

Reusable verification of a copying collector

Magnus O. Myreen

Computer Laboratory, University of Cambridge, UK

Abstract. Garbage collectors are very hard to implement correctly due to their low-level manipulation of memory. In this paper, we construct a copying garbage collector which we have proved to be functionally correct. Our verification proof is structured as a sequence of refinements to aid clarity and proof reuse; it is the first to map implementations into three different machine languages and, unlike some noteworthy published proofs, our verified implementations of memory allocation handle termination and the ‘out-of-memory’ case properly. The work presented here has been developed in the HOL4 theorem prover.

1 Introduction

Garbage collectors are important cornerstones of any implementation of a functional programming language and most object-oriented programming languages. They are hard to implement correctly due to their low-level manipulation of memory and very hard to test for faults since the property of a correct execution is rather abstract: every execution must produce a heap which is,

- isomorphic to the original heap, i.e. must be equivalent to the original heap modulo renaming of addresses, and
- minimal in the sense that it must not contain unnecessary heap elements, i.e. elements that are not reachable from root addresses.

In this paper we present the construction of a copying collector that we have proved formally correct, i.e. for which we know that the above properties hold for every possible execution of the verified code. A sample of the verified ARM, x86 and PowerPC code is listed in Figure 6.

There are numerous publications on the topic of garbage collector verification and some very impressive recent work on proving the correctness of assembly and C-like implementations of copying garbage collectors. These proofs, which we will describe in Section 2, are unfortunately tied to specific programming logics and mix reasoning for why the algorithm is correct (how heap isomorphism is achieved) with implementation specific details (such as how specific heap elements are represented in memory). As a result published proofs are cumbersome to adapt to new settings.

This paper attempts to remedy these shortcomings by presenting a verification proof which has been carefully designed to be reusable for any stop-the-world copying collector. The main contributions of this paper are as follows:

- We present a verification proof which is sufficient for proving low-level implementations correct and at the same time independent of any particular programming logic. The proof cleanly separates reasoning about the correctness of the core algorithm from all implementation specific details.
- Data refinement is used to map our proofs down to the level of verified ARM, x86 and PowerPC machine code. This is the first paper to construct garbage collectors that have been proved correct with respect to realistic models of machine languages.
- These collectors are the first verified collectors to be used as mere building blocks in a much larger verification effort: our verified garbage collectors are part of verified implementations of Lisp.¹
- Our verified implementations of allocation handle the ‘out-of-memory’ case properly. They terminate with an error message in case there is an insufficient amount of memory available after a full garbage collection. This is in contrast to noteworthy published work [3, 12] which only prove partial correctness of code that diverges in the ‘out-of-memory’ case.

The work presented here has been developed inside the HOL4 theorem prover.²

2 Related work

There are number of publications on the topic of specification and verification of garbage collection routines, e.g. [5–7, 11, 16]. However, few have proved copying collectors correct with respect to detailed models of realistic execution environments. Notable exceptions are work by Birkedal et al. [3], McCreight et al. [12], and Hawblitzel and Petrank [9].

Birkedal et al. used a version of separation logic to verify, on paper it seems, the correctness of a C-like program implementing the Cheney algorithm for a stop-the-world copying collector. McCreight et al. developed a general framework for collector proofs, in Coq, and verified MIPS-like code for several different collector algorithms, including two copying collector, one of which was incremental. The allocators verified by McCreight et al. and Birkedal et al. enter an infinite loop in case the heap is full after a complete collection cycle. In contrast, the allocators verified here have been proved to terminate: they terminate in the ‘out-of-memory’ case by jumping to code that can produce an appropriate error message.

In some very impressive recent work by Hawblitzel and Petrank, a copying collector and a mark-and-sweep collector, with competitive real-world performance, were verified mechanically. They did not use a theorem prover, instead they used the Boogie verification generator which links to the Z3 SMT solver. This system proved their x86-like implementations correct automatically given low-level code decorated with a substantial amount of annotations. They did

¹ Our work on verified Lisp interpreters has been published before [13], but the proof of its garbage collectors is published here for the first time.

² Our proofs are available at hol.sf.net in SVN under `HOL/examples/machine-code`.

not prove termination but were able to run a suite of benchmarks which showed that their collectors have competitive performance compared with other collectors. Similar proofs might be possible in theorem provers in the future, even in LCF-style provers, as their support for SMT solvers is starting to mature [4].

None of the above mentioned verified copying collectors have been used as a building block inside a verified run-time, i.e. it has not been tried whether the resulting correctness theorems are usable as components in further formal developments. Our verified garbage collectors have been used inside verified Lisp interpreters [13]. However, these Lisp interpreters fall short of being practically useful at present due to the restrictive subset of Lisp which they implement. As far as we know, the VLISP project [8] is the only project which has successfully built a usable verified run-time which included a verified garbage collector. The VLISP verification consists of lengthy pen-and-paper-style proofs.

Novel work by Benton on specification and verification of a memory allocator [1] should also be mentioned. Benton verified, using Coq, an implementation of an allocator in an invented assembly language. Instead of using conventional unary predicates for describing program properties, he used quantified binary relations and stated program properties in terms of contextual equivalence. This allowed him to show that his allocator transfers ownership of memory states to the client program and that the client program is parametrised by the allocator. The allocator specification presented here does not provide such clean logical separation, instead the allocator always ‘owns’ the allocated memory and the client is forced to view the heap as an abstraction of the real memory. However, it remains unclear whether the cost of adding these extra features to the specifications is worth the effort since Benton’s proofs seem to have been frustratingly hard work, as he commented in a separate note [2]. However, this might have been caused the fact that this was the first time Benton seriously used a theorem prover, instead of any feature in his approach that might have made it ill-suited for the automation provided by modern theorem provers.

3 Method

This paper presents verified garbage collectors which have been constructed through a sequence of refinements, starting from a high-level specification and going down to concrete machine code. We use the following refinement layers.

- L1. We start with a specification which states what a full garbage collection is to achieve, namely, to remove unreachable heap elements and rename addresses in a consistent manner. At this level of abstraction, which we call L1, garbage collection is a single transition.
- L2. At the second level of abstraction, we provide an abstract implementation of L1 as the transitive closure of a step relation. We prove that any complete execution of these steps implements L1. This proof, which is only 300 lines long, verifies the core idea behind the correctness of copying collection.

- L3. At the third level of abstraction, we refine the non-deterministic implementation from L2 into a deterministic function which operates over a more concrete notion of memory: heap elements now have sizes and temporary reference cells are stored in memory alongside heap elements.
- L4. At the next level, we introduce actual implementation types, e.g. addresses become real machine addresses (aligned 32-bit words). We also subdivide memory accesses into individual 32-bit memory reads and writes.
- L5. At the lowest level of abstraction, we have the concrete ARM, x86 and PowerPC machine code. These implementations are automatically synthesised from the L4 implementation using a previously developed compiler which produces a proof of correspondence for each compilation.

Each refinement is proved correct with respect to the layers above it. The sizes of the manual proofs are approximately 300, 800 and 700 lines for L1/L2, L2/L3 and L3/L4, respectively. In total these proofs are less than half of the length of the proofs described in McCreight et al. [12].

3.1 Specification – L1

We start by formalising what we mean by garbage collection in terms of a heap represented as a finite partial (\rightarrow) mapping h where addresses are natural numbers. The domain of h is a finite subset of \mathbb{N} and the codomain of h consists of pairs (as, d) where as is a list of addresses and d is some data. The type of heap h is defined as the following using two type variables $null$ and $data$. We let null pointers be of arbitrary type so that later refinements can store data inside null pointers, which is often done in practice.

$$\mathbb{N} \rightarrow (\mathbb{N} + null) \text{ list} \times data$$

We define the set of reachable addresses as the smallest inductively defined set such that a is reachable whenever a is a root or a is pointed to by some reachable element b . Let $\text{set } as$ be the set of non-null addresses in the list as .

$$\frac{a \in \text{set } roots}{a \in \text{reach}(h, roots)} \quad \frac{a \in \text{set } as \wedge h(b) = (as, data) \wedge b \in \text{reach}(h, roots)}{a \in \text{reach}(h, roots)}$$

The most abstract notion of garbage collection can now simply be defined as a function filter which restricts (1) the domain of a heap mapping h to only elements reachable from the root nodes.

$$\text{filter}(h, roots) = (h \upharpoonright (\text{reach}(h, roots)), roots)$$

In this paper we consider the verification of a copying garbage collector, i.e. one which must also have the right to rearrange heap elements. For this we define a function rename which updates all addresses in h by a given function $f : \mathbb{N} \rightarrow \mathbb{N}$. Let $\text{map } f$ update all non-null addresses of a list by application of f .

$$\begin{aligned} \text{domain}(\text{rename } f \ h) &= \text{image } f \ (\text{domain } h) \\ (\text{rename } f \ h)(f(x)) &= (\text{map } f \ as, d) \quad \text{whenever } h(x) = (as, d) \end{aligned}$$

Using `rename`, we define a valid rearrangement as a relation $\xrightarrow{\text{translate}}$ which relates two heaps whenever one heap can be converted into the other by applying a global swap function f , i.e. a function such that $f \circ f = \text{id}$.

$$\frac{f \circ f = \text{id}}{(h, \text{roots}) \xrightarrow{\text{translate}} (\text{rename } f \ h, \text{map } f \ \text{roots})}$$

We can now define that a garbage collection is a relation $\xrightarrow{\text{gc}}$, which filters out unreachable heap elements and renames the addresses.

$$x \xrightarrow{\text{gc}} y = (\text{filter } x) \xrightarrow{\text{translate}} y$$

3.2 Abstract implementation – L2

Our first refinement is to split the single-step implementation of garbage collection from L1 into a sequence of small step updates, and prove that the transitive closure of this step update implements L1. The step relation $\xrightarrow{\text{step}}$ is defined, below, using three rules that operate over a state which consists of:

- h — the heap, a finite partial mapping, mentioned above for L1,
- x — address set: completely processed heap elements,
- y — address set: moved elements with pointers to not-yet-moved elements,
- z — address set: elements that are still to be moved,
- f — a function which records where elements have been moved: $\mathbb{N} \rightarrow \mathbb{N}$

The main operation performed by the collector is to move an element $a \in z$ to a new unused location $b \notin \text{domain } h$. The source and target location must not have been part of earlier move operations, i.e. we must have $f(a) = a \wedge f(b) = b$. The new address b is inserted into the set of moved but not complete elements y , the addresses as stored at $h(a)$ are inserted into the set of addresses to be moved z and the swap of addresses $a \leftrightarrow b$ is recorded in function f .

$$\frac{a \in z \wedge b \notin \text{domain } h \wedge f(a) = a \wedge f(b) = b \wedge h(a) = (as, d)}{(h, x, y, z, f) \xrightarrow{\text{step}} (h[b \mapsto (as, d)] - \{a\}, x, y \cup \{b\}, z \cup \text{set } as, f[a \mapsto b][b \mapsto a])}$$

Addresses a that have been moved, i.e. for which $f(a) \neq a$, but which are still in the set of addresses that are to be moved z can be deleted from set z .

$$\frac{a \in z \wedge f(a) \neq a}{(h, x, y, z, f) \xrightarrow{\text{step}} (h, x, y, z - \{a\}, f)}$$

Once all of the addresses as , stored at some heap location $a \in y$, have been removed from set z , i.e. $\text{set } as \cap z = \{\}$, then we can finalise this heap element $h(a)$ by updating the addresses as with mapping f and moving address a from set y to set x .

$$\frac{a \in y \wedge h(a) = (as, d) \wedge \text{set } as \cap z = \{\}}{(h, x, y, z, f) \xrightarrow{\text{step}} (h[a \mapsto (\text{map } f \ as, d)], x \cup \{a\}, y - \{a\}, z, f)}$$

Correctness. The formal connection between L1 and L2 is summed up in the following theorem, which states that any execution of the transitive closure of the step relation $\xrightarrow{\text{step}^*}$, which starts with $x = y = \{\}$ and z initialised to the root addresses and ends in a state $y = z = \{\}$, is in fact a correct execution of the garbage collector $\xrightarrow{\text{gc}}$, defined for L1. The domain of the resulting heap h_2 is restricted to the set of moved addresses x , i.e. $h_2 \downarrow x$.

$$\begin{aligned} & \forall h \ h_2 \ roots \ x \ f. \\ & (h, \{\}, \{\}, \text{set roots}, \text{id}) \xrightarrow{\text{step}^*} (h_2, x, \{\}, \{\}, f) \wedge \text{ok_heap}(h, \text{roots}) \implies \\ & (h, \text{roots}) \xrightarrow{\text{gc}} (h_2 \downarrow x, \text{map } f \ \text{roots}) \end{aligned}$$

$$\begin{aligned} \text{where } \text{ok_heap}(h, \text{roots}) &= \text{pointers } h \cup \text{set roots} \subseteq \text{domain } h \\ \text{pointers } h &= \{ x \mid \exists a \ as \ d. x \in \text{set } as \wedge h(a) = (as, d) \} \end{aligned}$$

Invariant. Instead of delving into the details of our proof, we present the invariant inv which allows us to prove the above theorem.

$$\forall x \ s \ t. \ \text{inv } x \ s \wedge s \xrightarrow{\text{step}} t \implies \text{inv } x \ t$$

The full definition of our invariant is shown in Figure 1. It took approximately one week to get this invariant completely right. We believe that this invariant is sufficiently independent of the lower-level implementations L3, L4, L5 to be of use also in verification proofs of significantly different versions of L3, L4 and L5. A complete understanding of the invariant is not necessary to follow the rest of this paper. However, for those who are interested: line 0 defines an abbreviation old to denote the set of addresses that were originally the domain of h ; line 1 states that x and y are disjoint and that f must be its own inverse; line 2 states that all pointers from within h restricted to addresses x must point to heap elements in x or y ; line 3 ensures that all pointers outside of h restricted to x , i.e. inside h restricted to the complement of x , are in the set of old addresses;

$$\begin{aligned} & \text{inv}(h_0, \text{roots})(h, x, y, z, f) = \\ 0 \quad & \text{let } \text{old} = (\text{domain } h \cup \{ a \mid f(a) \neq a \}) - (x \cup y) \text{ in} \\ 1 \quad & (x \cap y = \{\}) \wedge (f \circ f = \text{id}) \wedge \\ 2 \quad & \text{pointers}(h \downarrow x) \subseteq x \cup y \wedge \\ 3 \quad & \text{pointers}(h \downarrow x^c) \subseteq \text{old} \wedge \\ 4 \quad & \text{pointers}(h \downarrow y) \cup \text{set roots} \subseteq \text{image } f(x \cup y) \cup z \subseteq \text{reach}(h_0, \text{roots}) \wedge \\ 5 \quad & (\forall a. a \in z \implies \text{if } f(a) = a \text{ then } a \in \text{old} \text{ else } f(a) \in x \cup y) \wedge \\ 6 \quad & (\forall a. f(a) \neq a \implies \neg(a \in x \cup y \iff f(a) \in x \cup y)) \wedge \\ 7 \quad & (\forall a. a \in x \cup y \iff f(a) \neq a \wedge a \in \text{domain } h) \wedge \\ 8 \quad & \text{domain } h = \text{image } f(\text{domain } h_0) \wedge \\ 9 \quad & (\forall a \ as \ d. f(a) \in \text{domain } h \wedge h(f(a)) = (as, d) \implies \\ & \quad h_0(a) = \text{if } f(a) \in x \text{ then } (\text{map } f \ as, d) \text{ else } (as, d)) \end{aligned}$$

Fig. 1. The invariant used for proving a connection between L1 and L2.

line 4 guarantees that elements from x , y and z are reachable; lines 5-6 state when f is allowed to point into $x \cup y$; line 7 states that $x \cup y$ is the set of new addresses; lines 8-9 ensure that f relates h to h_0 .

Our proofs using this invariant are relatively small, the proof connecting L1 and L2 is approximately 300 lines long. We achieve this brevity by stating the invariant in terms of sets and set operations, which leads to subgoals that are easily discharged using a standard first-order prover [10].

3.3 Implementation with memory – L3

The next refinement introduces a memory which makes the memory layout concrete. At this level of abstraction intermediate reference cells, called **Ref** elements, keep a record of renaming function f in memory alongside data stored in **Block** elements. The memory, we call it m , is a mapping from \mathbb{N} to a data-type with type constructors:

Block (as, l, d) — a block of length l which contains addresses as and data d
Ref a — a reference cell containing the address a
Emp — an empty location or ‘don’t care’

Memory m is a correct representation of h and f whenever, for any a :

$$\begin{aligned} m(a) = \text{Block } (h(a)) & \quad \text{if } a \in \text{domain } h \\ m(a) = \text{Ref } (f(a)) & \quad \text{if } a \notin \text{domain } h \text{ and } f(a) \neq a \\ m(a) = \text{Emp} & \quad \text{if } a \notin \text{domain } h \text{ and } f(a) = a \end{aligned}$$

Here the type variable $data$ in the type of h has been instantiated to $\mathbb{N} \times data$ to make $h(a)$ a triple of type: $(\mathbb{N} + null) \text{ list} \times \mathbb{N} \times data$. We will refer to the above relation between m , h and f as $\text{ref_mem}(h, f, m)$.

As mentioned above, each $m(a) = \text{Block}(as, l, data)$ stores a length l . Based on this we have a well-formedness criteria which states that the next l memory locations $m(a+1), m(a+2), \dots, m(a+l)$ must be **Emp**.

$$\text{empty}(a, l) \ m = \forall i \in \mathbb{N}. i < l \implies m(a + i + 1) = \text{Emp}$$

We formalise this criterion as an inductively defined relation $\text{part_heap}(a, b) \ m \ k$ which states that the memory locations in the range $a \dots b$ (we write $a \dots b$ to mean $\{ n \in \mathbb{N} \mid a \leq n \wedge n < b \}$) form a well-formed heap containing blocks of data that have a combined length of k .

$$\begin{array}{c} \text{part_heap}(a, a) \ m \ 0 \\ \hline m(a) = \text{Block}(as, l, data) \wedge \text{empty}(a, l) \ m \wedge \text{part_heap}(a + l + 1, b) \ m \ k \\ \hline \text{part_heap}(a, b) \ m \ (l + 1 + k) \\ \hline (m(a) = \text{Ref } i \vee m(a) = \text{Emp}) \wedge \text{part_heap}(a + 1, b) \ m \ k \\ \hline \text{part_heap}(a, b) \ m \ k \end{array}$$

```

move (RHS  $n, j, m$ ) = (RHS  $n, j, m$ )
move (LHS  $a, j, m$ ) = case  $m(a)$  of
  Ref  $i \rightarrow$  (LHS  $i, j, m$ )
| Block ( $as, l, d$ )  $\rightarrow$ 
  let  $m = m[a \mapsto \text{Ref } j]$  in
  let  $m = m[j \mapsto \text{Block } (as, l, d)]$  in
  (LHS  $j, j + l + 1, m$ )

move_list ( $[], j, m$ ) = ( $[], j, m$ )
move_list ( $r::rs, j, m$ ) =
  let ( $r, j, m$ ) = move ( $r, j, m$ ) in
  let ( $rs, j, m$ ) = move_list ( $rs, j, m$ ) in
  ( $r::rs, j, m$ )

readBlock (Block  $x$ ) =  $x$ 
cut ( $i, j$ )  $m = \lambda k. \text{ if } i \leq k \wedge k < j \text{ then } m\ j \text{ else Emp}$ 

loop ( $i, j, m$ ) =
  if  $i = j$  then ( $i, m$ ) else
  let ( $as, l, d$ ) = readBlock ( $m\ i$ ) in
  let ( $as, j, m$ ) = move_list ( $as, j, m$ ) in
  let  $m = m[i \mapsto \text{Block } (as, l, d)]$  in
  loop ( $i + l + 1, j, m$ )

collector ( $roots, b, i, e, b_2, e_2, m$ ) =
1  let ( $b_2, e_2, b, e$ ) = ( $b, e, b_2, e_2$ ) in
2  let ( $roots, j, m$ ) = move_list ( $roots, b, m$ ) in
3  let ( $i, m$ ) = loop ( $b, j, m$ ) in
4  let  $m = \text{cut } (b, i)\ m$  in
   ( $roots, b, i, e, b_2, e_2, m$ )

```

Fig. 2. Implementation at level L3.

Finally, the memory is split into two disjoint spaces, the so called *to-space* and *from-space*. During execution heap blocks are moved from the from-space into the to-space. The to-space consists of locations $b\dots e$ and the from-space are at locations $b_2\dots e_2$.

Our implementation of copying collection is listed in Figure 2. The top-level function is called *collector*. We give a brief overview of how it works here. Line 1 flips the meaning of the to-space and the from-space, i.e. what used to be the to-space is now the from-space. All elements are assumed to lie within the from-space at this stage. Line 2 then moves all heap elements pointed to by root addresses into the to-space. Line 3 starts a loop which moves all other reachable elements from the from-space into the to-space. Finally line 4 overwrites the entire from-space with ‘don’t care’ elements *Emp*.

Correctness. We have proved that our implementation at level L3, listed in Figure 2, is correct with respect to our definition at level L1, via L2. In order to state this formally, we define `ok_mem_heap` to assert what suffices as a valid initial/final state of memory as follows. Line 1: the heap must be split into two disjoint semi-heaps, $b \dots e$ and $b_2 \dots e_2$, of equal size, with an index i into heap $b \dots e$. Line 2: the memory inside of $b \dots i$ must form a well-formed heap and all other parts of the heap are empty. And line 3: the memory m must be related to some well-formed heap h according to `ref_mem` and `ok_heap`.

$$\begin{aligned}
& \text{ok_mem_heap } (h, \text{roots}) (b, i, e, b_2, e_2, m) = \\
1 & \quad b \leq i \leq e \wedge b_2 \leq e_2 \wedge e_2 - b_2 = e - b \wedge (e < b_2 \vee e_2 < b) \wedge \\
2 & \quad (\exists k. \text{part_heap } (b, i) m k) \wedge (\forall a. a \notin b \dots i \implies m(a) = \text{Emp}) \wedge \\
3 & \quad \text{ref_mem } (h, \text{id}) m \wedge \text{ok_heap } (h, \text{roots})
\end{aligned}$$

The guarantee for the final state is slightly stronger: the final state satisfies `part_heap` $(b, i) m (i - b)$. Let `precise` $(b, i, \dots, m) = \text{part_heap } (b, i) m (i - b)$.

The correctness of our L3 implementation is now stated as the following theorem: for any valid initial state x , which is related to high-level state (h, roots) , an execution of `collector` produces a state y , for which there exists a corresponding abstract heap h_2 such that our top-level definition of garbage collection ($\xrightarrow{\text{gc}}$) relates the initial heap h to the new heap h_2 .

$$\forall h \text{ roots roots}_2 x y.$$

$$\begin{aligned}
& \text{ok_mem_heap } (h, \text{roots}) x \wedge \text{collector } (\text{roots}, x) = (\text{roots}_2, y) \implies \\
& \exists h_2. \text{ok_mem_heap } (h_2, \text{roots}_2) y \wedge (h, \text{roots}) \xrightarrow{\text{gc}} (h_2, \text{roots}_2) \wedge \text{precise } y
\end{aligned}$$

The presence of `precise` y is important for proving allocation correct, Section 4.

Invariant. We will again not go into details of the correctness proof, but instead only explain the invariant which was used for the proof. Our invariant, called `mem_inv`, was used for proving the following property of the main loop:

$$\begin{aligned}
& \text{mem_inv } (h_0, \text{roots}_0, h, f) (b, i, j, e, b_2, e_2, m, \text{pointers } (h \upharpoonright (i \dots j))) \wedge \\
& \text{loop } (i, j, m) = (i_2, m_2) \implies \\
& \exists h_2 f_2. \text{mem_inv } (h_0, \text{roots}_0, h_2, f_2) (b, i_2, i_2, e, b_2, e_2, m_2, \{\}) \wedge \\
& \quad j \leq i_2 \wedge \forall a. f(a) \neq a \implies f_2(a) = f(a)
\end{aligned}$$

The definition of our invariant `mem_inv` is listed in Figure 3. The main idea behind this invariant should be clear from lines 5 and 6. They state that memory m is a refinement of (h, f) and that (h, f) is related, through the reflexive-transitive closure of $\xrightarrow{\text{step}}$, to an initial state (h_0, roots_0) which satisfies `ok_heap`. Lines 2–4 are less interesting; they ensure that the memory is correctly organised. Line 1 states that the heap is split into two semi-heaps, and that i and j are indexes in the to-heap.

3.4 Implementation with concrete types – L4

The previous refinements layer, called L3, produced an implementation with memory and concrete memory layout of the heap. However, L3 made no commitment to how memory elements, `Block` and `Ref`, are to be represented in actual

```

mem_inv (h0, roots0, h, f) (b, i, j, e, b2, e2, m, z) =
1   b ≤ i ≤ j ≤ e ∧ (e < b2 ∨ e2 < b) ∧
2   (∀a. a ∉ b2...e2 ∪ b...j ⇒ m(a) = Emp) ∧
3   part_heap (b, i) m (i - b) ∧ part_heap (i, j) m (j - i) ∧
4   (∃k. part_heap (b2, e2) m k ∧ k ≤ e - j) ∧
5   ref_mem (h, f) m ∧ ok_heap (h0, roots0) ∧
6   (h0, {}, {}, set roots0, id)  $\xrightarrow{\text{step}^*}$  (h, domain h ∩ (b...i), domain h ∩ (i...j), z, f)

```

Fig. 3. The invariant which relates implementations L3 with L2.

memory. In this layer, called L4, we make all types concrete: addresses and data are bit strings and memory is a partial function from machine addresses (aligned 32-bit words) to 32-bit words.

We choose to represent each **Block** as a sequence of 32-bit words. The header word contains a 22-bit number n which contains the length of the payload (a list of 32-bit words: w_1, w_2, \dots, w_n), an 8-bit field for data called the *tag*, and a 1-bit field b which indicates whether the payload consists of data, in case $b = 0$, or addresses, in case $b = 1$. The header is followed by the payload.

$$[n.tag.1.b], [w_1], [w_2], [w_3], \dots, [w_n]$$

We also allow data to be stored into ‘null-addresses’. A 32-bit word appearing in the place of an address but which is not word-aligned (i.e. not a multiple of four) is considered to be data. The reason for why this works with the abstraction layers above is that we left the type of null addresses as a type variable *null*, as mentioned at the start.

Our garbage collector is implemented at level L4 as a functional program which uses only concrete machine types. An extract of the lengthy implementation is listed in Figure 4, which shows the L4 implementation `mx_move` of the function `move` from level L3 (i.e. the code which copies a **Block** element from the from-heap to the to-heap). The first test, $r_2 \& 3 \neq 0$, tests whether the address is a real address (not a null-address). The second test, $r_4 \& 3 = 0$, checks whether the word that was read from memory is a header of a **Block** element or a references cell `Ref`. If the first word is the header of a **Block** element then we copy over the payload using the tail-recursive function `mc_move_loop`.

Correctness. The correctness of the L4 implementation is stated concisely in the following theorem: if level L3 state y is related to level L4 state z then an execution of the L3 function `collector` on y is related to L4 function `mc_collector` applied to z . The definition of `ok_mc_heap` will be presented below.

$$\forall x y z. \text{ok_mc_heap } x y z \implies \text{ok_mc_heap } x (\text{collector } y) (\text{mc_collector } z)$$

```

mc_move_loop (r2, r3, r4, g) =
  if r4 = 0 then (r2, r3, r4, g) else
    let r5 = g(r2) in
    let r4 = r4 - 1 in
    let r2 = r2 + 4 in
    let g = g[r3 ↦ r5] in
    let r3 = r3 + 4 in
    mc_move_loop (r2, r3, r4, g)

mc_move (r1, r2, r3, g) =
  if (r2 & 3 ≠ 0) then (r1, r3, g) else
    let r4 = g(r2) in
    if r4 & 3 = 0 then
      let g = g[r1 ↦ r4] in
      (r1, r3, g)
    else
      let g = g[r1 ↦ r3] in
      let g = g[r3 ↦ r4] in
      let g = g[r2 ↦ r3] in
      let r4 = r4 ≫ 10 in
      let r3 = r3 + 4 in
      let r2 = r2 + 4 in
      let (r2, r3, r4, g) = mc_move_loop (r2, r3, r4, g) in
      (r1, r3, g)

```

Fig. 4. Part of the implementation at level L4.

Invariant. In order to keep our statements and proofs clean and concise even at this low-level of abstraction, we will use some light-weight separation logic [15] for memory assertions. We need the separating conjunction $*$, which we define over sets: $p * q$ is true for set s if s can be partitioned into two sets t and u such that p holds for t and q holds for u .

$$(p * q) s = \exists t u. p t \wedge q u \wedge t \cup u = s \wedge t \cap u = \{\}$$

Now let `fun2set` map a partial function to a set of pairs, let `one` (x, y) assert the value of a pair in such a set, and let `emp` assert that the set is empty:

$$\begin{aligned}
\text{fun2set } g &= \{ (a, g(a)) \mid a \in \text{domain } g \} \\
\text{one } (x, y) &= \lambda s. (s = \{(x, y)\}) \\
\text{emp} &= \lambda s. (s = \{\}) \\
\langle b \rangle &= \lambda s. (s = \{\}) \wedge b
\end{aligned}$$

With these we can define `ref`, in Figure 5, which allows us to state that segments of L3 memory m are present in L4 memory g , e.g. the following line states that memory locations $b \dots e$ and $b_2 \dots e_2$ from memory m are represented correctly in L4 memory g , i.e. both halves of the heap are correctly represented.

$$(\text{ref } (b, e) m * \text{ref } (b_2, e_2) m * p) (\text{fun2set } g)$$

```

ref_heap_addr (RHS  $n$ ) =  $2 \times n + 1$ 
ref_heap_addr (LHS  $a$ ) =  $4 \times a$ 

one_list  $a$  [] = emp
one_list  $a$  ( $x :: xs$ ) = one ( $a, x$ ) * one_list ( $a+4$ )  $xs$ 

header ( $n, b, tag$ ) =  $1024 \times n + 4 \times tag + 2 +$  (if  $b$  then 1 else 0)

ref_aux  $a$  Emp =  $\exists x. \text{one} (a, x)$ 
ref_aux  $a$  (Ref  $n$ ) = one ( $a, 4 \times n$ )
ref_aux  $a$  (Block ( $xs, l, (tag, b, ys)$ )) =
  let  $zs =$  (if  $b$  then map ref_heap_addr  $xs$  else  $ys$ ) in
  one ( $a, \text{header} (length\ zs, b, tag)$ ) * one_list ( $a+4$ )  $zs$ 

ref_inc  $a$  Emp = 1
ref_inc  $a$  (Ref  $n$ ) = 1
ref_inc  $a$  (Block ( $xs, l, d$ )) =  $1 + l$ 

ref ( $a, e$ )  $m =$ 
  if  $e \leq a$  then  $\langle a = e \rangle$  else
  ref_aux ( $4 \times a$ ) ( $m(a)$ ) * ref ( $a + \text{ref\_inc} (m(a)), e$ )  $m$ 

```

Fig. 5. Part of the invariant which relates L3 to L4.

The main part of each proof is to show that this type of ref-relationship is maintained between m and g throughout execution of the two implementations.

There is still a further well-formedness criteria for memory m that needs to be mentioned: all lists in **Block** elements must be of reasonable size and the length field l must correspond to the actual payload:

```

ok_memory  $m =$ 
   $\forall a\ l\ xs\ b\ t\ ys.$ 
   $m(a) = \text{Block} (xs, l, (b, t, ys)) \implies$ 
  length  $ys < 2^{22} \wedge$  length  $xs < 2^{22} \wedge$ 
  if  $t$  then  $l = \text{length } xs$  else  $l = \text{length } ys \wedge xs = []$ 

```

3.5 Machine-code implementations – L5

The final leap from low-level functional implementations (L4) to concrete machine code (L5) would be very tedious to prove manually. To avoid a manual proof we use a previously developed compiler to produce correct machine code automatically from the functional implementation at level L4.

Our compiler is not verified, but instead produces a proof for each compilation run, i.e. the compiler steers the theorem prover to a proof which certifies that the input function is correctly executed by the generated machine code. For example, the following theorem is produced when the compiler compiles function `mc_move` from Figure 4 into ARM machine code. This theorem certifies that any execution of the machine code which starts at a state where (r_1, r_2, r_3, g) describes the values of registers 1, 2, 3 and memory, terminates in a state where

tst r2, #3	test ecx, 3	andi. 0, 2, 3
bne L0	jne L0	bne L0
ldr r4, [r2]	mov ebx, [ecx]	lwz 4, 0(2)
tst r4, #3	test ebx, 3	andi. 0, 4, 3
streq r4, [r1]	jne L2	bne L2
beq L0	mov [eax], ebx	stw 4, 0(1)
str r3, [r1]	jmp L0	b L0
str r4, [r3]	L2: mov [eax], edx	L2: stw 3, 0(1)
str r3, [r2], #4	mov [edx], ebx	stw 4, 0(3)
mov r4, r4, LSR #10	mov [ecx], edx	stw 3, 0(2)
add r3, r3, #4	shr ebx, 10	srawi 4, 4, 10
L1: cmp r4, #0	add edx, 4	addi 3, 3, 4
beq L0	add ecx, 4	addi 2, 2, 4
ldr r5, [r2]	L1: cmp ebx, 0	L1: cplwi 4, 0
sub r4, r4, #1	je L0	beq L0
add r2, r2, #4	mov edi, [ecx]	lwz 5, 0(2)
str r5, [r3]	dec ebx	addi 4, 4, -1
add r3, r3, #4	add ecx, 4	addi 2, 2, 4
b L1	mov [edx], edi	stw 5, 0(3)
L0:	add edx, 4	addi 3, 3, 4
	jmp L1	b L1
	L0:	L0:

Fig. 6. Verified ARM, x86 and PowerPC code, respectively, for `mc_move` from Figure 4.

`mc_move` (r_1, r_2, r_3, g) accurately describes the value of registers 1, 3 and memory. This is stated in terms of a machine-code Hoare triple [14], and conditioned on an automatically generated precondition `mc_move_pre`.

$$\begin{aligned}
& \forall r_1 r_2 r_3 g p. \\
& \text{mc_move_pre } (r_1, r_2, r_3, g) \implies \\
& \{ r1 \ r1 * r2 \ r2 * r3 \ r3 * r4 \ _ * r5 \ _ * \text{memory } g * s * \text{pc } p \} \\
& p : \text{E3120003 } 1\text{A000010 } \text{E5924000 } \text{E3140003 } 0\text{5814000 } 0\text{A00000C } \text{E5813000} \\
& \quad \text{E5834000 } \text{E5823000 } \text{E1A04524 } \text{E2833004 } \text{E2822004 } \text{E3540000 } 1\text{5925000} \\
& \quad 1\text{2444001 } 1\text{2822004 } 1\text{5835000 } 1\text{2833004 } 1\text{AFFFFFF8} \\
& \{ \text{let } (r_1, r_3, g) = \text{mc_move } (r_1, r_2, r_3, g) \text{ in} \\
& \quad r1 \ r1 * r2 \ _ * r3 \ r3 * r4 \ _ * r5 \ _ * \text{memory } g * s * \text{pc } (p+76) \}
\end{aligned}$$

We have used our proof-producing compiler to compile the top-level L4 function `mc_collector` into ARM, x86 and PowerPC code. Each of the resulting certificate theorems are conditioned on a precondition `mc_collector_pre`. This precondition simply asserts that each memory access was done properly, no load/store to unaligned addresses. We have proved that these preconditions are always met:

$$\forall x y z. \text{ok_mc_heap } x y z \implies \text{mc_collector_pre } z$$

4 Using the verified garbage collectors

In this section we will briefly explain how the verified garbage collectors have been used as components in the construction of verified interpreters for Lisp [13].

For our Lisp case study, we define allocation of a `cons` cell as follows at abstraction level L3. Note that it is tempting to define `allocate_cons` as recursive function to avoid writing `has_space` twice, but that would result in an unsatisfactory infinite loop when allocation runs out of memory (and allow for a trick if only partial correctness is to be proved).

```

has_space (roots, b, i, e, b2, e2, m) = 3 ≤ e - i
alloc_fail (r1::r2::roots, b, i, e, b2, e2, m) = (nil::r2::roots, b, i, e, b2, e2, m)
alloc_ok (r1::r2::roots, b, i, e, b2, e2, m) =
  (LHS i::r2::roots, b, i+3, e, b2, e2, m[i ↦ Block ([r1, r2], 2, (T, 0, []))])
allocate_cons state =
  if has_space state then alloc_ok state else
  let state = collector state in
  if has_space state then alloc_ok state else alloc_fail state

```

We write a similar L4 implementation and from these generate L5 implementations. The correctness theorems used in the Lisp case study for `cons` allocation are stated as follows. The following theorems use a heap assertion `lisp` which states that Lisp s-expressions $v_1 \dots v_6$ are stored in a heap with a capacity for l `cons` cells. Allocation of a new `cons` cell is guaranteed to be successful if the size of the six root s-expressions is strictly less than the heap limit l :

```

size v1 + size v2 + size v3 + size v4 + size v5 + size v6 < l ⇒
{ lisp (v1, v2, v3, v4, v5, v6, l) * pc p }
p : E50A3018 E50A4014 E50A5010 E50A600C ... E51A8004 E51A7008
{ lisp (cons v1 v2, v2, v3, v4, v5, v6, l) * pc (p + 324) }

```

We also have a different theorem describing all executions: all executions of the allocator will terminate either in a successful state, or jump to a special program point (`lisp_out_of_memory`) which generates an error message.

```

{ lisp (v1, v2, v3, v4, v5, v6, l) * pc p }
p : E50A3018 E50A4014 E50A5010 E50A600C ... E51A8004 E51A7008
{ lisp (cons v1 v2, v2, v3, v4, v5, v6, l) * pc (p + 324) ∨ lisp_out_of_memory }

```

5 Conclusions and future work

We aimed for a clear, understandable and reusable verification. By structuring the verification as a sequence of refinements, our work separates reasoning about the algorithm from implementation level details and as a result made each part of the proof (refinement step) clearly focused on separate aspects of the verification. The fact that only the lowest level of abstraction (L5) is tied to specific programming logics and program semantics ought to aid proof reuse.

Why did we not verify a generational garbage collector? The short answer is that we did not need one. However, we believe a generational collector is only a refinement step away (from implementation L3): the idea is to treat all pointers to previous generations as if they were pure data stored in null pointers.

Acknowledgements. I would like to thank Mike Gordon and the anonymous reviewers for helpful suggestions regarding presentation. This work was partially supported by EPSRC Research Grant EP/G007411/1.

References

1. Nick Benton. Abstracting allocation: The new new thing. In *Computer Science Logic (CSL)*. Springer-Verlag, 2006.
2. Nick Benton. Machine obstructed proof (abstract). ACM SIGPLAN Workshop on Mechanizing Metatheory, 2006.
3. L. Birkedal, N. Torp-Smith, and J. Reynolds. Local reasoning about a copying garbage collector. In *Principles of programming languages (POPL)*. ACM, 2004.
4. Sascha Böhme and Tjark Weber. Fast LCF-style proof reconstruction for Z3. In *Theorem Proving in Higher Order Logics (TPHOLs)*. Springer, 2010. To Appear.
5. Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.
6. Georges Gonthier. Verifying the safety of a practical concurrent garbage collector. In *CAV, LNCS*, pages 462–465. Springer, 1996.
7. David Gries. An exercise in proving parallel programs correct. *Commun. ACM*, 20(12):921–930, 1977.
8. Joshua Guttman, John Ramsdell, and Mitchell Wand. VLISP: A verified implementation of Scheme. *Lisp and Symbolic Computation*, 8(1/2):5–32, 1995.
9. Chris Hawblitzel and Erez Petrank. Automated verification of practical garbage collectors. In *Principles of Programming Languages (POPL)*. ACM, 2009.
10. Joe Hurd. First-order proof tactics in higher-order logic theorem provers. In *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, NASA/CP-2003-212448 in NASA Technical Reports, pages 56–68, 2003.
11. Paul B. Jackson. Verifying a garbage collection algorithm. In *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS, pages 225–244. Springer, 1998.
12. Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. A general framework for certifying garbage collectors and their mutators. In *Programming Language Design and Implementation (PLDI)*, pages 468–479. ACM, 2007.
13. Magnus O. Myreen and Michael J. C. Gordon. Verified LISP implementations on ARM, x86 and PowerPC. In *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS, pages 359–374. Springer, 2009.
14. Magnus O. Myreen, Konrad Slind, and Michael J. C. Gordon. Machine-code verification for multiple architectures – An application of decompilation into logic. In *Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2008.
15. John Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*. IEEE Computer Society, 2002.
16. David M. Russinoff. A mechanically verified incremental garbage collector. *Formal Asp. Comput.*, 6(4):359–390, 1994.