteqa[simp]: "¬ a=!!=a"*
⟨*proof*⟩
declare *Array.noteq_irefl[dest!]*

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

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 (∃Ax. Q x))"
⟨*proof*⟩

Example of how to build an extraction lemma.

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

lemmas *ent_triv = ent_true ent_false*

Dummy rule to detect Hoare triple goal

lemma *is_hoare_triple: "<P> c <Q> ⟹ <P> c <Q>"* ⟨*proof*⟩

Dummy rule to detect entailment goal

lemma *is_entails: "P⟹AQ ⟹ P ⟹AQ"* ⟨*proof*⟩

5.3 Frame Matcher

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

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

definition *[simp]: "SLN ≡ emp"*
lemma *SLN_left: "SLN * P = P"* ⟨*proof*⟩
lemma *SLN_right: "P * SLN = P"* ⟨*proof*⟩

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

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

definition *[simp]: "FI_QUERY P Q F ≡ 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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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 ⟹A fi_m_snd m"*
⟨*proof*⟩

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 ⟹A q"*
assumes *"FI ((p,q)#m) (ps*up) (qs*uq) SLN SLN f"*
shows *"FI m (ps*p) (qs*q) up uq f"*

⟨*proof*⟩

No match

lemma *FI_p_nomatch:*
assumes *"FI m ps (qs*q) (p*up) uq f"*
shows *"FI m (ps*p) (qs*q) up uq f"*
⟨*proof*⟩

Head of q could not be matched

lemma *FI_q_nomatch:*
assumes *"FI m (SLN*up) qs SLN (q*uq) f"*
shows *"FI m SLN (qs*q) up uq f"*
⟨*proof*⟩

5.4 Frame Inference

lemma *frame_inference_init:*
assumes *"FI_QUERY P Q F"*
shows *"P* $\Longrightarrow_A$ *Q * F"*
⟨*proof*⟩

lemma *frame_inference_finalize:*
shows *"FI_RESULT M F emp F"*
⟨*proof*⟩

5.5 Entailment Solver

lemma *entails_solve_init:*
"FI_QUERY P Q true $\Longrightarrow$ *P* $\Longrightarrow_A$ *Q * true"*
"FI_QUERY P Q emp $\Longrightarrow$ *P* $\Longrightarrow_A$ *Q"*
⟨*proof*⟩

lemma *entails_solve_finalize:*
"FI_RESULT M P emp true"
"FI_RESULT M emp emp emp"
⟨*proof*⟩

lemmas *solve_ent_preprocess_simps =*
ent_pure_post_iff ent_pure_post_iff_sng ent_pure_pre_iff ent_pure_pre_iff_sng

5.6 Verification Condition Generator

lemmas *normalize_rules = norm_pre_ex_rule norm_pre_pure_rule*

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

lemmas *return_cons_rule = cons_pre_rule[OF _ return_wp_rule]*

Useful frame-rule variant for manual proof:

lemma *frame_rule_left:*
*"<P> c <Q> ⟹ <R * P> c <λx. R * Q x>"*
⟨*proof*⟩

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

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

lemma *fi_rule:*
assumes *CMD: "<P> c <Q>"*
assumes *FRAME: "Ps* $\Longrightarrow_A$ *P * F"*
shows *"<Ps> c <λx. Q x * F>"*
⟨*proof*⟩

5.7 ML-setup

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

⟨*ML*⟩

lemmas *[sep_dflt_simps] = split*

declare *deconstruct_rules[sep_decon_rules]*
declare *heap_rules[sep_heap_rules]*

lemmas *[sep_eintros] = impI conjI exI*

5.8 Semi-Automatic Reasoning

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

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

lemma *ent_frame_fwd:*
assumes *R: "P* $\Longrightarrow_A$ *R"*
assumes *F: "Ps* $\Longrightarrow_A$ *P*F"*
assumes *I: "R*F* $\Longrightarrow_A$ *Q"*
shows *"Ps* $\Longrightarrow_A$ *Q"*
⟨*proof*⟩

lemma *mod_frame_fwd:*
assumes *M: "h*$\models$*Ps"*
assumes *R: "P*$\Longrightarrow_A$*R"*
assumes *F: "Ps* $\Longrightarrow_A$ *P*F"*
shows *"h*$\models$*R*F"*
⟨*proof*⟩

Apply precision rule with frame inference.

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

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

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

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

lemma *ent_wand_frameI:*
assumes *"(Q -* R) * F* $\Longrightarrow_A$ *S"*
assumes *"P* $\Longrightarrow_A$ *F * X"*
assumes *"Q*X* $\Longrightarrow_A$ *R"*
shows *"P* $\Longrightarrow_A$ *S"*
⟨*proof*⟩

5.8.1 Manual Frame Inference

lemma *ent_true_drop:*
*"P⟹$_A$Q*true ⟹ P*R⟹$_A$Q*true"*
*"P⟹$_A$Q ⟹ P⟹$_A$Q*true"*
⟨*proof*⟩

lemma *fr_refl: "A⟹$_A$B ⟹ A*C ⟹$_A$B*C"*
⟨*proof*⟩

lemma *fr_rot: "(A*B ⟹$_A$ C) ⟹ (B*A ⟹$_A$ C)"*
⟨*proof*⟩

lemma *fr_rot_rhs: "(A ⟹$_A$ B*C) ⟹ (A ⟹$_A$ C*B)"*
⟨*proof*⟩

lemma *ent_star_mono_true:*
assumes *"A ⟹$_A$ A' * true"*
assumes *"B ⟹$_A$ B' * true"*
shows *"A*B*true ⟹$_A$ A'*B'*true"*
⟨*proof*⟩

lemma *ent_refl_true: "A ⟹$_A$ A * true"*
⟨*proof*⟩

lemma *entt_fr_refl: "F⟹$_t$F' ⟹ F*A ⟹$_t$ F'*A"* ⟨*proof*⟩
lemma *entt_fr_drop: "F⟹$_t$F' ⟹ F*A ⟹$_t$ F'"*
⟨*proof*⟩

⟨*ML*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*pro*

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.

⟨*ML*⟩

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

⟨*ML*⟩

Selector Functions

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

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

Encoding to natural numbers, as required by Imperative/HOL

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

instance *node :: (heap) heap*
  ⟨*proof*⟩

8.2 List Segment Assertion

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

fun *lseg*
  :: "'a::heap list ⇒ 'a node ref option ⇒ 'a node ref option ⇒ assn"
  where
  "lseg [] p s = ↑(p=s)"
| "lseg (x#l) (Some p) s = (∃$_A$*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=[])"
  ⟨*proof*⟩

lemma *lseg_if_splitf2[simp, sep_dflt_simps]:*
"lseg (x#xs) p q
*= (∃ₐpp n. pp ↦ᵣ (Node x n) * lseg xs n q * ↑(p=Some pp))"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *lseg_append:*
*"lseg l p (Some s) * s↦ᵣNode x q ⟹ₐ lseg (l@[x]) p q"*
⟨*proof*⟩

lemma *lseg_conc: "lseg l1 p q * lseg l2 q r ⟹ₐ lseg (l1@l2) p r"*
⟨*proof*⟩

8.3.2 Splitting

lemma *lseg_split:*
*"lseg (l1@l2) p r ⟹ₐ ∃ₐq. lseg l1 p q * lseg l2 q r"*
⟨*proof*⟩

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'"
⟨*proof*⟩

lemma *lseg_prec2:*
"∀ l l'. (h⊨
*(lseg l p None * F1) ∧ₐ (lseg l' p None * F2))*
⟶ l=l'"
⟨*proof*⟩

lemma *lseg_prec3:*
*"∀ q q'. h ⊨ (lseg l p q * F1) ∧ₐ (lseg l p q' * F2) ⟶ q=q'"*
⟨*proof*⟩

end

9 Open Singly Linked Lists

theory *Open_List*
imports *List_Seg Imp_List_Spec*
begin

9.1 Definitions

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

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

9.2 Precision

lemma *os_prec:*
"precise os_list"
⟨*proof*⟩

lemma *os_imp_list_impl: "imp_list os_list"*
⟨*proof*⟩
interpretation *os: imp_list os_list* ⟨*proof*⟩

9.3 Operations

9.3.1 Allocate Empty List

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

lemma *os_empty_rule: "<emp> os_empty <os_list []>"*
⟨*proof*⟩

lemma *os_empty_impl: "imp_list_empty os_list os_empty"*
⟨*proof*⟩
interpretation *os: imp_list_empty os_list os_empty* ⟨*proof*⟩

9.3.2 Emptiness check

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

definition *os_is_empty :: "'a::heap os_list ⇒ 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 = [])>"*
⟨*proof*⟩

lemma *os_is_empty_impl: "imp_list_is_empty os_list os_is_empty"*
⟨*proof*⟩
interpretation *os: imp_list_is_empty os_list os_is_empty*

⟨*proof*⟩

9.3.3 Prepend

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

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

lemma *os_prepend_rule:*
"<os_list xs n> os_prepend x n <os_list (x # xs)>"
⟨*proof*⟩

lemma *os_prepend_impl: "imp_list_prepend os_list os_prepend"*
⟨*proof*⟩
interpretation *os: imp_list_prepend os_list os_prepend*
⟨*proof*⟩

9.3.4 Pop

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

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

declare *os_pop.simps[simp del]*

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

⟨*proof*⟩

lemma *os_pop_impl: "imp_list_pop os_list os_pop"*
⟨*proof*⟩
interpretation *os: imp_list_pop os_list os_pop* ⟨*proof*⟩

9.3.5 Reverse

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

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

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

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

definition "os_reverse p = os_reverse_aux None p"

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

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

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

9.3.6 Remove

Remove all appearances of an element from a linked list.

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

lemma [simp, sep_dflt_simps]:
```

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

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

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

9.3.7 Iterator

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

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

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

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

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

9.3.8 List-Sum

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

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

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

lemma os_sum'_rule[sep_heap_rules]:
  "<os_is_it l p l' it>
    os_sum' it s
  <λr. os_list l p * ↑(r = s + sum_list l')>t"
⟨proof⟩

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"
  ⟨proof⟩

end
```

10 Circular Singly Linked Lists

```
theory Circ_List
imports List_Seg Imp_List_Spec
begin
```

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

10.1 Datatype Definition

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

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

```
fun cs_list :: "'a::heap list ⇒ '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=[])"
  ⟨proof⟩

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

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

10.2 Precision

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

lemma *cs_imp_list_impl: "imp_list cs_list"*
⟨*proof*⟩
interpretation *cs: imp_list cs_list* ⟨*proof*⟩

10.3 Operations

10.3.1 Allocate Empty List

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

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

lemma *cs_empty_impl: "imp_list_empty cs_list cs_empty"*
⟨*proof*⟩
interpretation *cs: imp_list_empty cs_list cs_empty* ⟨*proof*⟩

10.3.2 Prepend Element

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

declare *cs_prepend.simps [simp del]*

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

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

interpretation *cs: imp_list_prepend cs_list cs_prepend*
⟨*proof*⟩

10.3.3 Append Element

fun *cs_append :: "'a ⇒ 'a::heap cs_list ⇒ 'a cs_list Heap"* **where**

```
  "cs_append x None = do {
    p ← ref (Node x None);
    p:=Node x (Some p);
    return (Some p) }"
| "cs_append x (Some p) = do {
    n ← !p;
    q ← ref (Node (val n) (next n));
    p := Node x (Some q);
    return (Some q)
  }"
```

declare *cs_append.simps [simp del]*

lemma *cs_append_rule:*

```
  "<cs_list l p> cs_append x p <cs_list (l@[x])>"
```

⟨*proof*⟩

lemma *cs_append_impl: "imp_list_append cs_list cs_append"*
⟨*proof*⟩
interpretation *cs: imp_list_append cs_list cs_append*
⟨*proof*⟩

10.3.4 Pop First Element

fun *cs_pop :: "'a::heap cs_list ⇒ ('a×'a cs_list) Heap"* **where**

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

declare *cs_pop.simps[simp del]*

lemma *cs_pop_rule:*

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

⟨*proof*⟩

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

10.3.5 Rotate

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

declare *cs_rotate.simps [simp del]*

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

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

10.4 Test

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

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

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

export_code *test* **checking** *SML_imp*

⟨*ML*⟩

hide_const (**open**) *test test_result*

end

11 Interface for Maps

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

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

locale *imp_map =*
  fixes *is_map :: "('k ⇀ '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. ∃Al. ↑(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 ⟹A is_map m p * true"*

end

12 Hash-Tables

theory *Hash_Table*

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

12.1 Datatype

12.1.1 Definition

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

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

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

12.1.2 Storable on Heap

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

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

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

12.2 Assertions

12.2.1 Assertion for Hashtable

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

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

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

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

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

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

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

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

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

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

12.3 New

12.3.1 Definition

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

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

12.3.2 Complete Correctness

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

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

⟨*proof*⟩

lemma *ht_size_replicate[simp, intro!]: "ht_size (replicate n []) 0"*
⟨*proof*⟩
lemma *complete_ht_new_sz: "1 < n ⟹ <emp> ht_new_sz n <is_hashtable (replicate n [])>"*
⟨*proof*⟩

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

12.4 Lookup

12.4.1 Definition

fun *ls_lookup :: "'k ⇒ ('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))) >"
⟨*proof*⟩

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)))>"
⟨*proof*⟩

12.5 Update

12.5.1 Definition

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

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

lemma *ls_update_snd_set: "snd (ls_update k v l) ⟷ k ∈ set (map fst l)"*
⟨*proof*⟩

lemma *ls_update_fst_set: "set (fst (ls_update k v l)) ⊆ insert (k, v) (set l)"*
⟨*proof*⟩

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

lemma *ls_update_distinct: "distinct (map fst l) ⟹ distinct (map fst (fst (ls_update k v l)))"*
⟨*proof*⟩

lemma *ls_update_length: "length (fst (ls_update k v l))*
= (if (k ∈ set (map fst l)) then length l else Suc (length l))"
⟨*proof*⟩

lemma *ls_update_length_snd_True:*
"snd (ls_update k v l) ⟹ length (fst (ls_update k v l)) = length l"
⟨*proof*⟩

lemma *ls_update_length_snd_False:*
"¬ snd (ls_update k v l) ⟹ length (fst (ls_update k v l)) = Suc (length l)"
⟨*proof*⟩

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

12.5.2 Complete Correctness

lemma *ht_hash_update:*
assumes *"ht_hash ls"*
shows *"ht_hash (abs_update k v ls)"*
⟨*proof*⟩

lemma *ht_distinct_update:*
assumes *"ht_distinct l"*
shows *"ht_distinct (abs_update k v l)"*
⟨*proof*⟩

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

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

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

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

Alternative, more automatic proof

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

12.6 Delete

12.6.1 Definition

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

lemma *ls_delete_snd_set: "snd (ls_delete k l) ⟷ k ∈ set (map fst l)"*
⟨*proof*⟩

lemma *ls_delete_fst_set: "set (fst (ls_delete k l)) ⊆ set l"*
⟨*proof*⟩

lemma *ls_delete_fst_map_set:*
"distinct (map fst l) ⟹
set (map fst (fst (ls_delete k l))) = (set (map fst l)) - {k}"
⟨*proof*⟩

lemma *ls_delete_distinct: "distinct (map fst l) ⟹ distinct (map fst*
(fst (ls_delete k l)))"
⟨*proof*⟩

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

lemma *ls_delete_length_snd_True:*
"snd (ls_delete k l) ⟹ length (fst (ls_delete k l)) = length l - 1"
⟨*proof*⟩

lemma *ls_delete_length_snd_False:*

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

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

12.6.2 Complete Correctness

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

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

lemma *ht_size_delete1:*
```
  assumes size: "ht_size l n"
  assumes i: "i < length l"
  assumes snd: "snd (ls_delete k (l ! i))"
  shows "ht_size (l[i := fst (ls_delete k (l!i))]) (n - 1)"
```
⟨*proof*⟩

lemma *ht_size_delete2:*
```
  assumes size: "ht_size l n"
  assumes i: "i < length l"
```

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

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

Alternative, more automatic proof

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

12.7 Re-Hashing

12.7.1 Auxiliary Functions

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

Abstract version

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

lemma *ht_hash_ls_insls:*
assumes *"ht_hash l"*
shows *"ht_hash (ls_insls ls l)"*
⟨*proof*⟩

lemma *ht_distinct_ls_insls:*
assumes *"ht_distinct l"*
shows *"ht_distinct (ls_insls ls l)"*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *complete_ht_insls:*
"<is_hashtable ls ht> ht_insls xs ht <is_hashtable (ls_insls xs ls)>"
⟨*proof*⟩

fun *ht_copy :: "nat ⇒ ('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)"*
⟨*proof*⟩

lemma *ht_distinct_ls_copy:*
assumes *"ht_distinct l"*
shows *"ht_distinct (ls_copy n ss l)"*
⟨*proof*⟩

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

lemma *complete_ht_copy: "n ≤ 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*⟩

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*⟩

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)"*
  ⟨*proof*⟩

lemma *ht_distinct_ls_rehash: "ht_distinct (ls_rehash l)"*
  ⟨*proof*⟩

lemma *length_ls_rehash:*
  assumes *"1 < length l"*
  shows *"1 < length (ls_rehash l)"*
⟨*proof*⟩

lemma *ht_imp_len: "is_hashtable l ht ⟹A is_hashtable l ht * ↑(length l > 0)"*
  ⟨*proof*⟩

lemma *complete_ht_rehash:*
```
  "<is_hashtable l ht> ht_rehash ht
  <λr. is_hashtable l ht * is_hashtable (ls_rehash l) r>"
```
  ⟨*proof*⟩

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

definition *ht_update*
```
  :: "'k::{heap,hashable} ⇒ '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)>"
```
⟨*proof*⟩

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

12.8 Conversion to List

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

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

end
Documentation

13 Hash-Maps

theory *Hash_Map*
imports *Hash_Table*
begin

13.1 Auxiliary Lemmas

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

⟨*proof*⟩

lemma *map_of_concat:*
"k ∈ dom (map_of(concat l))
⟹ ∃i. k ∈ dom (map_of(l!i)) ∧ i < length l"
⟨*proof*⟩

lemma *map_of_concat':*
"k ∈ dom (map_of(l!i)) ∧ i < length l ⟹ k ∈ dom (map_of(concat l))"
⟨*proof*⟩

lemma *map_of_concat''':*
assumes *"∃i. k ∈ dom (map_of(l!i)) ∧ i < length l"*
shows *"k ∈ dom (map_of(concat l))"*
⟨*proof*⟩

lemma *map_of_concat'':*
"(k ∈ dom (map_of(concat l)))
⟷ (∃i. k ∈ dom (map_of(l!i)) ∧ i < length l)"
⟨*proof*⟩

lemma *abs_update_length: "length (abs_update k v l) = length l"*
⟨*proof*⟩

lemma *ls_update_map_of_eq:*
"map_of (fst (ls_update k v ls)) k = Some v"
⟨*proof*⟩

lemma *ls_update_map_of_neq:*
"x ≠ k ⟹ map_of (fst (ls_update k v ls)) x = map_of ls x"
⟨*proof*⟩

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'"
⟨*proof*⟩

lemma *is_hashmap_prec: "precise is_hashmap"*
⟨*proof*⟩

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

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

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

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

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

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

lemma *ht_hash_in_dom_i_bounded_hashcode_nat_i:*
assumes *"ht_hash l"*

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

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

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

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

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

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

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

lemma *hm_lookup_rule:*
"<is_hashmap m ht> hm_lookup k ht
<λr. is_hashmap m ht ∗
↑(r = m k)>"
⟨*proof*⟩

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

lemma *abs_update_map_of_hceq:*
assumes *"ht_hash l"*
assumes *"ht_distinct l"*
assumes *"1 < length l"*
assumes *"x ≠ k"*
assumes *"bounded_hashcode_nat (length l) x*
= bounded_hashcode_nat (length l) k"
shows *"map_of (concat (abs_update k v l)) x = map_of (concat l) x"*
⟨*proof*⟩

lemma *abs_update_map_of_hcneq:*
assumes *"ht_hash l"*
assumes *"ht_distinct l"*
assumes *"1 < length l"*
assumes *"x ≠ k"*
assumes *"bounded_hashcode_nat (length l) x*
≠ bounded_hashcode_nat (length l) k"
shows *"map_of (concat (abs_update k v l)) x = map_of (concat l) x"*
⟨*proof*⟩

lemma *abs_update_map_of''':*
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"*
⟨*proof*⟩

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

⟨*proof*⟩

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

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

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

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

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

⟨*proof*⟩

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

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

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

lemma *ls_delete_map_of:*
assumes *"distinct (map fst l)"*
shows *"map_of (fst (ls_delete k l)) x = ((map_of l) |` (- {k})) x"*
⟨*proof*⟩

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

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

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

⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

13.3 Iterators

13.3.1 Definitions

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

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

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

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

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

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

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

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

13.3.2 Auxiliary Lemmas

lemma *concat_take_Suc_empty: "⟦ n < length l; l!n=[] ⟧*
```
  ⟹ concat (take (Suc n) l) = concat (take n l)"
```
⟨*proof*⟩

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

lemma *is_hashmap'_distinct:*
```
  "is_hashtable l ht
    ⟹$_A$ is_hashtable l ht * ↑(distinct (map fst (concat l)))"
```
⟨*proof*⟩

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

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

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

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

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

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

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

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

13.3.3 Main Lemmas

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

```
    <λ((k,v),it'). hm_is_it m ht (m' |` (-{k})) it' * ↑(m' k = Some v)>"
```
⟨*proof*⟩

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>$_t$"*
  ⟨*proof*⟩

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)>"*
  ⟨*proof*⟩

lemma *hm_it_finish: "hm_is_it m p m' it ⟹$_A$ is_hashmap m p"*
  ⟨*proof*⟩

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"*
  ⟨*proof*⟩
interpretation *hm: imp_map is_hashmap* ⟨*proof*⟩

lemma *hm_empty_impl: "imp_map_empty is_hashmap hm_new"*
  ⟨*proof*⟩
interpretation *hm: imp_map_empty is_hashmap hm_new* ⟨*proof*⟩

lemma *hm_lookup_impl: "imp_map_lookup is_hashmap hm_lookup"*
  ⟨*proof*⟩
interpretation *hm: imp_map_lookup is_hashmap hm_lookup* ⟨*proof*⟩

lemma *hm_update_impl: "imp_map_update is_hashmap hm_update"*
  ⟨*proof*⟩
interpretation *hm: imp_map_update is_hashmap hm_update* ⟨*proof*⟩

lemma *hm_delete_impl: "imp_map_delete is_hashmap hm_delete"*
  ⟨*proof*⟩
interpretation *hm: imp_map_delete is_hashmap hm_delete* ⟨*proof*⟩

lemma *hm_is_empty_impl: "imp_map_is_empty is_hashmap hm_isEmpty"*
  ⟨*proof*⟩
interpretation *hm: imp_map_is_empty is_hashmap hm_isEmpty*
  ⟨*proof*⟩

lemma *hm_size_impl: "imp_map_size is_hashmap hm_size"*
⟨*proof*⟩
interpretation *hm: imp_map_size is_hashmap hm_size* ⟨*proof*⟩

lemma *hm_iterate_impl:*
"imp_map_iterate is_hashmap hm_is_it hm_it_init hm_it_has_next hm_it_next"
⟨*proof*⟩
interpretation *hm:*
imp_map_iterate is_hashmap hm_is_it hm_it_init hm_it_has_next hm_it_next
⟨*proof*⟩

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

end

15 Interface for Sets

theory *Imp_Set_Spec*
imports *"../Sep_Main"*
begin

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

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

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

locale *imp_set_is_empty = imp_set +*
constrains *is_set :: "'a set ⇒ 's ⇒ assn"*
fixes *is_empty :: "'s ⇒ bool Heap"*
assumes *is_empty_rule[sep_heap_rules]:*
*"<is_set s p> is_empty p <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*⟩

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*⟩

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"*
  ⟨*proof*⟩

lemma *map_of_set_simps[simp]:*
  "dom (map_of_set s) = s"
  "map_of_set (dom m) = m"
  "map_of_set {} = Map.empty"
  "map_of_set s x = None ⟷ 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})"
  ⟨*proof*⟩

lemma *map_of_set_eq':*
  "map_of_set a = map_of_set b ⟷ a=b"
  ⟨*proof*⟩

lemma *map_of_set_eq[simp]:*
  "map_of_set s = m ⟷ dom m=s"
  ⟨*proof*⟩

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"*
  ⟨*proof*⟩
interpretation *hs: imp_set is_hashset* ⟨*proof*⟩

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

lemma *hs_new_impl: "imp_set_empty is_hashset hs_new"*
  ⟨*proof*⟩
interpretation *hs: imp_set_empty is_hashset hs_new* ⟨*proof*⟩

definition *hs_memb:: "'a::{heap,hashable} ⇒ '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"*
  ⟨*proof*⟩

interpretation *hs: imp_set_memb is_hashset hs_memb* ⟨*proof*⟩

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

lemma *hs_ins_impl: "imp_set_ins is_hashset hs_ins"*
  ⟨*proof*⟩
interpretation *hs: imp_set_ins is_hashset hs_ins* ⟨*proof*⟩

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

lemma *hs_delete_impl: "imp_set_delete is_hashset hs_delete"*
  ⟨*proof*⟩
interpretation *hs: imp_set_delete is_hashset hs_delete*
  ⟨*proof*⟩

definition *"hs_isEmpty == hm_isEmpty"*

lemma *hs_is_empty_impl: "imp_set_is_empty is_hashset hs_isEmpty"*
  ⟨*proof*⟩
interpretation *hs: imp_set_is_empty is_hashset hs_isEmpty*
  ⟨*proof*⟩

definition *"hs_size == hm_size"*

lemma *hs_size_impl: "imp_set_size is_hashset hs_size"*
  ⟨*proof*⟩
interpretation *hs: imp_set_size is_hashset hs_size* ⟨*proof*⟩

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

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

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

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

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

lemma *hs_iterate_impl: "imp_set_iterate*
is_hashset hs_is_it hs_it_init hs_it_has_next hs_it_next"
⟨*proof*⟩

interpretation *hs: imp_set_iterate*
is_hashset hs_is_it hs_it_init hs_it_has_next hs_it_next
⟨*proof*⟩

definition *"hs_union*
≡ union_loop_ins hs_it_init hs_it_has_next hs_it_next hs_ins"

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

lemma *hs_union_impl: "imp_set_union*
is_hashset hs_is_it hs_it_init hs_it_has_next hs_it_next hs_union"
⟨*proof*⟩

interpretation *hs: imp_set_union*
is_hashset hs_is_it hs_it_init hs_it_has_next hs_it_next hs_union
⟨*proof*⟩

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

end

17 Generic Algorithm to Convert Sets to Lists

theory *To_List_GA*
imports *Imp_Set_Spec Imp_List_Spec Hash_Set_Impl Open_List*
begin

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

17.1 Algorithm

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

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

lemma *to_list_ga_rec_rule:*
```
  assumes "imp_set_iterate is_set is_it it_init it_has_next it_next"
  assumes "imp_list_prepend is_list l_prepend"
  assumes FIN: "finite it"
  shows "
  < is_it s si it iti * is_list l li >
    to_list_ga_rec it_has_next it_next l_prepend iti li
  < λr. ∃_A l'. is_set s si
    * is_list l' r
    * ↑(set l' = set l ∪ it) >_t"
```
⟨*proof*⟩

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

lemma *to_list_ga_rule:*
```
  assumes IT: "imp_set_iterate is_set is_it it_init it_has_next it_next"
  assumes EM: "imp_list_empty is_list l_empty"
  assumes PREP: "imp_list_prepend is_list l_prepend"
  assumes FIN: "finite s"
  shows "
  <is_set s si>
  to_list_ga it_init it_has_next it_next
    l_empty l_prepend si
  <λr. ∃_A l. is_set s si * is_list l r * true * ↑(set l = s)>"
```
⟨*proof*⟩

17.2 Sample Instantiation for hash set and open list

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

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

export_code *hs_to_ol* **checking** *SML_imp*

end

18 Union-Find Data-Structure

theory *Union_Find*
imports
"../Sep_Main"
Collections.Partial_Equivalence_Relation
"HOL-Library.Code_Target_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))"*
⟨*proof*⟩

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

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

lemma *ufa_invarD:*
"⟦ufa_invar l; i<length l⟧ ⟹ rep_of_dom (l,i)"
"⟦ufa_invar l; i<length l⟧ ⟹ l!i<length l"
⟨*proof*⟩

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

lemma *rep_of_refl: "l!i=i ⟹ rep_of l i = i"*
⟨*proof*⟩

lemma *rep_of_step:*
"⟦ufa_invar l; i<length l; l!i≠i⟧ ⟹ rep_of l i = rep_of l (l!i)"
⟨*proof*⟩

lemmas *rep_of_simps = rep_of_refl rep_of_step*

lemma *rep_of_iff: "⟦ufa_invar l; i<length l⟧*
⟹ rep_of l i = (if l!i=i then i else rep_of l (l!i))"
⟨*proof*⟩

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

lemma *rep_of_induct[case_names base step, consumes 2]:*
assumes *I: "ufa_invar l"*
assumes *L: "i<length l"*
assumes *BASE: "⋀i. ⟦ ufa_invar l; i<length l; l!i=i ⟧ ⟹ P l i"*
assumes *STEP: "⋀i. ⟦ ufa_invar l; i<length l; l!i≠i; P l (l!i) ⟧*
⟹ P l i"
shows *"P l i"*
⟨*proof*⟩

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

lemma *rep_of_min:*
"⟦ ufa_invar l; i<length l ⟧ ⟹ l!(rep_of l i) = rep_of l i"
⟨*proof*⟩

lemma *rep_of_bound:*
"⟦ ufa_invar l; i<length l ⟧ ⟹ rep_of l i < length l"
⟨*proof*⟩

lemma *rep_of_idem:*
"⟦ ufa_invar l; i<length l ⟧ ⟹ rep_of l (rep_of l i) = rep_of l i"
⟨*proof*⟩

lemma *rep_of_min_upd: "⟦ ufa_invar l; x<length l; i<length l ⟧ ⟹*
rep_of (l[rep_of l x := rep_of l x]) i = rep_of l i"
⟨*proof*⟩

lemma *rep_of_idx:*
"⟦ufa_invar l; i<length l⟧ ⟹ rep_of l (l!i) = rep_of l i"
⟨*proof*⟩

18.1.2 Abstraction to Partial Equivalence Relation

definition *ufa_α :: "nat list ⇒ (nat×nat) set"*
  where *"ufa_α l*
    ≡ {(x,y). x<length l ∧ y<length l ∧ rep_of l x = rep_of l y}"

lemma *ufa_α_equiv[simp, intro!]: "part_equiv (ufa_α l)"*
  ⟨*proof*⟩

lemma *ufa_α_lenD:*
  "(x,y)∈ufa_α l ⟹ x<length l"
  "(x,y)∈ufa_α l ⟹ y<length l"
  ⟨*proof*⟩

lemma *ufa_α_dom[simp]: "Domain (ufa_α l) = {0..<length l}"*
  ⟨*proof*⟩

lemma *ufa_α_refl[simp]: "(i,i)∈ufa_α l ⟷ i<length l"*
  ⟨*proof*⟩

lemma *ufa_α_len_eq:*
  assumes *"ufa_α l = ufa_α l'"*
  shows *"length l = length l'"*
  ⟨*proof*⟩

18.1.3 Operations

lemma *ufa_init_invar: "ufa_invar [0..<n]"*
  ⟨*proof*⟩

lemma *ufa_init_correct: "ufa_α [0..<n] = {(x,x) | x. x<n}"*
  ⟨*proof*⟩

lemma *ufa_find_correct: "⟦ufa_invar l; x<length l; y<length l⟧*
  ⟹ rep_of l x = rep_of l y ⟷ (x,y)∈ufa_α l"
  ⟨*proof*⟩

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

lemma *ufa_union_invar:*
  assumes *I: "ufa_invar l"*
  assumes *L: "x<length l" "y<length l"*
  shows *"ufa_invar (ufa_union l x y)"*
  ⟨*proof*⟩

lemma *ufa_union_aux:*
  assumes *I: "ufa_invar l"*
  assumes *L: "x<length l" "y<length l"*
  assumes *IL: "i<length l"*
  shows *"rep_of (ufa_union l x y) i =*

```
    (if rep_of l i = rep_of l x then rep_of l y else rep_of l i)"
```
⟨*proof*⟩

lemma *ufa_union_correct:*
```
"⟦ ufa_invar l; x<length l; y<length l ⟧
  ⟹ ufa_α (ufa_union l x y) = per_union (ufa_α l) x y"
```
⟨*proof*⟩

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

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

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

18.2 Implementation with Imperative/HOL

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

type_synonym *uf =*
```
"nat array × nat array"
```

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

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

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

partial_function *(heap) uf_rep_of :: "nat array ⇒ 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↦$_a$l> uf_rep_of p i <λr. p↦$_a$l * ↑(r=rep_of l i)>"*
⟨*proof*⟩

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

partial_function *(heap) uf_compress :: "nat ⇒ 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↦$_a$l> uf_compress i ci 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))>"
⟨*proof*⟩

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↦$_a$l> uf_rep_of_c p i <λr. ∃$_A$l'. p↦$_a$l'
** ↑(r=rep_of l i ∧ ufa_invar l'*
∧ length l' = length l
∧ (∀i<length l. rep_of l' i = rep_of l i))>"
⟨*proof*⟩

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')"
```

⟨*proof*⟩

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)>"
```

⟨*proof*⟩

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

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

lemma *uf_union_rule[sep_heap_rules]: "⟦i∈Domain R; j∈ Domain R⟧*

```
⟹ <is_uf R u> uf_union u i j <is_uf (per_union R i j)>"
```

⟨*proof*⟩

export_code *uf_init uf_cmp uf_union* **checking** *SML_imp*

export_code *uf_init uf_cmp uf_union* **checking** *Scala_imp*

end

19 Common Proof Methods and Idioms

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

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

19.1 The Method *sep_auto*

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

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

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

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

```
lemma "<emp>
  test
  <λ(l1,l2). cs_list [''a'',''b''] l1
    * cs_list [''c'',''e'',''d''] l2>t"
  ⟨proof⟩
```

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

As a typical example, consider the following proof:

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

19.2 Applying Single Rules

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

lemma
"<os_list xs n> os_prepend x n <os_list (x # xs)>"
⟨*proof*⟩

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

lemma
"<os_list xs n> os_prepend x n <os_list (x # xs)>"
⟨*proof*⟩

Entailment This example presents an actual proof from the circular list theory, where we have to manually apply a rule and give some hints to frame inference

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

19.3 Functions with Explicit Recursion

If the termination argument of a function depends on one of its parameters, we can use the function package. For example, the following function inserts elements from a list into a hash-set:

fun *ins_from_list*
:: "('x::{heap,hashable}) list ⇒ '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*⟩

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*⟩

19.5 Precision Proofs

Precision lemmas show that an assertion uniquely determines some of its parameters. Our example shows that two list segments from the same start pointer and with the same list, also have to end at the same end pointer.

lemma *lseg_prec3:*
*"∀ q q'. h ⊨ (lseg l p q * F1) ∧$_A$ (lseg l p q' * F2) ⟶ q=q'"*
⟨*proof*⟩
end

20 Conclusion

We have presented a separation logic framework for Imperative HOL. It provides powerful proof methods for reasoning over imperative monadic programs, thus rectifying the lack of good proof support in the original Imperative HOL formalization.

We verified the applicability of our framework by proving algorithms on various data structures. Moreover, we showed how to construct an imperative collection framework, that supports generic algorithms and data refinement.

Acknowledgments We thank Thomas Tuerk, the author of Holfoot [8], for useful discussions on the automation of separation logic. Moreover, we thank Lukas Bulwahn and Brian Huffman for help with the Isabelle ML interface.

References

[1] J. Bengtson, J. B. Jensen, F. Sieczkowski, and L. Birkedal. Verifying object-oriented programs with higher-order separation logic in coq. In M. C. J. D. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *ITP*, volume 6898 of *Lecture Notes in Computer Science*, pages 22–38. Springer, 2011.

[2] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.

[3] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews. Imperative functional programming with isabelle/hol. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 134–149. Springer, 2008.

[4] G. Klein, R. Kolanski, and A. Boyton. Separation algebra. *Archive of Formal Proofs*, 2012, 2012.

[5] P. Lammich and A. Lochbihler. The isabelle collections framework. In M. Kaufmann and L. C. Paulson, editors, *ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2010.

[6] R. Meis. Integration von Separation Logic in das Imperative HOL-Framework. Diplomarbeit, University of Münster, April 2011.

[7] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *CSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[8] T. Tuerk. A separation logic framework for HOL. Technical Report UCAM-CL-TR-799, University of Cambridge, Computer Laboratory, June 2011.