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

Nicolas Marti† nicolas at yl.is.s.u-tokyo.ac.jp
Reynald Affeldt‡ reynald.affeldt at aist.go.jp
Akinori Yonezawa†‡ yonezawa at yl.is.s.u-tokyo.ac.jp

†Department of Computer Science, University of Tokyo
‡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, such a verification 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. Our main contribution in this paper is to verify the heap manager of an existing operating system, namely Topsy. For this purpose, we use separation logic, an extension of Hoare logic to deal with pointers. Thanks to our verification, we found several issues in the original source code. Another output of our verification is our Coq implementation of separation logic.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification — Formal Methods; D.4.6 [**Operating Systems**]: Reliability — Verification; F.3.1 [**Logics and Meanings of Programs**]: Mechanical verification

General Terms

Verification

Keywords

Operating systems, Dynamic memory allocation, Mechanical verification

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 [7].

Our main contribution in this paper is to verify the heap manager of an existing operating system, namely Topsy [2]. For this purpose, we use separation logic [1], an extension of Hoare logic to deal with pointers, and we implement the whole verification in the Coq proof assistant [4]. In fact, the heap manager proves harder to deal with than dynamic memory allocation facilities verified in previous studies (see Sect. 6.1 for a comparison). A direct side-effect of our approach of using an existing heap manager is to provide advanced debugging. Indeed, our verification highlights several issues in the original source code (see Sect. 5.4 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 provides 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 Topsy heap manager, and we explain our verification goal and approach. In Sect. 3, we formally specify the underlying data structure used by the heap manager. In Sect. 4, we explain the formal verification of the heap manager using a pencil-and-paper version of our mechanical proof. In Sect. 5, we give an overview of our Coq implementation of separation logic and of the verification in itself, and we discuss issues found in the original source code of the heap manager. In Sect. 6, we conclude, and comment on related and future work.

2. VERIFICATION GOAL AND APPROACH

In this section, we give an overview of the Topsy heap manager, and we explain our verification goal and approach.

General information about Topsy (including a browsable source code) is available online [2].

2.1 Topsy Heap Manager

The heap manager of Topsy is the set of functions that provides the operating system with dynamic memory allocation. These functions and related variables are defined in the files `Memory/MMHeapMemory.{h,c}`, with some macros in the file `Topsy/Configuration.h`.

2.1.1 The Heap

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, the repartition of allocated and free memory blocks in Topsy form a partition of the heap. This is achieved by implementing memory blocks as a simply-linked list. In the following, we refer to this data structure as a *heap-list*. See Fig. 1 for a concrete example of a heap-list.

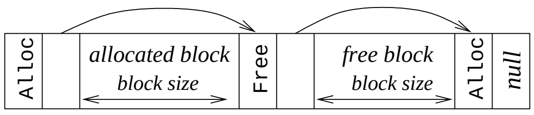

Figure 1: A heap-list with one allocated block and one free block

In a heap-list, each block consists of a two-fields header and an array of memory. The first field 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. 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.

2.1.2 Initialization

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`.

2.1.3 Allocation

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` makes several manipulations over free blocks. Precisely, it performs compaction of contiguous free blocks and splitting of free blocks.

2.1.4 Deallocation

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, otherwise it returns the value `HM_FREEFAILED`.

2.2 Verification Goal

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 "fresh" memory blocks (they do not override previously allocated memory blocks), the deallocation function turns the status of blocks into free (except the ending empty block), the three functions together correctly manage the heap-list data structure (the allocation and deallocation functions do not modify unrelated memory blocks, and the three functions preserve the heap-list structure). Guaranteeing the allocation of fresh memory blocks and the non-modification of unrelated 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. 4.

2.3 Verification Approach

Our approach is to use separation logic to formally specify and mechanically verify the goal informally stated above directly on the source code of the Topsy heap manager.

Although all the verification has been performed in the Coq proof assistant, we use the more convenient pencil-and-paper notation of separation logic to write assertions; the correspondence with the implementation is illustrated in Sect. 5.3. Concerning the source code of the Topsy heap manager, we comment on its Coq version because it more closely matches specifications. Evidence of the correspondence with the original C source code is given in Sect. 5.2.

3. THE HEAP-LIST DATA STRUCTURE

In this section, we formally specify the heap-list data structure and state its main properties.

3.1 The Heap-list Type

We formally specify the heap-list data structure that underlies the heap manager using a predicate: Heap-list $l\ x\ y$ holds for a contiguous area of memory that contains a heap-list starting at location x, and whose last block points to location y; since the terminal block of a heap-list points to null, $y = 0$ for a heap-list that covers all the heap. Information about the blocks contained in this heap-list is captured by the first parameter of the predicate: l is a (Coq) list of triples (*location*, *size*, *status*) characterizing the sequence of blocks.

Separation logic is very convenient to specify the heap-list data structure. In particular, the property that blocks are disjoint can be expressed directly using the separating conjunction. In addition, the fact the blocks form a partition of the heap can be represented using pointer arithmetic, that separation logic also handles.

Before giving the definition of the predicate Heap-list, we first define a predicate Array that characterizes sets of contiguous locations ($*$ is the separating conjunction, ϵ holds for an empty area of memory, and $\mapsto$ is the maps-to assertion of separation logic):

$$\begin{aligned}\mathsf{Array}\ l\ sz &\overset{def}{=} (sz = 0 \wedge \epsilon) \vee \\ &(sz > 0) \wedge (\exists e.(l \mapsto e) * \mathsf{Array}\ (l{+}1)\ (sz{-}1))\end{aligned}$$

We now give the definition of the predicate Heap-list. It takes the form of a recursive definition with three disjunctive clauses (:: is the list constructor):

$$\begin{aligned}&\mathsf{Heap\text{-}list}\ (l : (loc \times nat \times status)\ list)\ (x : loc)\ (y : loc) \overset{def}{=} \\ &\quad l = nil \wedge (x \mapsto \_, null) \wedge y = 0 \vee \\ &\quad l = nil \wedge x = y \geq 0 \wedge \epsilon \vee \\ &\quad \exists size. \exists status. \exists l'. \\ &\qquad (status = \mathtt{Alloc} \vee status = \mathtt{Free}) \wedge \\ &\qquad l = (x, size, status) :: l' \wedge x > 0 \wedge \\ &\qquad (x \mapsto status, x{+}2{+}size) * \\ &\qquad\quad \mathsf{Array}\ (x{+}2)\ size * \mathsf{Heap\text{-}list}\ l'\ (x{+}2{+}size)\ y\end{aligned}$$

The first clause holds for the terminal block (the one pointing to null). The second clause holds for empty heap-lists. The third clause holds for a memory block (allocated or free) such that its header points to an immediately following heap-list. Observe that the definition of the heap-list guarantees that there is no lost space between two blocks.

3.2 Main Properties of Heap-lists

We summarize the properties that are at the heart of the formal verification of the heap manager. They take the form of lemmas that make precise the conditions under which one can change the status of blocks, compact blocks, and split blocks. Since these operations rely on destructive updates, the properties in questions are expressed using the separating implication (noted $-\!\!*$).

3.2.1 Change of Status

The following lemma is used to change the status of a block (from free to allocated, the symmetric lemma is similar):

CHANGE-STATUS:

$$\begin{aligned}&\mathsf{Heap\text{-}list}\ (l_1 {+\!\!+} ((x, size, \_) :: nil) {+\!\!+} l_2)\ x_0\ 0 \rightarrow \\ &(x \mapsto \_) * \\ &\quad ((x \mapsto \mathtt{Alloc}) -\!\!* \\ &\qquad \mathsf{Heap\text{-}list}\ (l_1 {+\!\!+} ((x, size, \mathtt{Alloc}) :: nil) {+\!\!+} l_2)\ x_0\ 0)\end{aligned}$$

The left-hand side of the implication states the existence of some block x. The right-hand size of the implication is of the form $(x \mapsto \_) * ((x \mapsto \mathtt{Alloc}) -\!\!* \ldots)$: this captures destructive update of the status field of the block x. The conclusion of the lemma states the change of status to allocated.

3.2.2 Compaction

The following lemma expresses concatenation of contiguous free blocks:

COMPACTION:

$$\begin{aligned}&\mathsf{Heap\text{-}list}\ (l_1 {+\!\!+} ((x, size_1, \mathtt{Free}) :: spareblock :: nil) {+\!\!+} l_2)\ x_0\ 0 \rightarrow \\ &(x{+}1 \mapsto address\_spareblock) * \\ &\ ((x{+}1 \mapsto address\_nextblock) -\!\!* \\ &\quad \mathsf{Heap\text{-}list}\ (l_1 {+\!\!+} ((x, size_1{+}2{+}size_2, \mathtt{Free}) :: nil) {+\!\!+} l_2)\ x_0\ 0) \\ &\text{where} \\ &address\_spareblock = x{+}2{+}size_1 \\ &spareblock = (adress\_spareblock, size_2, \mathtt{Free}) \\ &address\_nextblock = x{+}size_1{+}4{+}size_2\end{aligned}$$

The left-hand side of the implication states the existence of some block x immediately followed by some "spare block". The right-hand side of the implication captures the destructive update of the next field of the block x: this block is made to point to the next block following the spare block. As a result, the block x sees its size increased by the size the of the spare block. Compaction is pictured in Fig. 2.

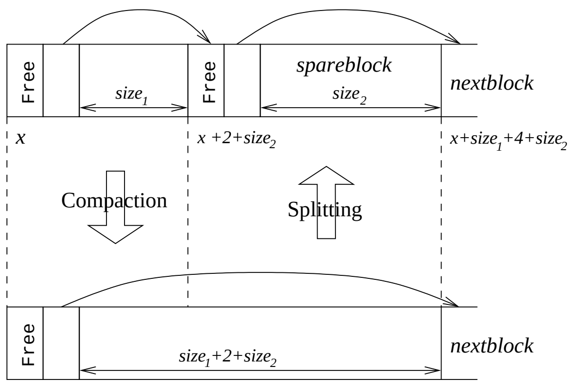

Figure 2: Compaction and Splitting of Blocks

3.2.3 Splitting

The following lemma is used for splitting a free block in two:

SPLITTING:

$$\begin{aligned}&\mathsf{Heap\text{-}list}\ (l_1 {+\!\!+} ((x, size_1{+}2{+}size_2, \mathtt{Free}) :: nil) {+\!\!+} l_2)\ x_0\ 0 \rightarrow \\ &((x{+}3{+}size_1 \mapsto \_) * \\ &((x{+}3{+}size_1 \mapsto address\_nextblock) -\!\!* \\ &\ ((address\_spareblock \mapsto \_) * \\ &\ ((address\_spareblock \mapsto \mathtt{Free}) -\!\!* \\ &\quad ((x{+}1 \mapsto \_) * \\ &\quad ((x{+}1 \mapsto address\_spareblock) -\!\!* \\ &\qquad \mathsf{Heap\text{-}list}\ (l_1 {+\!\!+} ((x, size_1, \mathtt{Free}) :: spareblock :: nil) {+\!\!+} l_2)\ x_0\ 0)))))) \\ &\text{where} \\ &address\_spareblock = x{+}2{+}size_1 \\ &spareblock = (adress\_spareblock, size_2, \mathtt{Free}) \\ &address\_nextblock = x{+}size_1{+}4{+}size_2\end{aligned}$$

The left-hand size of the implication states the existence of a block x of size $size_1{+}2{+}size_2$. The conclusion of the lemma states the splitting of this block into a block x of size $size_1$, and a new "spare block" of size $size_2$. The three destructive updates represent the modifications of the headers that are necessary to introduce the new spare block: (1) link from the spare block to the next block, (2) spare block status set to Free, and (3) link from block x to the spare block. Splitting is pictured in Fig. 2 (together with compaction).

4. FORMAL VERIFICATION

In this section, we explain the formal verification of the functions of the Topsy heap manager. For each function (initialization, allocation, deallocation), we give its formal specification using Hoare triples. We explain in details the verification of the allocation function, which is the most involved.

4.1 Formal Verification of Initialization

The initialization function hmInit transforms a given area of memory, starting at location hmStart and of fixed length KERNELHEAPSIZE, into an initial heap-list. This initial heap-list consists of a single free block that spans the whole heap. Using the Array and Heap-list predicates, the formal specification of hmInit becomes:

$$\left\{\ \mathsf{Array}\ \mathtt{hmStart}\ \mathtt{KERNELHEAPSIZE}\ \right\}$$
$$\mathtt{(hmInit\ hmStart\ KERNELHEAPSIZE)}$$
$$\left\{\begin{array}{c}\mathsf{Heap\text{-}list}\ ((\mathtt{hmStart}, \mathtt{KERNELHEAPSIZE}-4, \mathtt{Free}) :: nil)\\ \mathtt{hmStart}\ 0\end{array}\right\}$$

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 headers (including the header of the terminal block).

The verification of hmInit amounts to the symbolic evaluation of the function and check that the resulting heap-list is an instance of the eponymous predicate. Despite its apparent simplicity, this function turns out to be buggy, as we explain in Sect. 5.4.

4.2 Formal Verification of Allocation

The allocation function hmAlloc is implemented by searching for a large-enough free block in the heap-list, possibly performing compaction of free blocks in the process. 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.

The specification of the allocation function consists in checking that (1) newly allocated blocks do not use already allocated locations (they are "fresh") and (2) already allocated blocks are not modified. In the pre-condition, we isolate some already allocated block (block $(x, size_x, \mathtt{Alloc})$ below), and its contents (captured by the predicate Init-array, and by $list_x$, a list of cell values, which length is $size_x$). Note that the variables that are not existentially quantified in an assertion are universally quantified. In the post-condition, we ensure that the newly allocated block (starting at some location y) has the appropriate size, that it does not override the already allocated block, and that the contents of the latter are unchanged. The second disjunction in the post-condition applies when allocation fails:

$$\left\{\begin{array}{c}\exists l.\mathsf{Heap\text{-}list}\ l\ \mathtt{hmStart}\ 0 \wedge (x, size_x, \mathtt{Alloc}) \in l\ \wedge\\ \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x)\end{array}\right\}$$
$$\mathtt{(hmAlloc\ result\ size\ entry\ cptr\ fnd\ stts\ nptr\ sz)}$$
$$\left\{\begin{array}{c}\exists l.\mathsf{Heap\text{-}list}\ l\ \mathtt{hmStart}\ 0 \wedge (x, size_x, \mathtt{Alloc}) \in l\ \wedge\\ \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x)\ \wedge\\ \left(\begin{array}{c}\exists y.\exists size_y.size_y \geq \mathtt{size} \wedge (y, size_y, \mathtt{Alloc}) \in l\ \wedge\\ \mathtt{entry}{=}y \wedge \mathtt{result}{=}\mathtt{entry}{+}2 \wedge x \neq y\\ \vee\\ \mathtt{result}{=}0\end{array}\right)\end{array}\right\}$$

The predicate Init-array is built on the model of the predicate Array:

$$\mathsf{Init\text{-}array}\ l\ lst \overset{def}{=} (lst = nil)\ \vee$$
$$(lst{=}hd{::}tl) \wedge ((l \mapsto hd) * \mathsf{Init\text{-}array}\ (l{+}1)\ tl)$$

The whole formal verification of hmAlloc is summarized in Fig. 3. For the sake of explanation, we decompose the verification in three phases matching the underlying implementation (traversal, compaction, splitting, in the next three sections).

4.2.1 Traversal

The first phase of allocation consists in traversing the heap-list to find a large-enough free block. It is implemented by the function findFree whose main arguments are the required size (variable size) and an integer to be filled with the location of an appropriate block if any (variable entry) (other arguments represent local variables of the function). Taken in isolation, verification of findFree consists in verifying the following Hoare triple:

$$\left\{\begin{array}{c}\exists l.\mathsf{Heap\text{-}list}\ l\ \mathtt{hmStart}\ 0 \wedge (x, size_x, \mathtt{Alloc}) \in l\ \wedge\\ \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x)\end{array}\right\}$$
$$\mathtt{(findFree\ size\ entry\ fnd\ sz\ stts)}$$
$$\left\{\begin{array}{c}\exists l.\ \mathsf{Heap\text{-}list}\ l\ \mathtt{hmStart}\ 0 \wedge (x, size_x, \mathtt{Alloc}) \in l\ \wedge\\ \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x)\ \wedge\\ \left(\begin{array}{c}\exists y.\exists size_y.size_y \geq \mathtt{size}\ \wedge\\ (y, size_y, \mathtt{Free}) \in l \wedge \mathtt{entry}{=}y\\ \vee\\ \mathtt{entry} = 0\end{array}\right)\end{array}\right\}$$

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, in which case the return value is null. (We have omitted the predicate $\mathtt{result}=0$ in pre/post-conditions for readability.)

The verification of findFree is displayed in Fig. 4. The traversal is implemented as a loop. It makes use of the variables entry, stts, and sz to keep track of the location of current block, its status, and its size. The traversal terminates either if it finds a large-enough free block (in which case the variable fnd in set to non-zero), or if it reaches the end of the heap-list (entry is null).

4.2.2 Compaction

The compaction phase only occurs when the 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. Compaction is implemented by the function compact. Its specification amounts to assert that it preserves the heap-list structure:

$$\left\{\begin{array}{c}\exists l.\mathsf{Heap\text{-}list}\ l\ \mathtt{hmStart}\ 0 \wedge (x, size_x, \mathtt{Alloc}) \in l\ \wedge\\ \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \mathtt{cptr}{=}\mathtt{hmStart}\end{array}\right\}$$
$$\mathtt{(compact\ cptr\ nptr\ stts)}$$
$$\left\{\begin{array}{c}\exists l.\mathsf{Heap\text{-}list}\ l\ \mathtt{hmStart}\ 0 \wedge (x, size_x, \mathtt{Alloc}) \in l\ \wedge\\ \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x)\end{array}\right\}$$

(We have omitted the predicate $\mathtt{result} = 0 \wedge \mathtt{entry} = 0$ for readability.)

The verification of compaction is displayed in Fig. 5. It is technically involved because it features two nested loops (and therefore large invariants), and because variables are overloaded (the variable stts is used both to hold integers

	`Definition hmAlloc result size entry`
	`cptr fnd stts nptr sz :=`
		$\{\ \exists l.\mathsf{Hp}(l) \land (x, size_x, \texttt{Alloc}) \in l \land \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x)\ \}$

1	`result <- null;`	$\{\ \exists l.\mathsf{Hp}(l) \land (x, size_x, \texttt{Alloc}) \in l \land \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \land \texttt{result} = 0\ \}$
2	`findFree size entry fnd sz stts;`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \land (x, size_x, \texttt{Alloc}) \in l \land \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \land \texttt{result} = 0 \land \\ \left(\begin{array}{c} \exists y.\exists size_y.size_y \geq \texttt{size} \land (y, size_y, \texttt{Free}) \in l \land \texttt{entry} = y \\ \lor \\ \texttt{entry} = 0 \end{array} \right) \end{array} \right\}$
3	`ifte (entry == null) thendo (`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \land (x, size_x, \texttt{Alloc}) \in l \land \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \land \texttt{result} = 0 \land \\ \texttt{entry} = 0 \end{array} \right\}$
4	`  cptr <- hmStart;`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \land (x, size_x, \texttt{Alloc}) \in l \land \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \land \texttt{result} = 0 \land \\ \texttt{entry} = 0 \land \texttt{cptr} = \texttt{hmStart} \end{array} \right\}$
5	`  compact cptr nptr stts;`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \land (x, size_x, \texttt{Alloc}) \in l \land \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \land \texttt{result} = 0 \land \\ \texttt{entry} = 0 \end{array} \right\}$
6	`  findFree size entry fnd sz stts;`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \land (x, size_x, \texttt{Alloc}) \in l \land \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \land \texttt{result} = 0 \land \\ \left(\begin{array}{c} \exists y.\exists size_y.size_y \geq \texttt{size} \land (y, size_y, \texttt{Free}) \in l \land \texttt{entry} = y \\ \lor \\ \texttt{entry} = 0 \end{array} \right) \end{array} \right\}$
7	`) elsedo (`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \land (x, size_x, \texttt{Alloc}) \in l \land \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \land \texttt{result} = 0 \land \\ \exists y.\exists size_y.size_y \geq \texttt{size} \land (y, size_y, \texttt{Free}) \in l \land \texttt{entry} = y \end{array} \right\}$
8	`  skip;`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \land (x, size_x, \texttt{Alloc}) \in l \land \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \land \texttt{result} = 0 \land \\ \exists y.\exists size_y.size_y \geq \texttt{size} \land (y, size_y, \texttt{Free}) \in l \land \texttt{entry} = y \end{array} \right\}$
9	`)`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \land (x, size_x, \texttt{Alloc}) \in l \land \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \land \texttt{result} = 0 \land \\ \left(\begin{array}{c} \exists y.\exists size_y.size_y \geq \texttt{size} \land (y, size_y, \texttt{Free}) \in l \land \texttt{entry} = y \\ \lor \\ \texttt{entry} = 0 \end{array} \right) \end{array} \right\}$
10	`ifte (entry == null) thendo (`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \land (x, size_x, \texttt{Alloc}) \in l \land \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \land \\ \texttt{result} = 0 \land \texttt{entry} = 0 \end{array} \right\}$
	`  (* HM_ALLOCFAILED is equal to 0 *)`	
11	`  result <- HM_ALLOCFAILED;`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \land (x, size_x, \texttt{Alloc}) \in l \land \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \land \\ \texttt{result} = 0 \land \texttt{entry} = 0 \end{array} \right\}$
12	`) elsedo (`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \land (x, size_x, \texttt{Alloc}) \in l \land \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \land \texttt{result} = 0 \land \\ \exists y.\exists size_y.size_y \geq \texttt{size} \land (y, size_y, \texttt{Free}) \in l \land \texttt{entry} = y \land x \neq y \end{array} \right\}$
13	`  split entry size cptr sz;`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \land (x, size_x, \texttt{Alloc}) \in l \land \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \land \texttt{result} = 0 \land \\ \exists y.\exists size_y.size_y \geq \texttt{size} \land (y, size_y, \texttt{Alloc}) \in l \land \texttt{entry} = y \land x \neq y \end{array} \right\}$
14	`  result <- entry + 2;`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \land (x, size_x, \texttt{Alloc}) \in l \land \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \land \\ \exists y.\exists size_y.size_y \geq \texttt{size} \land (y, size_y, \texttt{Alloc}) \in l \land \\ \texttt{entry} = y \land \texttt{result} = \texttt{entry}{+}2 \land x \neq y \end{array} \right\}$
15	`).`	$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \land (x, size_x, \texttt{Alloc}) \in l \land \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \land \\ \left(\begin{array}{c} \exists y.\exists size_y.size_y \geq \texttt{size} \land (y, size_y, \texttt{Alloc}) \in l \land \\ \texttt{entry} = y \land \texttt{result} = \texttt{entry}{+}2 \land x \neq y \\ \lor \\ \texttt{result} = 0 \end{array} \right) \end{array} \right\}$

$\mathsf{Hp}(l) \overset{def}{=} \mathsf{Heap\text{-}list}\ l\ \texttt{hmStart}\ 0$

Figure 3: Sketch of `hmAlloc` proof (the proofs for grayed instructions appear in Fig. 4, Fig. 5, and Fig. 6)

	`Definition findFree size entry fnd sz stts :=`	$\{\ \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x)\ \}$
1	`entry <- hmStart;`	$\{\ \ldots\ \}$
2	`stts <-* (entry -.> status);`	$\{\ \ldots\ \}$
3	`fnd <- 0;`	$\left\{\begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \\ \texttt{entry} = \texttt{hmStart} \wedge \\ \exists status.(status = \texttt{Alloc} \vee status = \texttt{Free}) \wedge \texttt{stts} = status \wedge \\ \exists size'.(\texttt{hmStart}, size', status) \in l \wedge \texttt{fnd} = 0 \end{array}\right\}$
4	`while ((entry =/= null) &&& (fnd =/= 1)) (`	$\boxed{\left\{\begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \\ \wedge \\ \left(\begin{array}{c} \left(\begin{array}{c} \exists bloc\_adr.\texttt{entry} = bloc\_adr \wedge bloc\_adr > 0 \wedge \\ \texttt{fnd} = 1 \wedge \\ \exists size'.size' \geq \texttt{size} \wedge (bloc\_adr, size', \texttt{Free}) \in l \\ \vee \\ \texttt{fnd} = 0 \wedge \\ \exists status.(status = \texttt{Alloc} \vee status = \texttt{Free}) \wedge \\ \texttt{stts} = status \wedge \exists size'.(bloc\_adr, size', \texttt{Free}) \in l \\ \vee \\ \texttt{fnd} = 0 \wedge \\ (\mathsf{Heap\text{-}list}\ l\ \texttt{hmStart}\ bloc\_adr\ * \\ \mathsf{Heap\text{-}list}\ nil\ bloc\_adr\ 0) \end{array}\right) \\ \vee \\ \texttt{entry} = 0 \end{array}\right) \end{array}\right\}}$
5	`  stts <-* (entry -.> status);`	$\{\ \ldots\ \}$
6	`  ENTRYSIZE entry sz;`	$\{\ \ldots\ \}$
7	`  ifte ((stts == Free) &&& (sz >>= size)) thendo`	$\{\ \ldots\ \}$
8	`    fnd <- 1`	$\left\{\begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \\ \exists bloc\_adr.\texttt{entry} = bloc\_adr \wedge bloc\_adr > 0 \wedge \\ \texttt{fnd} = 1 \wedge \\ \exists size'.size' \geq \texttt{size} \wedge (bloc\_adr, size', \texttt{Free}) \in l \end{array}\right\}$
9	`  elsedo`	$\{\ \ldots\ \}$
10	`    entry <-* (entry -.> next)`	$\{\ \ldots\ \}$
11	`).`	$\left\{\begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \\ \left(\begin{array}{c} \exists y.\exists size_y.\ size_y \geq \texttt{size} \wedge \\ (y, size_y, \texttt{Free}) \in l\ \wedge\ \texttt{entry} = y \\ \vee \\ \texttt{entry} = 0 \end{array}\right) \end{array}\right\}$

$\mathsf{Hp}(l) \overset{def}{=} \mathsf{Heap\text{-}list}\ l\ \texttt{hmStart}\ 0$, only relevant assertions are displayed, the loop invariant is boxed

Figure 4: Sketch of `findFree` proof (partial proof of `hmAlloc` in Fig. 3)

and pointers). To summarize the code, the first loop does a list traversal; when the current block is free, it enters the nested loop, that compacts the current block with its successors if the latter are also free. As emphasized in the proof sketch, the heart of the verification of `compact` is the application of the COMPACTION lemma given in Sect. 3.2.2.

4.2.3 Splitting

The role of the splitting phase is to split the block found into two blocks of appropriate sizes. This is implemented by the function `split`. Concretely, it transforms a free block into an allocated one, eventually into a couple of allocated/free blocks to save space. 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 list traversal). The post-condition asserts the existence of an allocated block of size greater than `size` (which is in general smaller than the original free block used to be):

$$\left\{\begin{array}{c}\exists l.\mathsf{Heap\text{-}list}\ l\ \mathtt{hmStart}\ 0 \wedge (x, size_x, \mathtt{Alloc}) \in l \wedge \\ \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \\ \exists y.\exists size_y.size_y \geq \mathtt{size} \wedge \\ (y, size_y, \mathtt{Free}) \in l \wedge \mathtt{entry}{=}y \wedge x \neq y\end{array}\right\}$$

$$\mathtt{(split\ entry\ size\ cptr\ sz)}$$

$$\left\{\begin{array}{c}\exists l.\mathsf{Hp}(l) \wedge (x, size_x, \mathtt{Alloc}) \in l \wedge \\ \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x) \wedge \\ \exists y.\exists size_y.size_y \geq \mathtt{size} \wedge (y, size_y, \mathtt{Alloc}) \in l \wedge \\ \mathtt{entry}{=}y \wedge x \neq y\end{array}\right\}$$

(We have omitted the predicate $\mathtt{result}=0$ for readability.)

The verification of `split` is displayed in Fig. 6. The function first computes the size of the block pointed by `entry`, and then decides if there it is large enough to be split. In this case, it builds a new header inside the data and modify the links so that the original free block is split into a couple of free blocks. Finally, the function changes the status of the block pointed by `entry` to `Alloc`. Observe that both the splitting and the change of status are handled directly by the lemmas SPLITTING and CHANGE-STATUS given in Sect. 3.2.1 and Sect. 3.2.3.

4.3 Formal Verification of Deallocation

The deallocation function `hmFree` is implemented as a list traversal; if it runs into the address passed to it, it frees the corresponding block, and fails otherwise. In addition, we must also check that it does not modify allocated blocks:

$$\left\{\begin{array}{c}\exists l.\mathsf{Heap\text{-}list}\ l\ \mathtt{hmStart}\ 0 \wedge (x, size_x, status) \in l \wedge \\ (y, size_y, status') \in l \wedge y \neq x \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x)\end{array}\right\}$$

$$\mathtt{(hmFree\ (y+2)\ entry\ cptr\ nptr\ result)}$$

$$\left\{\begin{array}{c}\exists l.\mathsf{Heap\text{-}list}\ l\ \mathtt{hmStart}\ 0 \wedge (x, size_x, status) \in l \wedge \\ (y, size_y, \mathtt{Free}) \in l \wedge y \neq x \wedge \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x)\end{array}\right\}$$

The main difficulty of this verification was to identify a bug that allows for deallocation of the terminal block (see details in Sect 5.4). The formal verification amounts to check the preservation of a loop invariant and an application of a CHANGE-STATUS lemma. The proof sketch is omitted for lack of space.

5. IMPLEMENTATION IN COQ

In this section, we comment on the practical side of our verification. We first give an overview of our implementation of separation logic in Coq. Second, we comment on the translation of C source code into Coq. Then, we illustrate the Coq implementation with some predicates and lemmas used during the verification. Last, we explain several issues in the original source code of the Topsy heap manager we found during verification.

5.1 Separation Logic in Coq

The heart of our implementation of separation logic in Coq is a module interface for heaps. This interface defines heaps as partial functions from natural numbers (type `nat` in Coq) to signed integers (type `Z` in Coq). The module is implemented using lists.

Using the module interface for heaps, we implement separation logic as defined by Reynolds in [1]. We define separating connectives by a *shallow embedding*, i.e., they are implemented as functions from heaps to the type `Prop` of Coq, enabling mixed use of separating connectives with Coq connectives. This has the advantage to simplify the writing of assertions. All Reynolds' axioms appear as Coq lemmas, proved sound. This basic implementation is extended with standard lemmas such as the frame rule and a weakest-precondition generator.

For the sake of completeness, we provide malloc and free commands, although our verification of the Topsy heap manager now provides us with an alternative solution for dynamic memory allocation. Indeed, we can consider specifications of `hmAlloc` and `hmFree` to be axioms and use them to verify programs. For example, let us consider the following sample specification:

$$\{\ \mathsf{Heap\text{-}list}\ l\ \mathtt{hmStart}\ 0 \wedge (x, 1, \mathtt{Alloc}) \in l \wedge (x \mapsto e) * T\ \}$$

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

$$\left\{\begin{array}{c}\mathsf{Heap\text{-}list}\ l\ \mathtt{hmStart}\ 0 \wedge (x, 1, \mathtt{Alloc}) \in l \wedge \\ (y > 0 \rightarrow (x \mapsto e) * (\mathtt{y} \mapsto \mathtt{v}) * T)\end{array}\right\}$$

The program allocates a new block using `hmAlloc`, stores its location into variable `y`, and stores some value `v` into the block. The post-condition asserts that, in case of successful allocation, the newly allocated block is separated from any previously allocated one. This specification is easily provable using the specification of `hmAlloc` proved in Sect. 4.2.

We develop many other reusable lemmas and tactics during our experiment. In particular, 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, let us consider the proof of the lemma SPLITTING given in Sect. 3.2.3. This lemma features three destructive updates, that are responsible in its proof for the creation of 27 sub-heaps, 16 hypotheses of equality and 13 hypotheses of 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.

The whole library is summarized in Table 1.

5.2 Translation from C Source Code

As stated in Sect. 2, we verify directly the source code of

```
Definition compact cptr nptr stts :=
```

$$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \texttt{cptr} = \texttt{hmStart} \wedge \\ \textsf{Init-array}\ (x+2)\ (list_x) \end{array} \right\}$$

```
  (* cptr points to the current block *)
  while (cptr =/= null) (
```

$$\boxed{\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \\ \textsf{Init-array}\ (x+2)\ (list_x) \wedge \exists y.\texttt{cptr} = y \\ \wedge \\ \left(\begin{array}{c} \exists sz.\exists st.\exists l_1.\exists l_2. \\ l = (l_1 +\!\!+ ((y, sz, st) :: nil) +\!\!+ l_2) \\ \vee \\ (\textsf{Heap-list}\ l\ \texttt{hmStart}\ y * \textsf{Heap-list}\ nil\ y\ 0) \\ \vee \\ y = 0 \end{array} \right) \end{array} \right\}}$$

```
   stts <-* (cptr -.> status);             { ... }
   ifte (stts == Free) thendo (            { ... }
     nptr <-* (cptr -.> next);             { ... }
```

```
      (* nptr points to the block
         next to cptr *)
      while (nptr =/= null) (
```

$$\boxed{\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \\ \textsf{Init-array}\ (x+2)\ (list_x) \wedge \exists y.\texttt{cptr} = y \\ \wedge \\ \left(\begin{array}{c} \exists sz.\exists l_1.\exists l_2. \\ l = (l_1 +\!\!+ ((y, sz, \texttt{Free}) :: nil) +\!\!+ l_2) \wedge \\ \texttt{nptr} = y+2+sz \\ \vee \\ \exists sz.\exists sz'.\exists l_1.\exists l_2. \\ l = (l_1 +\!\!+ ((y, sz, \texttt{Free}) :: \\ (y+2+sz, sz', \texttt{Alloc}) :: nil) +\!\!+ l_2) \wedge \\ \texttt{nptr} = y+2+sz \\ \vee \\ \exists sz.\exists l_1. \\ l = (l_1 +\!\!+ ((y, sz, \texttt{Free}) :: nil)) \wedge \\ \texttt{nptr} = 0 \end{array} \right) \end{array} \right\}}$$

```
        stts <-* (nptr -.> status);        { ... }
        ifte (stts == Free) thendo (       { ... }
```

```
          stts <-* (nptr -.> next);
```

$$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \\ \textsf{Init-array}\ (x+2)\ (list_x) \wedge \exists y.\texttt{cptr} = y \\ \wedge \\ \exists sz.\exists sz'.\exists l_1.\exists l_2. \\ l = (l_1 +\!\!+ ((y, sz, \texttt{Free}) :: (y+2+sz, sz', \texttt{Free}) :: nil) +\!\!+ l_2) \wedge \\ \texttt{nptr} = y+2+sz \wedge \texttt{stts} = y+4+sz+sz' \end{array} \right\}$$

COMPACTION

```
          (cptr -.> next) *<- stts;
```

$$\left\{ \begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \\ \textsf{Init-array}\ (x+2)\ (list_x) \wedge \exists y.\texttt{cptr} = y \\ \wedge \\ \exists sz.\exists sz'.\exists l_1.\exists l_2. \\ l = (l_1 +\!\!+ ((y, sz+2+sz', \texttt{Free}) :: nil) +\!\!+ l_2) \wedge \\ \texttt{nptr} = y+2+sz \wedge \texttt{stts} = y+4+sz+sz' \end{array} \right\}$$

```
          nptr <- stts                     { ... }
        ) elsedo (                         { ... }
          nptr <- null                     { ... }
        )                                  { ... }
      )                                    { ... }
   ) elsedo (                              { ... }
     skip                                  { ... }
   )                                       { ... }
   cptr <-* (cptr -.> next)                { ... }
).
```

$$\left\{ \exists l.\mathsf{Hp}(l) \wedge (x, size, \texttt{Alloc}) \in l \wedge \textsf{Init-array}\ (x+2)\ (list_x) \right\}$$

$\mathsf{Hp}(l) \overset{def}{=} \textsf{Heap-list}\ l\ \texttt{hmStart}\ 0$, only relevant assertions are displayed, loop invariants are boxed, the grayed area corresponds to a heap-list lemma application

Figure 5: Sketch of `compact` proof (partial proof of `hmAlloc` in Fig. 3)

	`Definition split entry size cptr sz :=`	$\left\{\begin{array}{c} \exists l.\ \mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \textsf{Init-array}\ (x{+}2)\ (list_x) \wedge \\ \exists y.\exists size_y.size_y \geq \texttt{size} \ \wedge\ (y, size_y, \texttt{Free}) \in l \wedge \\ \texttt{entry} = y \wedge x \neq y \end{array}\right\}$
1	`ENTRYSIZE entry sz;`	$\left\{\begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \textsf{Init-array}\ (x{+}2)\ (list_x) \wedge \\ \exists y.\exists size_y.size_y \geq \texttt{size} \ \wedge\ (y, size_y, \texttt{Free}) \in l \wedge \\ \texttt{entry} = y \wedge x \neq y \wedge \texttt{sz} = size_y \end{array}\right\}$
2	`ifte (sz >>= (size + LEFTOVER + 2) thendo (`	$\left\{\begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \textsf{Init-array}\ (x{+}2)\ (list_x) \wedge \\ \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Free}) \in l \wedge \\ \texttt{entry} = y \wedge x \neq y \wedge \texttt{sz} = size_y \end{array}\right\}$
3	`  cptr <- (entry + 2 + size);`	$\{\ \ldots\ \}$
4	`  sz <-* (entry -.> next);`	$\left\{\begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \textsf{Init-array}\ (x{+}2)\ (list_x) \wedge \\ \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Free}) \in l \wedge \\ \texttt{cptr} = \texttt{entry}{+}2{+}\texttt{size} \wedge \texttt{sz} = y{+}2{+}size_y \wedge \\ \texttt{entry} = y \wedge x \neq y \end{array}\right\}$
	SPLITTING	
5	`  (cptr -.> next) *<- sz;`	$\{\ \ldots\ \}$
6	`  (cptr -.> status) *<- Free;`	$\{\ \ldots\ \}$
7	`  (entry -.> next) *<- cptr`	$\left\{\begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \textsf{Init-array}\ (x{+}2)\ (list_x) \wedge \\ \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Free}) \in l \wedge \\ \texttt{entry} = y \wedge x \neq y \end{array}\right\}$
8	`) elsedo (`	$\{\ \ldots\ \}$
9	`  skip`	$\{\ \ldots\ \}$
10	`);`	$\left\{\begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \textsf{Init-array}\ (x{+}2)\ (list_x) \wedge \\ \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Free}) \in l \wedge \\ \texttt{entry} = y \wedge x \neq y \end{array}\right\}$
	CHANGE-STATUS	
11	`(entry -.> status) *<- Allocated.`	$\left\{\begin{array}{c} \exists l.\mathsf{Hp}(l) \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \textsf{Init-array}\ (x{+}2)\ (list_x) \wedge \\ \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Alloc}) \in l \wedge \\ \texttt{entry} = y \wedge x \neq y \end{array}\right\}$

$\mathsf{Hp}(l) \overset{def}{=} \textsf{Heap-list}\ l\ \texttt{hmStart}\ 0$, only relevant assertions are displayed, grayed areas correspond to a heap-list lemma application

Figure 6: Sketch of `split` proof (partial proof of `hmAlloc` in Fig. 3)

Script file	Contents (lines)
`util.v`	Non-standard lemmas on integers, lists (486)
`heap.v`	Modules for locations, values, and heaps (3044)
`sep.v`	Separation logic assertions, Reynolds' axioms, soundness, frame rule (2096)
`vc.v`	Weakest-precondition generator (190)
`contrib.v`	Lemmas on arrays, etc. (1310)
`sep_con_tactic.v`	Tactics to decide disjointness and equality for heaps (542)
`examples.v`	Examples (700)
	total size: 8428 lines

Table 1: Separation Logic Library

Topsy using separation logic. The programming language of separation logic we implemented in Coq is closed enough to the C programming language to enable almost direct translation (that we do by hand though). By way of example, let us comment on the sample translation of one Topsy function to its Coq counterpart. Fig. 7 displays side-by-side the original `findFree` function and its Coq implementation. There is almost a syntactic correspondence between both programs. The main difference is the translation of the C instruction `break` using an extra variable (namely, `fnd`) and a branching test in Coq. Also, we need to pass extra variables to the Coq function to represent the return value and to serve as local variables. These are trivial changes.

5.3 Illustration of Formal Verification

We illustrate the Coq implementation of specifications with some predicates and lemmas used during the verification.

Here follows the Coq definition of the Heap-list predicate defined in Sect. 3.1. It is implemented as an inductive data type, with one constructor for each disjunctive clause. Because of the shallow embedding, the original signature is extended with extra arguments `store.v` and `heap.h`, and the resulting type is `Prop`. This definition makes use of the separating connectives `(...|-->(...::...))` and `**`, corresponding respectively to the maps-to formula ($\_ \mapsto \_, \_$) and the separating conjunction $*$. This definition also makes use of another predicate `Array` whose definition is omitted for lack of space. Given the explanation of Sect. 3.1, the rest of the definition is self-explaining:

```
Inductive Heap_List :
 list (loc*nat*expr)->loc->loc->store.s->heap.h->Prop :=
  Heap_List_last: forall s next startl endl h,
  endl=0 -> next=0 -> startl>0 ->
  ((nat_e startl) |--> (Alloc::(nat_e next)::nil)) s h ->
  Heap_List nil startl endl s h
| Heap_List_trans: forall s startl endl h,
  endl>0 -> startl=endl -> Emp s h ->
  Heap_List nil startl endl s h
| Heap_List_suiv: forall
    s h next startl endl h1 h2 hd_adr hd_size hd_expr tl,
  heap.disjoint h1 h2 ->
  heap.equal h (heap.union h1 h2) ->
  hd_expr = Alloc \/ hd_expr = Free ->
  next = (startl + 2 + hd_size) ->
  startl = hd_adr ->
  startl > 0 ->
  (((nat_e startl) |--> (hd_expr::(nat_e next)::nil)) **
    (Array (startl+2) (hd_size))) s h1 ->
  Heap_List tl next endl s h2 ->
  Heap_List
   ((hd_adr, hd_size,hd_expr)::tl) startl endl s h.
```

Here follows the Coq definition of the COMPACTION lemma defined in Sect. 3.2.2 (proof omitted). This lemmas makes use of the separating implication (noted `-*`). Given the one-to-one correspondence with the original lemma, the rest of the definition is self-explaining.

```
Lemma Heap_List_compact :
 forall l1 l2 x1 sizex1 sizex2 startl s h,
  startl > 0 ->
  (Heap_List (l1 ++
     ((x1,sizex1,Free)::(x1+2+sizex1,sizex2,Free)::nil) ++
     l2) startl 0) s h ->
  (((nat_e x1 +e (int_e 1))|-> nat_e (x1+2+sizex1)) **
  (((nat_e x1 +e (int_e 1))|-> nat_e (x1+sizex1+4+sizex2)) -*
  (Heap_List (l1 ++
     ((x1, sizex1+2+sizex2, Free)::nil) ++
     l2 ) startl 0))) s h.
```

Last, we show the Coq definition of the specification for the `compact` function verified in Sect. 4.2.2. The Hoare triples is noted `{{...}}...{{...}}` and corresponds to a Coq lemma, that we need to prove in order to verify the function:

```
Lemma compact_verif :
 (* hypotheses omitted *)
{{fun s => fun h => exists l1, Heap_List l1 adr 0 s h /\
  In (x,size,Alloc) l1 /\
  eval (var_e hmStart) s = eval (nat_e adr) s /\
  eval (var_e result) s = eval null s /\
  eval (var_e cptr) s = eval (nat_e adr) s /\
  (ArrayI (x+2) lx ** TT) s h}}
(compact cptr nptr stts)
{{fun s => fun h => exists l1, Heap_List l1 adr 0 s h /\
  In (x,size,Alloc) l1 /\
  eval (var_e hmStart) s = eval (nat_e adr) s /\
  eval (var_e result) s = eval null s /\
  (ArrayI (x+2) lx ** TT) s h}}.
```

The whole verification implementation is summarized in Table 2.

Script file	Contents (lines)
`topsy_hm.v`	Heap-list definition and properties (1519)
`topsy_hmInit.v`	Initialization code, specification, and verification (401)
`topsy_hmAlloc.v`	Allocation code, specifications, and verifications (4159)
`topsy_hmFree.v`	Deallocation code, specification, and verification (1343)
`example_hmAlloc.v`	Example of Sect. 5.1 (417)
	total size: 7839 lines

Table 2: Topsy Heap Manager Verification

5.4 Issues found in the Original Source Code

In this section, we explained the issues (including bugs) that we found during verification.

5.4.1 Out of Range Initialization

When verifying the initialization function of the heap manager (Sect. 4.1), we found out 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 predicate unprovable because the

```c
static HmEntry findFree(unsigned long size)
{
 /* start is a global variable */
 HmEntry entry = start;

 while (entry != NULL) {
  if (entry->status == HM_FREED
       && ENTRYSIZE(entry) >= size) break;
  entry = entry->next;
 }
 return entry;
}
```

```
Definition findFree size entry fnd sz stts :=
 entry <- (var_e hmStart);
 stts <-* (entry -.> status);
 fnd <- (int_e 0);
 (while ((var_e entry =/= null) &&& (var_e fnd =/= (int_e 1))) (
  stts <-* (entry -.> status);
  (ENTRYSIZE entry sz);
  (ifte ((var_e stts == Free) &&&
           (var_e sz >>= nat_e size)) thendo
    fnd <- (int_e 1)
   elsedo
    (entry <-* (entry -.> next)))
)).
```

Figure 7: Code Translation from C (on the left) to Coq (on the right)

latter holds for a fixed are 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 the one we are using for verification.

5.4.2 *Useless Operations During Allocation*

When verifying the allocation function (Sect. 4.2), we found several useless operations that suggested immediate improvements.

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`.

Another 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.

5.4.3 *Deallocation of the Terminal Block*

When verifying the deallocation function (Sect. 4.3), 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 sample scenario in Fig. 8. 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 predicate that enforces the terminal block to be marked as allocated. We fixed this problem by adding a test over the `next` field of the block to be deallocated in the deallocation function.

6. CONCLUSION

In this paper, we formally specified and verified the heap manager of the Topsy operating system using separation logic inside the Coq proof assistant. The verification approach proved very effective since it enables us to find bugs in the original source code. In addition, this use case led us to develop a Coq library of lemmas and tactics that are reusable in the context of other experiments.

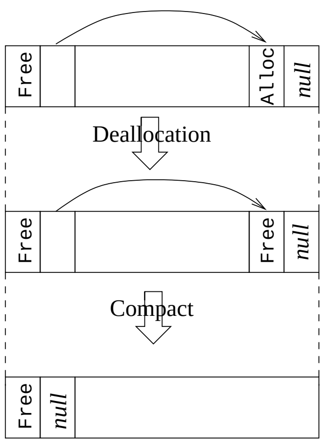

Figure 8: A scenario that illustrates the `hmFree` issue

6.1 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 [8]. 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. 5.4. Another difference between both allocators is that the Topsy heap manager is a real allocation facility in the sense that the allocation function is self-content (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 Isabelle [9]. 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.

6.2 Future Work

Portions of code that only deal with assignments and branching instructions should be handled automatically. For that purpose, we plan to interface our Coq implementation of separation logic with existing work on automation of verification of separation logic or alias types [10, 11].

7. 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://web.yl.is.s.u-tokyo.ac.jp/~affeldt/seplog`. Work in progress.

[7] 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).*

[8] 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.

[9] 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.

[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] Toshiyuki Maeda and Akinori Yonezawa. Writing practical memory management code with a strictly typed assembly language. In *3rd Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE 2006).* Jan. 2006.