Formal Verification of the Heap Manager of an Operating System using Separation Logic*

Nicolas Marti[1], Reynald Affeldt[2], and Akinori Yonezawa[1,2]

[1] Department of Computer Science, University of Tokyo
[2] Research Center for Information Security,
National Institute of Advanced Industrial Science and Technology

Abstract. In order to ensure memory properties of an operating system, it is important to verify the implementation of its heap manager. In the case of an existing operating system, this is a difficult task because the heap manager is usually written in a low-level language that makes use of pointers, and it is usually not written with verification in mind. In this paper, our main contribution is the formal verification of the heap manager of an existing embedded operating system, namely Topsy. For this purpose, we develop in the Coq proof assistant a library for separation logic, an extension of Hoare logic to deal with pointers. Using this library, we were able to verify the C source code of the Topsy heap manager, and to find and correct bugs.

1 Introduction

In order to ensure memory properties of an operating system, it is important to verify the implementation of its heap manager. The heap manager is the set of functions that provides the operating system with dynamic memory allocation. Incorrect implementation of these functions can invalidate essential memory properties. For example, task isolation, the property that user processes cannot tamper with the memory of kernel processes, is such a property: the relation with dynamic memory allocation comes from the fact that privilege levels of processes are usually stored in dynamically allocated memory blocks (see [5] for a detailed illustration).

However, the verification of the heap manager of an existing operating system is a difficult task because it is usually written in a low-level language that makes use of pointers, and it is usually not written with verification in mind. For these reasons, the verification of dynamic memory allocation is sometimes considered as a challenge for mechanical verification [15].

In this paper, our main contribution is to formally verify the heap manager of an existing embedded operating system, namely Topsy [2]. For this purpose, we develop in the Coq proof assistant [4] a library for separation logic [1], an extension of Hoare logic to deal with pointers. Using this library, we verify the

* This is a version with appendices of a paper published in ICFEM 2006 (`http://www.iist.unu.edu/icfem06`).

C source code of the Topsy heap manager. In fact, this heap manager proves harder to deal with than dynamic memory allocation facilities verified in previous studies (see Sect. 8 for a comparison). A direct side-effect of our approach is to provide advanced debugging. Indeed, our verification highlights several issues and bugs in the original source code (see Sect. 7.1 for a discussion).

We chose the Topsy operating system as a test-bed for formal verification of memory properties. Topsy was initially created for educational use and has recently evolved into an embedded operating system for network cards [3]. It is well-suited for mechanical verification because it is small and simple, yet it is a realistic use-case because it includes most classical features of operating systems.

The paper is organized as follows. In Sect. 2, we give an overview of the Topsy heap manager, and we explain our verification goal and approach. In Sect. 3, we introduce separation logic and explain how we encode it in Coq. In Sect. 4, we formally specify and prove the properties of the underlying data structure used by the heap manager. In Sect. 5, we formally specify and explain the verification of the functions of the heap manager. In Sect. 6, we discuss practical aspects of the verification such as automation and translation from the original C source code. In Sect. 7, we discuss the outputs of our experiment: in particular, issues and bugs found in the original source code of the heap manager. In Sect. 8, we comment on related work. In Sect. 9, we conclude and comment on future work.

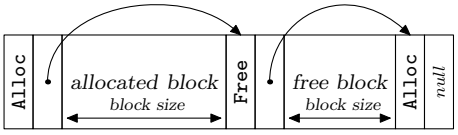
2 Verification Goal and Approach

2.1 Topsy Heap Manager

The heap manager of an operating system is the set of functions that provides dynamic memory allocation. In Topsy, these functions and related variables are defined in the files `Memory/MMHeapMemory.{h,c}`, with some macros in the file `Topsy/Configuration.h`. We are dealing here with the heap manager of Topsy version 2; a browsable source code is available online [2].

The *heap* is the area of memory reserved by Topsy for the heap manager. The latter divides the heap into allocated and free memory blocks: allocated blocks are memory blocks in use by programs, and free blocks form a pool of memory available for new allocations. In order to make an optimal use of the memory, allocated and free memory blocks form a partition of the heap. This is achieved by implementing memory blocks as a simply-linked list of contiguous blocks. In the following, we refer to this data structure as a *heap-list*.

In a heap-list, each block consists of a two-fields header and an array of memory. The first field of the header gives information on the status of the block (allocated or free, corresponding to the `Alloc` and `Free` flags); the second field is a pointer to the next block, which starts just after the current block. For example, here is a heap-list with one allocated block and one free block:



Observe that the size of the arrays of memory associated to blocks can be computed using the values of pointers. (In this paper, when we talk about the size of a block, we talk about its "effective" size, that is the size of the array of memory associated to it, this excludes the header.) The terminal block of the heap-list always consists of a sole header, marked as allocated, and pointing to null.

Initialization of the heap manager is provided by the following function:

```
Error hmInit(Address addr) {...}
```

Concretely, `hmInit` initializes the heap-list by building a heap-list with a single free block that spans the whole heap. The argument is the starting location of the heap. The size of the heap-list is defined by the macro `KERNELHEAPSIZE`. The function always returns `HM_INITOK`.

Allocation is provided by the following function:

```
Error hmAlloc(Address* addressPtr, unsigned long int size) {...}
```

The role of `hmAlloc` is to insert new blocks marked as allocated into the heap-list. The first argument is a pointer provided by the user to get back the address of the allocated block, the second argument is the desired size. In case of successful allocation, the pointer contains the address of the newly allocated block and the value `HM_ALLOCOK` is returned, otherwise the value `HM_ALLOCFAILED` is returned. In order to limit fragmentation, `hmAlloc` performs compaction of contiguous free blocks and splitting of free blocks.

Deallocation is provided by the following function:

```
Error hmFree(Address address) {...}
```

Concretely, `hmFree` turns allocated blocks into free ones. The argument corresponds to the address of the allocated block to free. The function returns `HM_FREEOK` if the block was successfully deallocated, or `HM_FREEFAILED` otherwise.

2.2 Verification Goal and Approach

Our goal is to verify that the implementation of the Topsy heap manager is "correct". By correct, we mean that the heap manager provides the intended service: the allocation function allocates large-enough memory blocks, these memory blocks are "fresh" (they do not overlap with previously allocated memory blocks), the deallocation function turns the status of blocks into free (except for the terminal block), and the allocation and deallocation functions does not behave in unexpected ways (in particular, they do not modify neither previously allocated memory blocks nor the rest of the memory). Guaranteeing the allocation of fresh memory blocks and the non-modification of previously allocated memory blocks is a necessary condition to ensure that the heap manager preserves exclusive usage of allocated blocks. Formal specification goals corresponding to the above informal discussion are explained later in Sect. 5.

Our approach is to use separation logic to formally specify and mechanically verify the goal informally stated above. We choose separation logic for this purpose because it provides a native notion of pointer and memory separation that

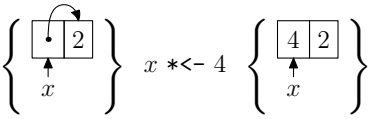
facilitates the specification of heap-lists. Another advantage of separation logic is that it is close enough to the C language to enable systematic translation from the original source code of Topsy.

In the next sections, we explain how we encode separation logic in the Coq proof assistant and how we use this encoding to specify and verify the Topsy heap manager. All the verification is available online [6].

3 Encoding of Separation Logic

Separation logic is an extension of Hoare logic to reason about low-level programs with shared, mutable data structures [1]. Before entering the details of the formal encoding, we introduce the basic ideas behind separation logic.

Brief Introduction to Separation Logic Let us consider the program x `*<-` 4 that puts the value 4 into a memory cell pointed to by the variable x. Let us assume that this cell originally contained a pointer to a contiguous cell with the value 2. Informally, the corresponding Hoare triple could be written as follows:



Separation logic provides connectives to conveniently specify and reason about such Hoare triples. In particular, it extends the language of assertions of Hoare logic with a *separating conjunction* $\star$ that asserts that its subformulas hold for disjoint parts of the memory. For illustration, the pre/post-conditions above would be respectively written $(x \mapsto p) \star (p \mapsto 2)$ and $(x \mapsto 4) \star (p \mapsto 2)$, where p is the *location* held by variable x. Separation logic also provides us with "axioms" to verify such triples. For example, by applying the "axiom of backward reasoning for mutation" (to be defined formally later in this section), the verification is reduced to the proof of the (classical) implication $(x \mapsto p) \star (p \mapsto 2) \rightarrow (x \mapsto p) \star ((x \mapsto 4) \mathrel{-\!\!\star} ((x \mapsto 4) \star (p \mapsto 2)))$ where $\mathrel{-\!\!\star}$ is the *separating implication*; this formula is easily provable using the properties of separation logic.

In the rest of this section, we explain the formal definition of separation logic that we implemented in Coq to perform such reasoning as above. The code displayed is directly taken from the implementation; we use traditional mathematical notations instead of ASCII for Coq primitives (e.g., $\forall$, $\exists$, $\rightarrow$, $\wedge$, $\neq$, $\geq$ instead of `forall`, `exists`, `->`, `/\`, `<>`, `>=`).

3.1 The Programming Language

The programming language of separation logic is imperative. The current state of execution is represented by a pair of a store (that maps local variables to values) and a heap (a finite map from locations to values). We have an abstract type `var.v` for variables (ranged over by `x`, `y`), a type `loc` for locations (ranged

over by `p`, `adr`), and a type `val` for values (ranged over by `v`, `w`) with the condition that all values can be seen as locations (so as to enable pointer arithmetic). Our implementation is essentially abstracted over the choice of types, yet, in our experiments, we have taken the native Coq types of naturals `nat` and relative integers `Z` for `loc` and `val` so as to benefit from better automation. Stores and heaps are implemented by two modules `store` and `heap` whose types are (excerpts):

```
Module Type STORE.
 Parameter s : Set. (* the abstract type of stores *)
 Parameter lookup : var.v → s → val.
 Parameter update : var.v → val → s → s.
End STORE.

Module Type HEAP.
 Parameter l : Set. (* locations *)
 Parameter v : Set. (* values *)
 Parameter h : Set. (* the abstract type of heaps *)
 Parameter emp : h. (* the empty heap *)
 Parameter singleton : l → v → h. (* singleton heaps *)
 Parameter lookup : l → h → option v.
 Parameter update : l → v → h → h.
 Parameter union : h → h → h.        Notation "h1 ⊎ h2" := (union h1 h2).
 Parameter disjoint : h → h → Prop.  Notation "h1 ⊥ h2" := (disjoint h1 h2).
End HEAP.

Definition state := prod store.s heap.h.
```

To paraphrase the implementation, `(store.lookup x s)` is the value of the variable `x` in store `s`; `(store.update x v s)` is the store `s` in which the variable `x` has been updated with the value `v`; `(heap.lookup p h)` is the contents (if any) of location `p`; `(heap.update p v h)` is the heap `h` in which the location `p` has been mutated with the value `v`; `h ⊎ h'` is the disjoint union of `h` and `h'`; and `h ⊥ h'` holds when `h` and `h'` have disjoint domains.

The programming language of separation logic manipulates arithmetic and boolean expressions that are evaluated w.r.t. the store. They are encoded by the inductive types `expr` and `expr_b` (the parts of the definitions which are not essential to the understanding of this paper are abbreviated with "..."):

```
Inductive expr : Set :=
  var_e : var.v → expr
| int_e : val → expr
| add_e : expr → expr → expr         Notation "e1 '+e' e2" := (add_e e1 e2).
...
Definition null := int_e 0%Z.
Definition nat_e x := int_e (Z_of_nat x).
Definition field x f := var_e x +e int_e f.
                                     Notation "x '-.>' f " := (field x f).
Inductive expr_b : Set :=
  eq_b : expr → expr → expr_b        Notation "e == e'" := (eq_b e e').
| neq_b : expr → expr → expr_b       Notation "e =/= e'" := (neq_b e e').
| and_b : expr_b → expr_b → expr_b   Notation "e &&& e'" := (and_b e e').
| gt_b : expr → expr → expr_b        Notation "e >> e'" := (gt_b e e').
...
```

There is an evaluation function `eval` such that `(eval e s)` is the result of evaluating the expression `e` w.r.t. the store `s`.

The commands of the programming language of separation logic are also encoded by an inductive type:

```
Inductive cmd : Set :=
  assign : var.v → expr → cmd          Notation "x <- e" := (assign x e).
| lookup : var.v → expr → cmd          Notation "x '<-*' e" := (lookup x e).
| mutation : expr → expr → cmd         Notation "e '*<-' f" := (mutation e f).
| seq : cmd → cmd → cmd                Notation "c ; d" := (seq c d).
| while : expr_b → cmd → cmd
| ifte : expr_b → cmd → cmd → cmd     Notation "'ifte' b 'thendo' c 'elsedo' c"
...                                      := (ifte b c d).
```

From this presentation, we omit the memory allocation and deallocation commands of separation logic (they are not useful for our use-case precisely because we verify the implementation of a memory allocation facility).

The operational semantics of the programming language of separation logic is defined by the following inductive type. An object of type `(exec s c s')` represents the execution of the command `c` from state `s` to state `s'`. Because heaps are finite maps, lookup and mutation may fail; to take this possibility into account, we use an option type.

```
Inductive exec : option state → cmd → option state → Prop :=
    exec_assign : ∀ s h x e,
     exec (Some (s, h)) (x <- e) (Some (store.update x (eval e s) s, h))
  | exec_lookup : ∀ s h x e p v,
     val2loc (eval e s) = p → heap.lookup p h = Some v →
     exec (Some (s, h)) (x <-* e) (Some (store.update x v s, h))
  | exec_lookup_err : ∀ s h x e p,
     val2loc (eval e s) = p → heap.lookup p h = None →
     exec (Some (s, h)) (x <-* e) None
  | exec_mutation : ∀ s h e e' p v,
     val2loc (eval e s) = p → heap.lookup p h = Some v →
     exec (Some (s, h)) (e *<- e') (Some (s, heap.update p (eval e' s) h))
  | exec_mutation_err : ∀ s h e e' p,
     val2loc (eval e s) = p → heap.lookup p h = None →
     exec (Some (s, h)) (e *<- e') None
  ...
```

3.2 Assertions and Reynolds' Axioms

Assertions of Hoare logic are predicate calculus formulas with the same expressions as the programming language. In consequence, the validity of an assertion depends on the current execution state of the program. There are mainly two ways to encode the semantics of such assertions in a proof assistant:

1. *Deep encoding*: define a syntax for assertions and a satisfaction relation between states and assertions.
2. *Shallow encoding*: identify formulas with functions from states to some "boolean type".

The advantage of shallow encoding over deep encoding is that deciding the validity of formulas becomes a function computation, for which the proof assistant provides native facilities (for example, tactics to prove tautologies).

We have developed a shallow encoding of separation logic in Coq. For this purpose, we identify assertions of separation logic with functions from states to `Prop`, the native type for predicate calculus formulas. For example, `True:Prop` represents truth and `∧:Prop → Prop → Prop` represents classical conjunction in Coq. This gives rise to the type `assert` below. By way of example, we also show the encoding of truth and conjunction in separation logic.

```
Definition assert := store.s → heap.h → Prop.
Definition TT : assert := fun s h => True.
Definition And (P Q:assert) : assert := fun s h => P s h ∧ Q s h.
```

Assertions of Separation Logic The assertion that holds for empty heaps is defined by testing whether the heap is empty:

```
Definition emp : assert := fun s h => h = heap.emp.
```

`e ↦ e'` is the formula that holds for a singleton heap whose only location is the result of evaluating `e` and this location has for contents the result of evaluating `e'`:

```
Definition mapsto e e' s h := ∃ p,
  val2loc (eval e s) = p ∧ h = heap.singleton p (eval e' s).
Notation "e1 ↦ e2" := (mapsto e1 e2).
```

For example, `(var_e x ↦ int_e 4)` asserts that the variable `x` points to a cell that contains the integer 4. The following derived definitions will prove useful later: `e ↦ _` asserts that the cell `e` has some undefined contents, and `e ⤇ l` asserts that there is a list `l` of contiguous cell contents starting from `e`.

The separating conjunction `P ⋆ Q` holds for a heap that can be decomposed into two disjoint heaps for which `P` and `Q` respectively hold:

```
Definition con (P Q:assert) : assert := fun s h =>
  ∃ h1, ∃ h2, h1 ⊥ h2 ∧ h = h1 ⊎ h2 ∧ P s h1 ∧ Q s h2.
Notation "P ⋆ Q" := (con P Q).
```

For example, `(var_e x ↦ nat_e p) ⋆ (nat_e p ↦ int_e 2)` is the formal version of the example given in the beginning of this section.

The separating implication `P -⋆ Q` is less intuitive. It is used to represent logically mutations. In particular, the idiom `(e ↦ _ ⋆ (e ↦ e' -⋆ P))` holds for a heap such that the mutation of location `e` to contents `e'` leads to a heap that satisfies `P`. Section 4.2 gives a concrete example of such a formula together with its utilization. For the time being, we limit ourselves to the formal definition:

```
Definition imp (P Q:assert) : assert := fun s h =>
  ∀ h', h ⊥ h' ∧ P s h' → ∀ h'', h'' = h ⊎ h' → Q s h''.
Notation "P -⋆ Q" := (imp P Q).
```

Reynolds' Axioms The axioms of separation logic are defined by the following inductive type. An object of type `(semax P c Q)` represents the fact that, going from a state satisfying `P`, the execution of the command `c` leads to a state satisfying `Q`:

```
Inductive semax : assert → cmd → assert → Prop :=
    semax_assign : ∀ P x e,
     semax (update_store2 x e P) (x <- e) P
  | semax_lookup : ∀ P x e,
     semax (lookup2 x e P) (x <-* e) P
  | semax_mutation : ∀ P e e',
     semax (update_heap2 e e' P) (e *<- e') P
  | semax_seq : ∀ P Q R c d,
     semax P c Q → semax Q d R → semax P (c ; d) R
  ...
Notation "{{ P }} c {{ Q }}" := (semax P c Q).
```

where `update_store2`, etc. are predicate transformers, for example:

```
Definition update_store2 (x:var.v) (e:expr) (P:assert) : assert :=
  fun s h => P (store.update x (eval e s) s) h.
```

Using these definitions, we have implemented much of [1], including in particular the proof of soundness of the axioms of separation logic, the "frame rule", various axioms for backward reasoning, etc. For example, let us just give the axiom for backward reasoning used in the example at the beginning of this section:

```
Lemma semax_mutation_backwards : ∀ P e e',
  {{ fun s h => ∃ e'', (e ↦ e'' ⋆ (e ↦ e' -⋆ P)) s h }} e *<- e' {{ P }}.
```

4 The Heap-list Data Structure

4.1 The Heap-list Assertion

We define an assertion called `Heap_List` that holds for heaps that contain a well-formed heap-list. Separation logic is very convenient for this purpose. In particular, the property that blocks are disjoint can be expressed using the separating conjunction. The fact the blocks are contiguous relies on pointer arithmetic and this can also be expressed directly in separation logic.

Before defining the `Heap_List` assertion, we define an assertion to represent arrays of memory, i.e. sets of contiguous locations. `Array p sz` holds for a heap whose locations `p`, ..., `p+sz-1` have some contents:

```
Fixpoint Array (p:loc) (size:nat) {struct size} : assert :=
  match size with
    0 => emp
    | S n => (fun s h => ∃ y, (nat_e p ↦ int_e y) s h) ⋆ Array (p+1) n
  end.
```

We now come to the definition of heap-lists *without* terminal block (let us call them *pre-heap-lists* for convenience). Intuitively, `(hl p l)` represents the set of headers of a pre-heap-list whose first block starts at location `p` together with

the set of free blocks (the allocated blocks are left outside); information about the blocks is captured by the parameter `(l:list (nat*bool))`: the list of sizes and flags of the blocks (:: is the list constructor and `nil` is the empty list):

```
Inductive hl : loc → list (nat*bool) → assert :=
  | hl_last: ∀ s p h,
     emp s h → hl p nil s h
  | hl_Free: ∀ s h p h1 h2 size tl,
     h1 ⊥ h2 → h = h1 ⊎ h2 →
     ((nat_e p ↦ Free::nat_e (p+2+size)::nil) ⋆ (Array (p+2) size)) s h1 →
     hl (p+2+size) tl s h2 →
     hl p ((size,free)::tl) s h
  | hl_Allocated: ∀ s h p h1 h2 size tl,
     h1 ⊥ h2 → h = h1 ⊎ h2 →
     (nat_e p ↦ Allocated::nat_e (p+2+size)::nil) s h1 →
     hl (p+2+size) tl s h2 →
     hl p ((size,alloc)::tl) s h.
```

where `free` and `alloc` are synonymous for booleans. The first constructor specifies empty pre-heap-lists. The second constructor specifies pre-heap-lists that start with a free memory block (that is, a header marked as free and its associated block) followed by a pre-heap-list. The third constructor specifies pre-heap-lists that start with an allocated memory header (in this case, the associated block is left outside). Observe that the definition above uses pointer arithmetic to guarantee that there is no lost space between linked blocks.

Finally, we define heap-lists (*with* terminal block). This is simply the separating conjunction of a pre-heap-list with a terminal block (an allocated block pointing to null):

```
Definition Heap_List (l:list (nat*bool)) (p:nat) : assert :=
  (hl p l) ⋆ (nat_e (get_endl l p) ↦ Allocated::null::nil).
```

where `(get_endl l p)` returns the size of the domain covered by the list `l` from location `p`, i.e., the location of (the header of) the terminal block.

4.2 Properties of Heap-lists

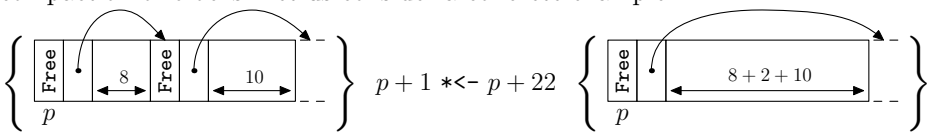
The heart of our verification of the Topsy heap manager consists of a few basic lemmas capturing the properties of operations such as compaction of blocks, splitting of a block, changing the status of blocks, etc. Since these operations rely on destructive updates, the properties in question are adequately expressed using the separating implication.

For example, the following lemma expresses compaction of two contiguous free blocks (++ is the list append function of Coq):

```
Lemma hl_compaction: ∀ l1 l2 size size' p s h,
  Heap_List (l1 ++ (size,free)::(size',free)::nil ++ l2) p s h →
  ∃ y, (nat_e (get_endl l1 p + 1) ↦ y ⋆
    (nat_e (get_endl l1 p + 1) ↦ nat_e (get_endl l1 p + size + size' + 4) -⋆
      Heap_List (l1 ++ (size+size'+2,free)::nil ++ l2) p)) s h.
```

The left-hand side of the (classical) implication states the existence of two contiguous free blocks `(size,free)` and `(size',free)`. The right-hand side represents the destructive update of the "next" field of the first block that is made to point to the block following the second block. As a result, the first block sees its size increased by the size of the second block. The function `get_endl` is used to compute the starting location of a block.

We can use this lemma to verify that a destructive update really performs compaction of blocks. Let us consider a concrete example:



In Coq, we input the following goal, that makes use of `Heap_List` assertions:

```
Goal ∀ p, {{ Heap_List ((8,free)::(10,free)::nil) p }}
           nat_e p +e int_e 1 *<- nat_e p +e int_e 22
          {{ Heap_List ((20,free)::nil) p }}.
```

The application of the axiom for backward reasoning (seen in Sect. 3.2) leads to:

```
p : nat
s : store.s
h : heap.h
H : Heap_List ((8, free) :: (10, free) :: nil) p s h
============================
 ∃ e'' : expr,
   ((nat_e p +e int_e 1) ↦ e'' ⋆
    ((nat_e p +e int_e 1) ↦ (nat_e p +e int_e 22) -⋆
     Heap_List ((20, free) :: nil) p)) s h
```

This new goal is precisely the conclusion of the lemma we gave above. Application of this lemma terminates the proof.

5 Formal Verification

For each function of the heap manager, we give formal specifications using Hoare triples written with the encoding of Sect. 3 and the assertions of Sect. 4. We explain in more details the verification of the allocation function, because it is the most involved. All proof sketches can be found in Appendix A.

Prior to verification, the C source code of each function is translated into the programming language of separation logic. As a result of this translation, the signature of each function is augmented with parameters to represent local variables and the return value. This explains the differences between the signatures given in this section and in Sect. 2.1. The translation is explained in Sect. 6.2.

5.1 Formal Verification of Initialization

The initialization function `hmInit` transforms a given area of raw memory into an initial heap-list that consists of a single free block. In the source code, this area

starts at location `hmStart` and has a fixed length `KERNELHEAPSIZE`. We formally verify `hmInit` for the general case of any starting location and any size greater than 4: the minimal space needed for two headers (the header of the free block and the header of the terminal block):

```
Definition hmInit_specif := ∀ p size, size ≥ 4 →
 {{ Array p size }} hmInit p size {{ Heap_List ((size-4,free)::nil) p }}.
```

The size of the array of memory corresponding to the free block is the size of the whole area of memory minus the size of the two headers. The verification of this triple is done almost automatically using a tactic provided by our Coq implementation. The non-automatic part is due to the translation of the assertions `Array` and `Heap_List` into the fragment of separation logic handled by this tactic. See Sect. 6.1 for more details.

Despite its apparent simplicity, this function turns out to be buggy, as we explain in Sect. 7.1.

5.2 Formal Verification of Allocation

The allocation function `hmAlloc` searches for a large-enough free block in the heap-list, possibly performing compaction of free blocks if needed. If an adequate block is found, it is split into an allocated block (whose location is returned) and a free block (available for further allocations); otherwise, an error is returned.

We introduce new assertions to simplify specifications. Under the hypothesis that `(Heap_List lst p0)` holds, the assertion `(In_hl lst (p,size,flag) p0)` means that the block starting at location `p` has size `size` and flag `flag`. The assertion `(s |= b)` holds when `b` is true in the store `s`.

As stated informally in Sect. 2.2, the specification of the allocation function consists in checking that (1) newly allocated blocks have at least the requested size, (2) they do not overlap with already allocated memory blocks (they are "fresh"), and (3) neither previously allocated memory blocks nor the rest of the memory is modified.

The formal specification of `hmAlloc` follows. In the pre-condition, we isolate some already allocated block `(x,sizex,alloc)`. In the post-condition, we ensure that (1) the newly allocated block `(y,size'',alloc)` has an appropriate size (i.e., greater than the requested `size`), (2) this newly allocated block does not overlap with previously allocated blocks (more precisely, the newly allocated block is built out of free blocks since `(Heap_List l adr ⋆ Array (y+2) size'')`, and it cannot be the previously allocated block `x` since `x ≠ y`), and (3) previously allocated memory blocks and the rest of the memory are not modified (because these areas are left outside of the area described by the `Heap_List` assertion). The second disjunction in the post-condition applies when allocation fails.

```
Definition hmAlloc_specif := ∀ adr x sizex size, adr > 0 → size > 0 →
  {{ fun s h => ∃ l, Heap_List l adr s h ∧ In_hl l (x,sizex,alloc) adr ∧
     (s |= var_e hmStart == nat_e adr) }}
   hmAlloc result size entry cptr fnd stts nptr sz
  {{ fun s h => (∃ l, ∃ y, y > 0 ∧ (s |= var_e result == nat_e (y+2)) ∧
```

```
 ∃ size'', size'' ≥ size ∧ (Heap_List l adr ⋆ Array (y+2) size'') s h ∧
 In_hl l (x,sizex,alloc) adr ∧ In_hl l (y,size'',alloc) adr ∧ x ≠ y)
∨
(∃ l, (s |= var_e result == nat_e 0) ∧
  Heap_List l adr s h ∧ In_hl l (x,sizex,alloc) adr) }}.
```

Other assertions are essentially technical. The equality about the variable `hmStart` and the location `adr` is necessary because the variable `hmStart` is actually global and written explicitly in the original C source code of the allocation function. The inequality about the location `adr` is necessary because the function implicitly assumes that there is no block starting at the `null` location. The inequality about the requested size is not necessary, it is just to emphasize that null-allocation is a special case (see Sect. 7.1 for a discussion).

The allocation function relies on three functions to do (heap-)list traversals, compaction of free blocks, and eventually splitting of free blocks. In the rest of this section, we briefly comment on the verification of these three functions.

Traversal The function `findFree` traverses the heap-list in order to find a large-enough free block. It takes as parameters the requested `size` and a return variable `entry` to be filled with the location of an appropriate block if any:

```
Definition findFree_specif := ∀ adr x sizex size, size > 0 → adr > 0 →
  {{ fun s h => ∃ l, Heap_List l adr s h ∧ In_hl l (x,sizex,alloc) adr ∧
     (s |= (var_e hmStart == nat_e adr) &&& (var_e result == null)) }}
   findFree size entry fnd sz stts
  {{ fun s h => ∃ l, Heap_List l adr s h ∧ In_hl l (x,sizex,alloc) adr ∧
     (s |= (var_e hmStart == nat_e adr) &&& (var_e result == null)) ∧
     ((∃ y, ∃ size'', size'' ≥ size ∧ In_hl l (y,size'',free) adr ∧
      (s |= (var_e entry == nat_e y) &&& (nat_e y >> null)))
      ∨
      s |= var_e entry == null) }}.
```

The post-condition asserts that the search succeeds and the return value corresponds to the starting location of a large-enough free block, or the search fails and the return value is `null`.

Compaction The function `compact` is invoked when traversal fails. Its role is to merge all the contiguous free blocks of the heap-list, so that a new traversal can take place and hopefully succeeds:

```
Definition compact_specif:= ∀ adr size sizex x, size > 0 → adr > 0 →
 {{ fun s h => ∃ l, Heap_List l adr s h ∧ In_hl l (x,sizex,alloc) adr ∧
    (s |= (var_e hmStart == nat_e adr) &&& (var_e result == null) &&&
           (var_e cptr == nat_e adr)) }}
  compact cptr nptr stts
 {{ fun s h => ∃ l, Heap_List l adr s h ∧ In_hl l (x,sizex,alloc) adr ∧
    (s |= (var_e hmStart == nat_e adr) &&& (var_e result == null)) }}.
```

The formal specification of `compact` asserts that it preserves the heap-list structure. Its verification is technically involved because it features two nested loops and therefore large invariants. The heart of this verification is the application of the compaction lemma already given in Sect. 4.2.

Splitting The function `split` splits the candidate free block into an allocated block of appropriate size and a new free block:

```
Definition split_specif := ∀ adr size sizex x, size > 0 → adr > 0 →
  {{ fun s h =>  ∃ l, Heap_List l adr s h ∧ In_hl l (x,sizex,alloc) adr ∧
     (s |= (var_e hmStart == nat_e adr) &&& (var_e result == null)) ∧
     (∃ y, ∃ size'', size'' ≥ size ∧ In_hl l (y,size'',free) adr ∧
       (s |= var_e entry == nat_e y) ∧ y > 0 ∧ y ≠ x) }}
   split entry size cptr sz
  {{ fun s h => ∃ l, In_hl l (x,sizex,alloc) adr ∧
     (∃ y, y > 0 ∧ (s |= var_e entry == int_e y) ∧
       (∃ size'', size'' ≥ size ∧
         (Heap_List l adr ⋆ Array (y+2) size'') s h ∧
         In_hl l (y,size'',alloc) adr ∧ y ≠ x)) }}.
```

The pre-condition asserts that there is a free block of size greater than `size` starting at the location pointed by `entry` (this is the block found by the previous list traversal). The post-condition asserts the existence of an allocated block of size greater than `size` (that is in general smaller than the original free block used to be).

5.3 Formal Verification of Deallocation

The deallocation function `hmFree` does a list traversal; if it runs into the location passed to it, it frees the corresponding block, and fails otherwise. Besides the fact that an allocated block becomes free, we must also ensure that `hmFree` does not modify previously allocated blocks nor the rest of the memory; here again, this is taken into account by the definition of `Heap_List`:

```
Definition hmFree_specif := ∀ p x sizex y sizey statusy, p > 0 →
  {{ fun s h => ∃ l, (Heap_List l p ⋆ Array (x+2) sizex) s h ∧
     In_hl l (x,sizex,alloc) p ∧ In_hl l (y,sizey,statusy) p ∧
     x ≠ y ∧ s |= var_e hmStart == nat_e p }}
   hmFree (x+2) entry cptr nptr result
  {{ fun s h => ∃ l, Heap_List l p s h ∧
     In_hl l (x,sizex,free) p ∧ In_hl l (y,sizey,statusy) p ∧
     s |= var_e result == HM_FREEOK }}.
```

The main difficulty of this verification was to identify a bug that allows for deallocation of the terminal block, as we explain in Sect 7.1.

6 Practical Aspects of the Implementation

6.1 About Automation

Since our specifications take into account many details of the actual implementation, a number of Coq tactics needed to be written to make them tractable.

Tactics to decide disjointness and equality for heaps turned out to be very important. In practice, proofs of disjointness and equality of heaps are ubiquitous, but tedious because one always needs to prove disjointness to make unions

of heaps commute; this situation rapidly leads to intricate proofs. For example, the proof of the lemma `hl_compaction` given in Sect. 4.2 leads to the creation of 15 sub-heaps, and 14 hypotheses of equality and disjointness. With these hypotheses, we need to prove several goals of disjointness and equality. Fortunately, the tactic language of Coq provides us with a means to automate such reasoning.

We also developed a certified tactic to verify automatically programs whose specifications belong to a fragment of separation logic without the separating implication (to compare with related work, this is the fragment of [9] without inductively defined datatypes).

We used this tactic to verify the `hmInit` function, leading to a proof script three times smaller than the corresponding interactive proof we made (58 lines/167 lines). Although the code in this case is straight-line, the verification is not fully automatic because our tactic does not deal directly with assertions such as `Array` and `Heap_List`.

Let us briefly comment on the implementation of this tactic. The target fragment is defined by the inductive type `assrt`. The tactic relies on a weakest-precondition generator `wp_frag` whose outputs are captured by another inductive type `L_assrt`. Using this weakest-precondition generator, a Hoare triple whose pre/post-conditions fall into the type `assrt` is amenable to a goal of the form `assrt → L_assrt → Prop`. Given a proof system `LWP` for such entailments, one can use the following lemma to automatically verify Hoare triples:

```
Lemma LWP_use: ∀ c P Q R,
  wp_frag (Some (L_elt Q)) c = Some R →
  LWP P R →
  {{ assrt_interp P }} c {{ assrt_interp Q }}.
```

—The function `assrt_interp` projects objects of type `assrt` (a deep encoding) into the type `assert` (the shallow encoding introduced in this paper).
Goals of the form `assrt → L_assrt → Prop` can in general be solved automatically because the weakest-precondition generator returns goals that are inside the range of Presburger arithmetic (pointers are rarely multiplied between each other) for which Coq provides a native tactic (namely, the Omega test).

6.2 Translation from C Source Code

The programming language of separation logic is close enough to the subset of the C language used in the Topsy heap manager to enable a translation that preserves a syntactic correspondence. Thanks to this correspondence, it is immediate to identify a bug found during verification with its origin in the C source code. Below, we explain the main ideas behind the translation in question. Though it is systematic enough to be automated, we defer its certified implementation to future work and do it by hand for the time being.

The main difficulty in translating the original C source code is the lack of function calls and labelled jumps (in particular, the `break` instruction) in separation logic. To deal with function calls, we add global variables to serve as local variables and to carry the return value. To deal with the `break` instruction, we

```
static void compact(HmEntry at) {
 HmEntry atNext;

 while (at != NULL) {
  atNext = at->next;


  while ((at->status == HM_FREED) &&
         (atNext != NULL)) {


    if (atNext->status != HM_FREED)
         break;

    at->next = atNext->next;

    atNext = atNext->next;
  }
  at = at->next;}
}
```

```
Definition compact (at
  atNext
  brk tmp cstts nstts:var.v) :=
while (var_e at =/= null) (
  atNext <-* (at -.> next);
  brk <- nat_e 1 ;
  cstts <-* (at -.> status);
  while ((var_e cstts == Free) &&&
         (var_e atNext =/= null) &&&
         (var_e brk == nat_e 1)) (
     nstts <-* (atNext -.> status);
     ifte (var_e nstts =/= Free) thendo (
        brk <- nat_e 0
     ) elsedo (
        tmp <-* atNext -.> next;
        at -.> next *<- var_e tmp;
        atNext <-* atNext -.> next
     ));
  at <-* (at -.> next)
).
```

Fig. 1. Code Translation from C to Coq—Example

add a global variable and a conditional branching to force exiting where loops can break.

Another minor point is that we need to add temporary variables to make up for the restricted set of expressions and commands of separation logic. For example, the evaluation of an expression in separation logic never returns a location, only values, thus we need beforehand to load a location into variable to be able to use it in a boolean expression; also, there is no command to lookup *and* mutate memory at the same time. We overcome these restrictions by decomposing complex expressions and commands, and using temporary variables. These temporary variables correspond to the parameters written without vowels in our specifications.

By way of example, Fig. 1 displays side-by-side the original `compact` function and its Coq counterpart.

Table 1 summarizes the whole Coq implementation.

7 Benefits of Formal Verification

The main output of our experiment is that we have found several issues and bugs in the original source code of the Topsy heap manager. Another output is the Coq implementation of separation logic, that is readily available for other experiments. In particular, the verification of the Topsy heap manager in itself can actually be used for other verifications.

Script files	Contents (lines)
`util.v`	Non-standard lemmas about integers, lists, etc. (825)
`heap.v`	Modules for locations, values, and heaps (2388)
`bipl.v`	Separation logic connectives (with tactics) (1579)
`axiomatic.v`	Separation logic triples, frame rule (1080)
`vc.v`	Weakest-precondition generator (196)
`contrib.v`	Various lemmas (arrays, etc.) (1077)
`contrib_tactics.v`	Various tactics (Omega extensions, etc.) (324)
`examples.v`	Small examples (411)
`example_reverse_list.v`	Reverse-list example (383)
`frag.v`	Tactic for a fragment of separation logic (1972)
`frag_examples.v`	Examples for the tactic above (176)

total: 10411 lines

Script files	Contents (lines)
`topsy_hm.v`	Heap-list definition and properties (1015)
`topsy_hmInit.v`	Initialization code, specification, and verification (313)
`topsy_hmAlloc.v`	Allocation code, specifications, and verifications (2762)
`topsy_hmFree.v`	Deallocation code, specification, and verification (536)
`hmAlloc_example.v`	Example of Sect. 7.2 (130)

total: 4756 lines

Table 1. Coq Implementation Overview

7.1 Issues and Bugs found in the Original Source Code

Out of Range Initialization When verifying the initialization function of the heap manager (Sect. 5.1), we found that the header of the terminal block was actually written outside of the memory area reserved for the heap manager. This illegal destructive update made the `Heap_List` assertion unprovable because the latter holds for a fixed area of memory. We corrected this bug by changing a single arithmetic operation, suggesting a programming miss. In all fairness, we must say that this bug was corrected in versions of Topsy posterior to version 2 (that we are using for verification).

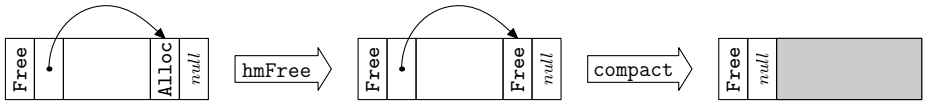
Optimizations of Allocation When verifying the allocation function (Sect. 5.2), we found several useless operations that suggested immediate optimizations.

One such useless operation is the possibility to allocate a non-empty memory block (that is, a header and a non-empty array of memory) when performing a null-size allocation. Since null-size allocations are not filtered out, the alignment calculation is applied anyway, resulting in a non-empty allocation (in addition to the header). This was highlighted when writing assertions. We improved the implementation by forcing failure for null-size allocation.

Among other optimizations, there were useless assignments (to dead variables) and useless tests. For example, there were two identical variables assignments before calling and at the beginning of the `findFree` function; this was

highlighted when writing the loop invariant in `findFree`. More interestingly, there was a useless test in the `compact` function. The second conjunct of the test of the inner loop (see Fig. 1) is useless because only the terminal block marked as allocated can point to `null`. Such an optimization cannot be done by an ordinary compilers, contrary to the former one.

Deallocation of the Terminal Block When verifying the deallocation function (Sect. 5.3), we found that it was possible to suppress allocable space without performing any allocation. This is because it is possible to deallocate the terminal block of the heap-list to trick compaction. The problem is better explained by the following scenario:



In this scenario, the terminal block is preceded by a free block. If we deallocate the terminal block and try to allocate a too-large block, this will trigger compaction and cause the leading free block to point to null. This problem is easily identified by the `Heap_List` assertion that enforces the terminal block to be marked as allocated. We fixed this problem by adding a test on the "next" field of the block to be deallocated in the deallocation function.

7.2 Using the Verification Result to Verify Other Code

Our verification of the Topsy heap manager provides us with new separation logic axioms that can be used for dynamic memory allocation without resorting to the native malloc/free commands of separation logic. In other words, we can use the specifications of `hmAlloc` and `hmFree` as triples to verify programs. For example, let us consider the following program:

```
Definition hmAlloc_example result entry cptr fnd stts nptr sz v :=
 hmAlloc result 1 entry cptr fnd stts nptr sz;
 ifte (var_e result =/= nat_e 0) thendo (
   (var_e result *<- int_e v)
 ) elsedo ( skip ).
```

This program allocates a new block using `hmAlloc`, stores its location into the variable `result`, and stores some value `v` into this block. Using the specification of `hmAlloc` proved in Sect. 5.2, we can prove the following specification:

```
Definition hmAlloc_example_specif := ∀ v x e p, p > 0 →
 {{ (nat_e x ↦ int_e e) ⋆
    (fun s h => ∃ l, (s |= var_e hmStart == nat_e p) ∧
     Heap_List l p s h ∧ In_hl l (x,1,alloc) p) }}
  hmAlloc_example result entry cptr fnd stts nptr sz v
 {{ fun s h => s |= var_e result =/= nat_e 0 →
    ((nat_e x ↦ int_e e) ⋆ (var_e result ↦ int_e v) ⋆ TT ⋆
     (fun s h => ∃ l, Heap_List l p s h ∧ In_hl l (x,1,alloc) p)) s h }}.
```

The post-condition asserts that, in case of successful allocation, the newly allocated block is separated from any previously allocated block.

8 Related Work

Our use case is reminiscent of work by Yu et al. that propose an assembly language for proof-carrying code and apply it to certification of dynamic storage allocation [7]. The main difference is that we deal with existing C code, whose verification is more involved because it has not been written with verification in mind. In particular, the heap-list data structure has been designed to optimize space usage; this leads to trickier manipulations (e.g., nested loop in `compact`), longer source code, and ultimately bugs, as we saw in Sect. 7.1. Also both allocators differ: the Topsy heap manager is a real allocation facility in the sense that the allocation function is self-contained (the allocator of Yu et al. relies on a pre-existing allocator) and that the deallocation function deallocated only valid blocks (the deallocator of Yu et al. can deallocate partial blocks).

The implementation of separation logic we did in the Coq proof assistant improves the work by Weber in the Isabelle proof assistant [8]. We think that our implementation is richer since it benefits from a substantial use case. In particular, we have developed several practical lemmas and tactics. Both implementations also differ in the way they implement heaps: we use an abstract data type implemented by means of modules for the heap whereas Weber uses partial functions.

Mehta and Nipkow developed modeling and reasoning methods for imperative programs with pointers in Isabelle [12]. The key idea of their approach is to model each heap-based data structure as a mapping from locations to values together with a relation, from which one derives required lemmas such as separation lemmas. The combination of this approach with Isabelle leads to compact proofs, as exemplified by the verification of the Schorr-Waite algorithm. In contrast, separation logic provides native notions of heap and separation, making it easier to model, for example, a heap containing different data structures (as it is the case for the `hmInit` function). The downside of separation logic is its special connectives that call for more implementation work regarding automation.

Tuch and Klein extended the ideas of Mehta and Nipkow to accommodate multiple datatypes in the heap by adding a mapping from locations to types [13]. Thanks to this extension, the authors certified an abstraction of the virtual mapping mechanism in the L4 kernel from which they generate verified C code. Obviously, such a refinement strategy is not directly applicable to the verification of existing code such as the Topsy heap manager. More importantly, the authors point that the verification of the implementation of malloc/free primitives is not possible in their setting because they "break the abstraction barrier" (Sect. 6 of [13]).

Schirmer also developed a framework for Hoare logic-style verification inside Isabelle [11]. The encoded programming language is very rich, including in particular procedure calls, and heap-based data structures can be modeled using the same techniques as Mehta and Nipkow. Thanks to the encoding of procedure calls, it becomes easier to model existing source code (by avoiding, for example, the numerous variables we needed to add to translate the source code of the Topsy heap manager into our encoding of separation logic). However, it is

not clear whether this richer encoding scales well for verification of non-trivial examples.

Caduceus [14] is a tool that takes a C program annotated with assertions and generates verification conditions that can be validated with various theorem provers and proof assistants. It has been used to verify several non-trivial C programs including the Schorr-Waite algorithm [15]. The verification of the Topsy heap manager could have been done equally well using a combination of Caduceus and Coq. However, Caduceus does support separation logic. Also, we needed a verification tool for assembly code in Topsy; for this purpose, a large part of our implementation for separation logic is readily reusable (this is actually work in progress). Last, we wanted to certify automation inside Coq instead of relying on a external verification condition generator.

Berdine, Calcagno and O'Hearn have developed Smallfoot, a tool for checking separation logic specifications [10]. It uses symbolic execution to produce verification conditions, and a decision procedure to prove them. Although Smallfoot is automatic (even for recursive and concurrent procedures), the assertions only describe the shape of data structures without pointer arithmetic. Such a limitation excludes its use for data structures such as heap-lists.

9 Conclusion

In this paper, we formally specified and verified the heap manager of the Topsy operating system inside the Coq proof assistant. In order to deal with pointers and ensure the separation of memory blocks, we used separation logic. This verification approach proved very effective since it enabled us to find bugs in the original C source code. In addition, this use-case led us to develop a Coq library of lemmas and tactics that is reusable for other formal verifications of low-level code.

Recent Work According to the specification described in this paper, an allocation function that always fails is correct. A complementary specification should make clearer the condition under which the allocation function is expected to succeed. In Topsy, allocation always succeeds when there is a list of contiguous free blocks whose compaction has the requested size. We have recently completed the verification of such a specification:

```
Definition hmAlloc_specif2 := ∀ adr size, adr > 0 → size > 0 →
  {{ fun s h => ∃ l1, ∃ l2, ∃ l,
    (Heap_List (l1 ++ (Free_block_list l) ++ l2) adr) s h ∧
    Free_block_compact_size l ≥ size ∧
    (s |= var_e hmStart == nat_e adr) }}
  hmAlloc result size entry cptr fnd stts nptr sz
  {{ fun s h => ∃ l, ∃ y,
      y > 0 ∧ (s |= var_e result == nat_e (y+2)) ∧
      ∃ size'', size'' ≥ size ∧
      (Heap_List l adr ⋆ Array (y+2) size'') s h ∧
      In_hl l (y,size'',alloc) adr }}.
```

This verification turned out to be technically more involved than the one described in this paper because of the numerous clauses required by the verification of `compact`.

Acknowledgments The authors would like to thank Prof. Andrew W. Appel and his colleagues at INRIA for numerous suggestions that substantially improved the Coq implementation.

References

1. John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, p. 55–74. Invited lecture.
2. Lukas Ruf and various contributors. TOPSY – A Teachable Operating System. `http://www.topsy.net/`.
3. Lukas Ruf, Claudio Jeker, Boris Lutz, and Bernhard Plattner. Topsy v3: A NodeOS For Network Processors. In *2nd International Workshop on Active Network Technologies and Applications (ANTA 2003)*.
4. Various contributors. The Coq Proof assistant. `http://coq.inria.fr`.
5. Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Towards Formal Verification of Memory Properties using Separation Logic. In *22nd Workshop of the Japan Society for Software Science and Technology (JSSST 2005)*.
6. Reynald Affeldt and Nicolas Marti. Towards Formal Verification of Memory Properties using Separation Logic. `http://savannah.nongnu.org/projects/seplog`. Online CVS.
7. Dachuan Yu, Nadeem Abdul Hamid, and Zhong Shao. Building Certified Libraries for PCC: Dynamic Storage Allocation. *Science of Computer Programming*, 50(1-3):101–127. Elsevier, Mar. 2004.
8. Tjark Weber. Towards Mechanized Program Verification with Separation Logic. In *13th Conference on Computer Science Logic (CSL 2004)*, volume 3210 of *LNCS*, p. 250–264. Springer.
9. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A Decidable Fragment of Separation Logic. In *24th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2004)*, volume 3328 of *LNCS*, p. 97–109. Springer.
10. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic Execution with Separation Logic. In *3rd Asian Symposium on Programming Languages and Systems (APLAS 2005)*, volume 3780 of *LNCS*, p. 52–68. Springer.
11. Norbert Schirmer. A Verification Environment for Sequential Imperative Programs in Isabelle/HOL. In *11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2004)*, volume 3452 of *LNCS*, p. 398–414. Springer.
12. Farhad Mehta and Tobias Nipkow. Proving Pointer Programs in Higher-Order Logic. In *Information and Computation*, 199:200–227. Elsevier, 2005.
13. Harvey Tuch and Gerwin Klein. A Unified Memory Model for Pointers. In *12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2005)*, volume 3835 of *LNCS*, p. 474–488. Springer.
14. Jean-Christophe Filliâtre. Multi-Prover Verification of C Programs. In *6th International Conference on Formal Engineering Methods (ICFEM 2004)*, volume 3308 of *LNCS*, p. 15–29. Springer.

15. Thierry Hubert and Claude Marché. A case study of C source code verification: the Schorr-Waite algorithm. In *3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005)*.

A Proof Sketches for `hmAlloc` and `hmFree`

We provide a bird's-eye view of the formal verification of `hmAlloc` and `hmFree` using proof sketches. For each function, the Coq model is displayed side-by-side with relevant assertions: each command is juxtaposed with its post-condition, its pre-condition being the post-condition of the previous command. Assertions corresponding to loop invariants are boxed. The grayed areas indicate where the key lemmas are applied (the name of the corresponding lemma is underlined).

The assertions are written in an abbreviated syntax to save space:

- We write $\texttt{HL}(l, \texttt{adr})$ for the assertion `Heap_List` l `adr`.
- $(x, y, z) \in_{\texttt{adr}} l$ stands for the assertion `In_hl` l (x, y, z) `adr`.
- In all the proof sketches, the assertion `hmStart` = `adr` holds.
- In the proof of `compact`, we write $P_{\texttt{Allocated}}$ for all the assertions where `cstts` = `Allocated`; for states corresponding to these assertions, the loop is actually not executed.

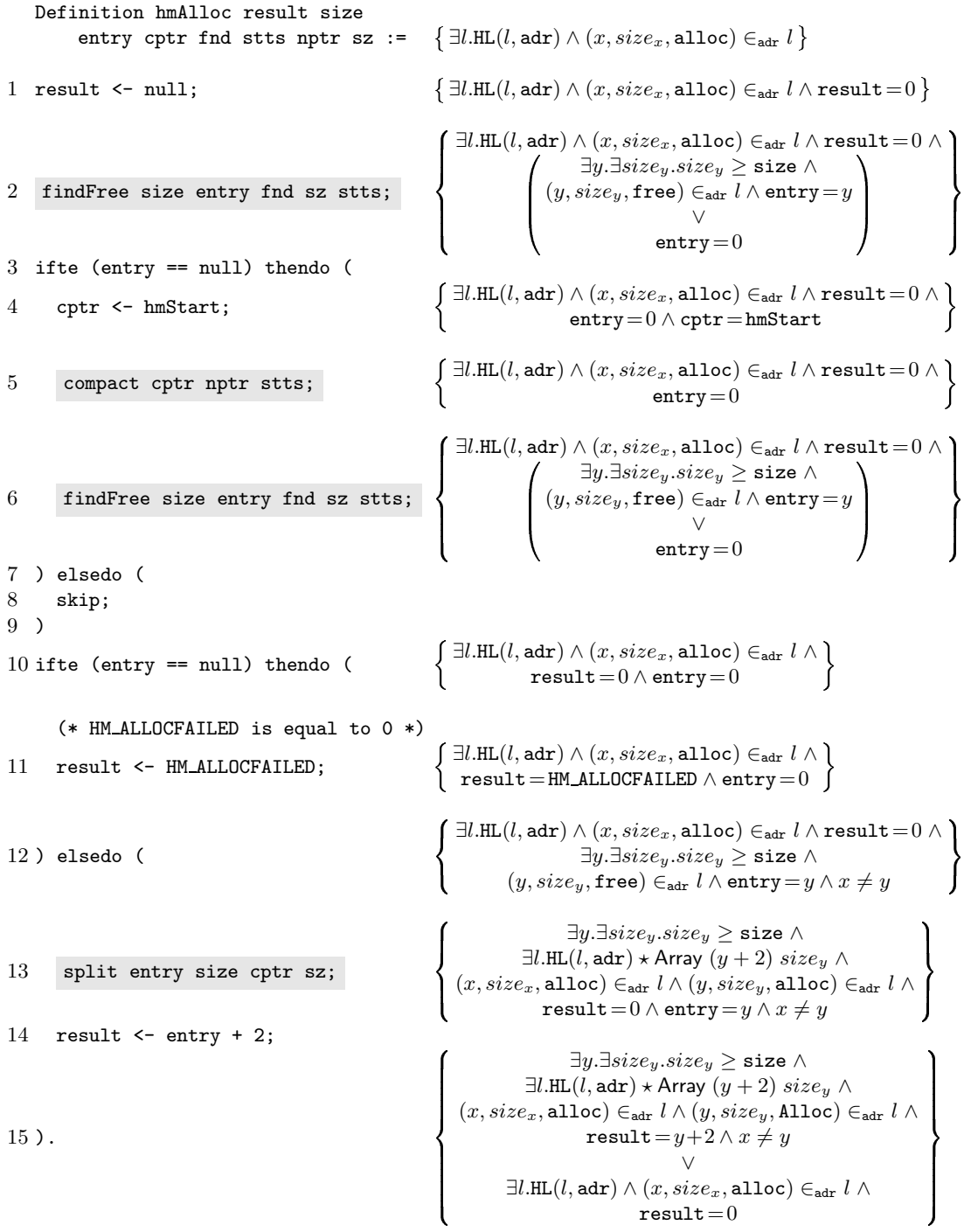


Fig. 2. Proof sketch of `hmAlloc` (see Fig. 3, 4, and 5 for the proof sketches of `findFree`, `compact`, and `split`)

Fig. 3. Proof sketch of `findFree` (partial proof of `hmAlloc` in Fig. 2)

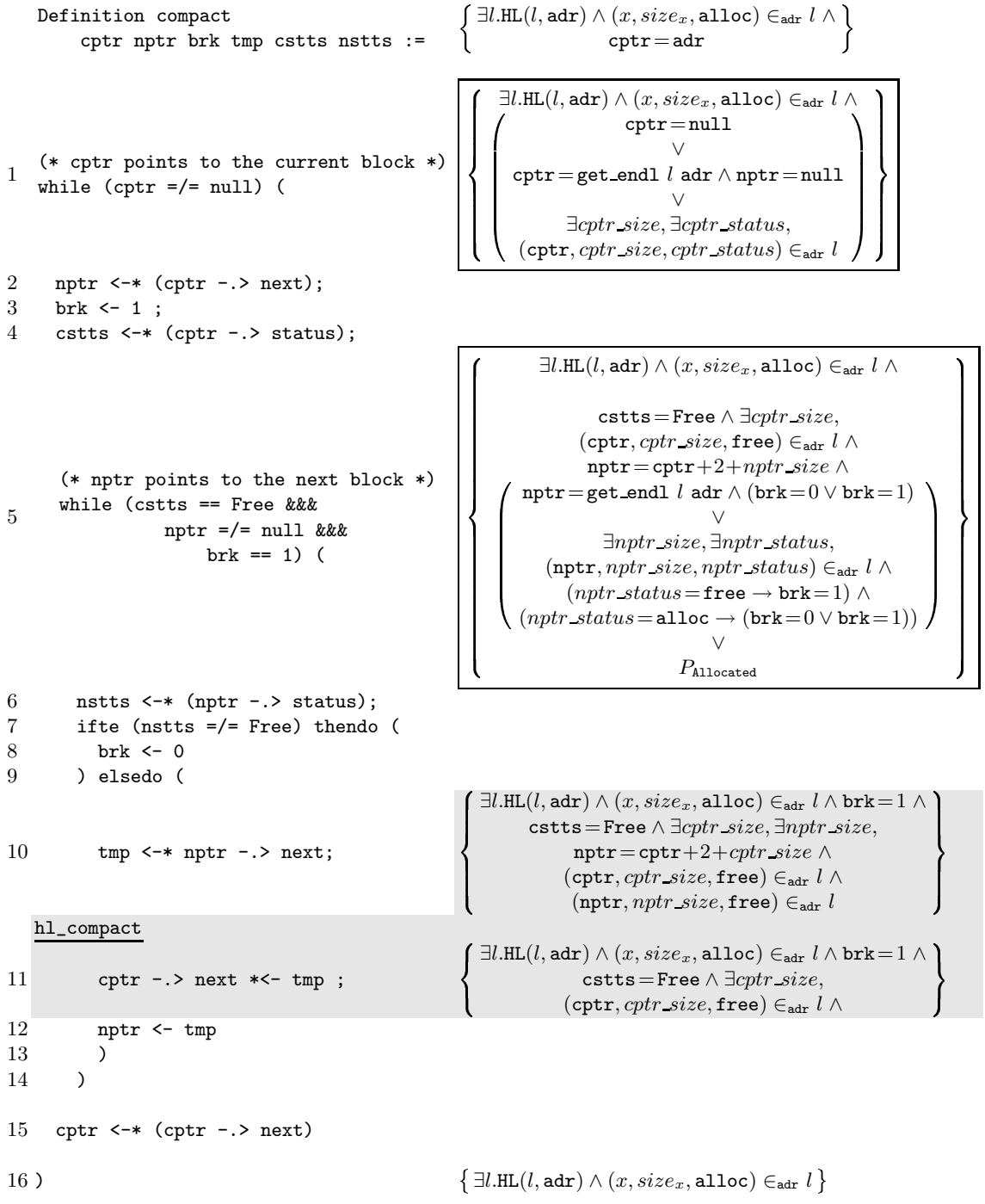


Fig. 4. Proof sketch of `compact` (partial proof of `hmAlloc` in Fig. 2)

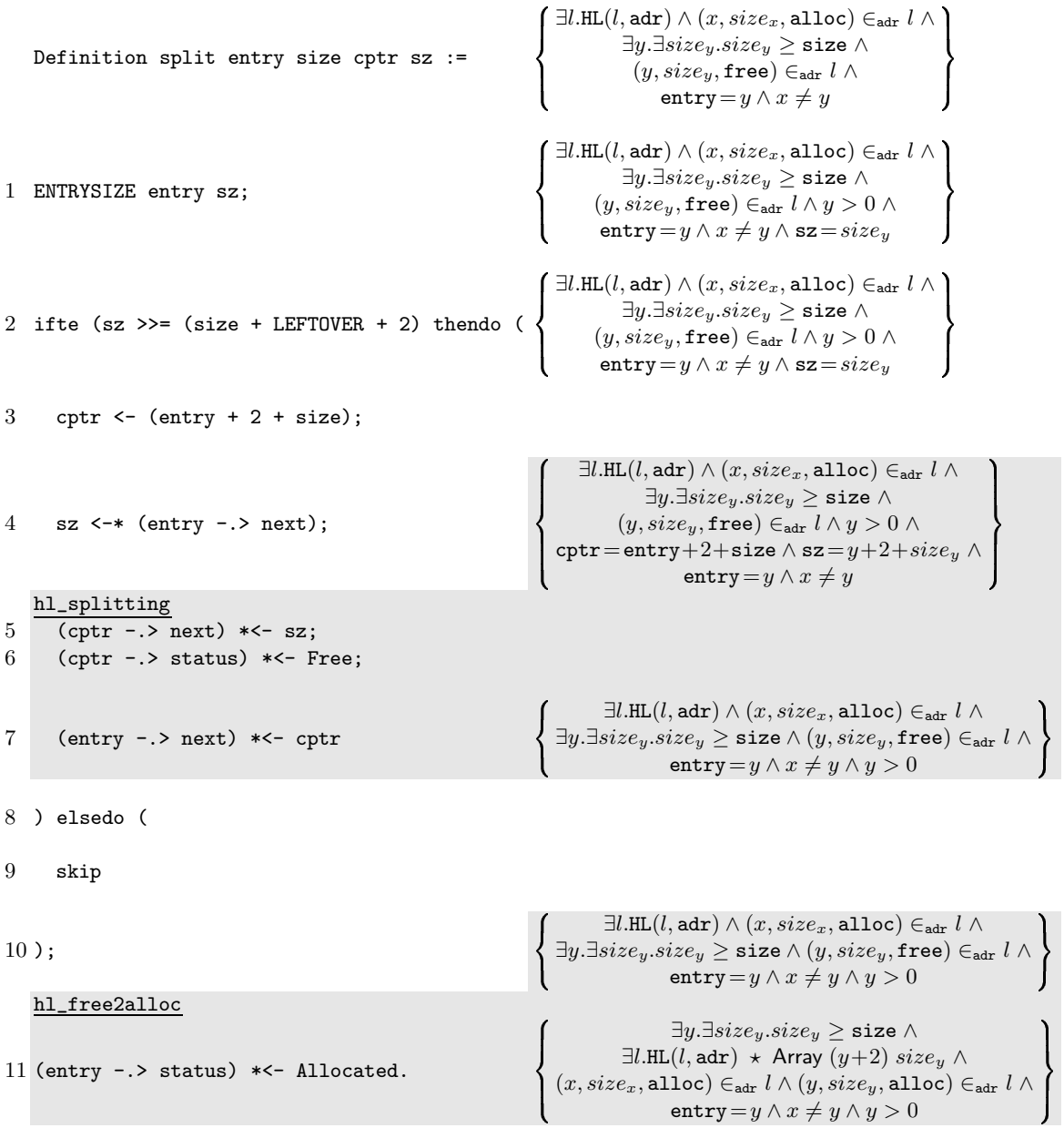


Fig. 5. Proof sketch of `split` (partial proof of `hmAlloc` in Fig. 2)

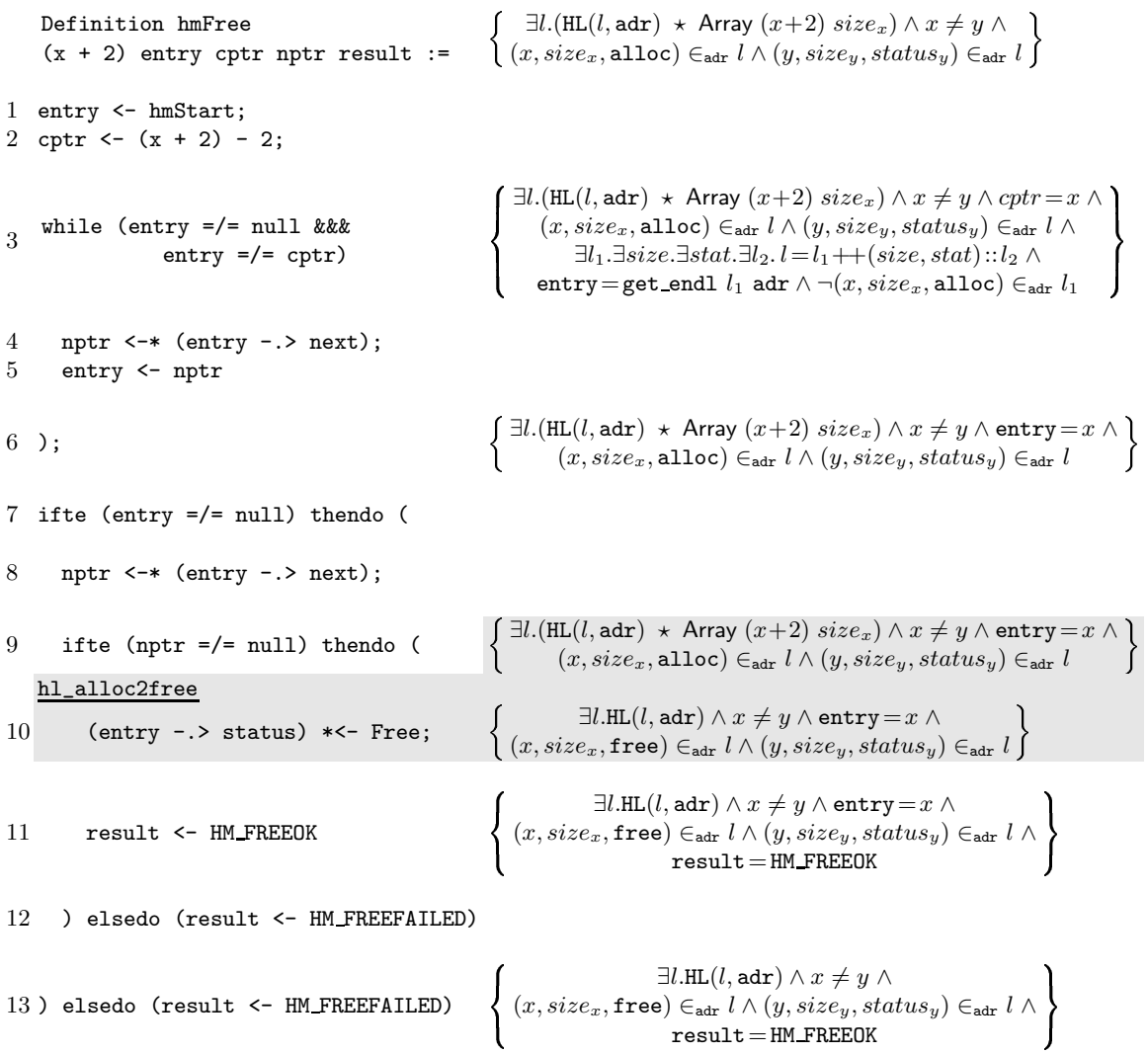


Fig. 6. Proof sketch of `hmFree`