A Verified Generational Garbage Collector for CakeML

Downloaded from: https://research.chalmers.se, 2026-10-14 17:15 UTC

Citation for the original published paper (version of record):
Sandberg Eriksson, A., Myreen, M., Åman Pohjala, J. (2019). A Verified Generational Garbage Collector for CakeML. Journal of Automated Reasoning, 63(2): 463-488.
http://dx.doi.org/10.1007/s10817-018-9487-z

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

Journal of Automated Reasoning (2019) 63:463–488
https://doi.org/10.1007/s10817-018-9487-z

A Verified Generational Garbage Collector for CakeML

Adam Sandberg Ericsson[1] · **Magnus O. Myreen**[1] · **Johannes Åman Pohjola**[2]

Received: 27 February 2018 / Accepted: 10 October 2018 / Published online: 3 November 2018
© The Author(s) 2018

Abstract
This paper presents the verification of a generational copying garbage collector for the CakeML runtime system. The proof is split into an algorithm proof and an implementation proof. The algorithm proof follows the structure of the informal intuition for the generational collector's correctness, namely, a partial collection cycle in a generational collector is the same as running a full collection on part of the heap, if one views pointers to old data as non-pointers. We present a pragmatic way of dealing with ML-style mutable state, such as references and arrays, in the proofs. The development has been fully integrated into the in-logic bootstrapped CakeML compiler, which now includes command-line arguments that allow configuration of the generational collector. All proofs were carried out in the HOL4 theorem prover.

Keywords Interactive theorem proving · Formal methods · Garbage collection · Compiler verification

1 Introduction

High-level programming languages such as ML, Haskell, Java, Javascript and Python provide an abstraction of memory which removes the burden of memory management from the application programmer. The most common way to implement this memory abstraction is to use garbage collectors in the language runtimes. The garbage collector is a routine which is invoked when the memory allocator finds that there is not enough free space to perform allocation. The collector's purpose is to produce new free space. It does so by traversing the data in memory and deleting data that is unreachable from the running application. There are two classic algorithms: mark-and-sweep collectors mark all live objects and delete the others; copying collectors copy all live objects to a new heap and then discard the old heap and its dead objects.

Since garbage collectors are an integral part of programming language implementations, it is essential that they are performant. As a result, there have been numerous improvements

✉ Johannes Åman Pohjola
johannes.amanpohjola@data61.csiro.au

[1] Chalmers University of Technology, Gothenburg, Sweden

[2] Data61/CSIRO, Sydney, Australia

to the classic algorithms mentioned above. There are variants of the classic algorithms that make them incremental (do a bit of garbage collection often), generational (run the collector only on recent data in the heap), or concurrent (run the collector as a separate thread alongside the program).

This paper's topic is the verification of a generational copying collector for the CakeML compiler and runtime system [18]. The CakeML project has produced a formally verified compiler for an ML-like language called CakeML. The compiler produces binaries that include a verified language runtime, with supporting routines such as an arbitrary precision arithmetic library and a garbage collector. One of the main aims of the CakeML compiler project is to produce a verified system that is as realistic as possible. This is why we want the garbage collector to be more than just an implementation of one of the basic algorithms.

Contributions

- To the best of our knowledge, this paper presents the first completed formal verification of a generational garbage collector. However, it seems that the CertiCoq project [1] is in the process of verifying a generational garbage collector.
- We present a pragmatic approach to dealing with mutable state, such as ML-style references and arrays, in the context of implementation and verification of a generational garbage collector. Mutable state adds a layer of complexity since generational collectors need to treat pointers from old data to new data with special care. The CertiCoq project does not include mutable data, i.e. their setting is simpler than ours in this respect.
- We describe how the generational algorithm can be verified separately from the concrete implementation. Furthermore, we show how the proof can be structured so that it follows the intuition of informal explanations of the form: a partial collection cycle in a generational collector is the same as running a full collection on part of the heap if one views pointers to old data as non-pointers.
- This paper provides more detail than any previous CakeML publication on how algorithm-level proofs can be used to write and verify concrete implementations of garbage collectors for CakeML, and how these are integrated into the full CakeML compiler and runtime. The updated in-logic bootstrapped compiler comes with new command-line arguments that allow configuration of the generational garbage collector.

Differences from Conference Version This paper extends a previous conference paper [4] by providing more detailed explanations, a new section on timing the new GC, and stronger theorems about the GC algorithms. Explanations have been expanded in Sects. 3.2, 3.3, and, in particular, Sect. 4. The new timing section (Sect. 5) compares the generational garbage collector with the previous non-generational version. In Sect. 3.3, the correctness theorems have been strengthened to cover GC completeness, i.e., that a collection cycle collects all garbage.

2 Approach

In this section, we give a high-level overview of the work and our approach to it. Subsequent sections will cover these topics in more detail.

Algorithm-Level Modelling and Verification

- The intuition behind the copying garbage collection is important in order to understand this paper. Section 3.1 provides an explanation of the basic Cheney copying collector

algorithm. Section 3.2 continues with how the basic algorithm can be modified to run as a generational collector. It also describes how we deal with mutable state such as ML-style references and arrays.
- Section 3.3 describes how the algorithm has been modelled as HOL functions. These algorithm-level HOL functions model memory abstractly, in particular we use HOL lists to represent heap segments. This representation neatly allows us to avoid awkward reasoning about potential overlap between memory segments in the algorithm-level proofs. It also works well with the separation logic we use later to map the abstract heaps to their concrete memory representations, in Sect. 4.2.
- Section 3.4 defines the main correctness property, gc_related, that any garbage collector must satisfy: for every pointer traversal that exists in the original heap from some root (i.e. program variable), there must be a similar pointer traversal possible in the new heap.
- A generational collector can run either a partial collection, which collects only some part of the heap, or a full collection of the entire heap. We show that the full collection satisfies gc_related. To show that a run of the partial collector also satisfies gc_related, we exploit a simulation argument that allows us to reuse the proofs for the full collector. Intuitively, a run of the partial collector on a heap segment h simulates a run of the full collector on a heap containing only h. Section 3.4 provides some details on this.

Implementation and Integration into the CakeML Compiler

- The CakeML compiler goes through several intermediate languages on the way from source syntax to machine code. The garbage collector is introduced gradually in the intermediate languages DataLang (abstract data), WordLang (machine words, concrete memory, but abstract stack) and StackLang (more concrete stack).
- The verification of the compiler phase from DataLang to WordLang specifies how abstract values of DataLang are mapped to instantiations of the heap types that the algorithm-level garbage collection operates over, Sect. 4.1. We prove that gc_related implies that from DataLang's point of view, nothing changes when a garbage collector is run.
- For the verification of the DataLang to WordLang compiler, we also specify how each instantiation of the algorithm-level heap types maps into WordLang's concrete machine words and memory, Sect. 4.2. Here we implement and verify a *shallow embedding* of the garbage collection algorithm. This shallow embedding is used as a primitive by the semantics of WordLang.
- Further down in the compiler, the garbage collection primitive needs to be implemented by a *deep embedding* that can be compiled with the rest of the code. This happens in StackLang, where a compiler phase attaches an implementation of the garbage collector to the currently compiled program and replaces all occurrences of Alloc by a call to the new routine. Implementing the collector in StackLang is tedious because StackLang is very low- level—it comes after instruction selection and register allocation. However, the verification proof is relatively straight-forward since the proof only needs to show that the StackLang deep embedding computes the same function as the shallow embedding mentioned above.
- Finally, the CakeML compiler's in-logic bootstrap needs updating to work with the new garbage collection algorithm. The bootstrap process itself does not need much updating, illustrating the resilience of the bootstrapping procedure to such changes. We extend the bootstrapped compiler to recognise command-line options specifying which garbage collector is to be generated: `-gc=none` for no garbage collector; `-gc=simple` for the previous non-generational copying collector; and `-gc=gen`*size* for the generational collector described in the present paper. Here *size* is the size of the nursery generation in number

of machine words. With these command-line options, users can generate a binary with a specific instance of the garbage collector installed.

Mechanised Proofs The development was carried out in HOL4. The sources are available at http://code.cakeml.org/. The algorithm and its proofs are under `compiler/backend/gc`; the first implementation at the word-level, i.e. the shallow embedding, is in `compiler/backend/proofs/word_gcFunctionsScript.sml` and its verification is in `compiler/backend/proofs/data_to_word_gcProofScript.sml`; the StackLang deep embedding is in `compiler/backend/stack_allocScript.sml`; its verification is in `compiler/backend/proofs/stack_allocProofScript.sml`.

Terminology The *heap* is the region of memory where heap elements are allocated and which is to be garbage collected. A *heap element* is the unit of memory allocation. A heap element can contain pointers to other heap elements. The collection of all program visible variables is called the *roots*.

3 Algorithm Modelling and Verification

Garbage collectors are complicated pieces of code. As such, it makes sense to separate the reasoning about algorithm correctness from the reasoning about the details of its more concrete implementations. Such a split also makes the algorithm proofs more reusable than proofs that depend on implementation details. This section focuses on the algorithm level.

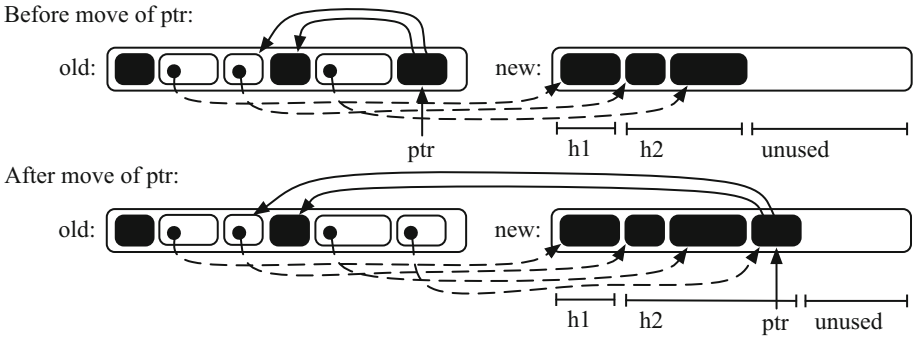
3.1 Intuition for Basic Algorithm

Intuitively, a Cheney copying garbage collector copies the live elements from the current heap into a new heap. We will call the heaps old and new. In its simplest form, the algorithm keeps track of two boundaries inside the new heap. These split the new heap into three parts, which we will call h1, h2, and unused space.

Throughout execution, all pointers in the heap segment h1 will point to the new heap, and all pointers in heap segment h2 will only point to the old heap, i.e. pointers that are yet to be processed.

The algorithm's most primitive operation is to move a pointer ptr, and the data element d that ptr points at, from the old heap to the new one. The move primitive's behaviour depends on whether d is a *forwarding pointer* or not. A forwarding pointer is a heap element with a special tag to distinguish it from other heap elements. Forwarding pointers will only ever occur in the heap if the garbage collector puts them there; between collection cycles, they are never present nor created.

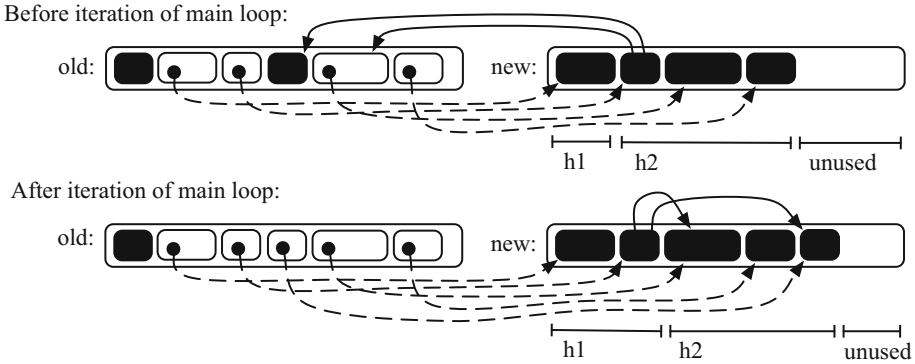
If d is not a forwarding pointer, then d will be copied to the end of heap segment h2, consuming some of the unused space, and ptr is updated to be the address of the new location of d. A forwarding pointer to the new location is inserted at the old location of d, namely at the original value of ptr. We draw forwarding pointers as hollow boxes with dashed arrows illustrating where they point. Solid arrows that are irrelevant for the example are omitted in these diagrams.



If d is already a forwarding pointer, the move primitive knows that this element has been moved previously; it reads the new pointer value from the forwarding pointer, and leaves the memory unchanged.

The algorithm starts from a state where the new heap consists of only free space. It then runs the move primitive on each pointer in the list of roots. This processing of the roots populates h2.

Once the roots have been processed, the main loop starts. The main loop picks the first heap element from h2 and applies the move primitive to each of the pointers that that heap element contains. Once the pointers have been updated, the boundary between h1 and h2 can be moved, so that the recently processed element becomes part of h1.



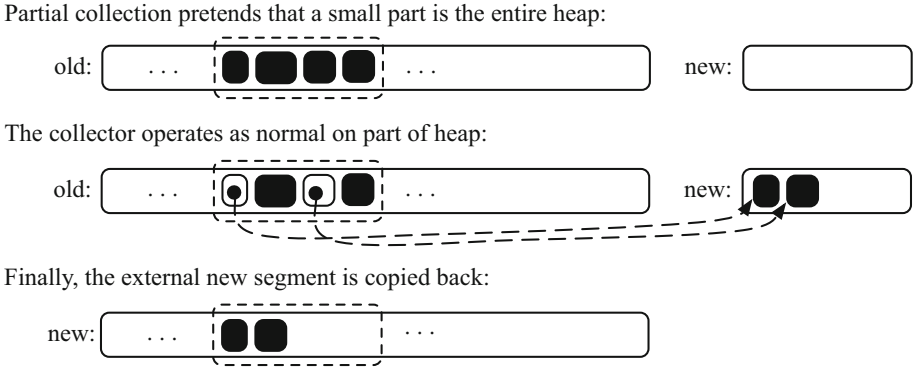
This process is repeated until h2 becomes empty, and the new heap contains no pointers to the old heap. The old heap can then be discarded, since it only contains data that is unreachable from the roots. The next time the garbage collector runs, the previous old heap is used as the new heap.

3.2 Intuition for Generational Algorithm

Generational garbage collectors switch between running *full* and *partial* collection cycles. In a partial collection cycle, we run the collector only on part of the heap. The motivation is that new data tends to be short-lived while old data tends to stay live. By running the collector on new data only, one avoids copying around old data unnecessarily. Full collection cycles consider the entire heap; hence they are slower, but can potentially free up more space.

The intuition is that a partial collection focuses on a small segment of the full heap and ignores the rest, but operates as a normal full collection on this small segment. The cleanup

after a partial collection cycle differs from a Cheney copying collector: of course, we cannot simply discard the old heap, since it may still contain live data outside the current segment. Rather, we copy the new segment back into its previous location in the old heap.



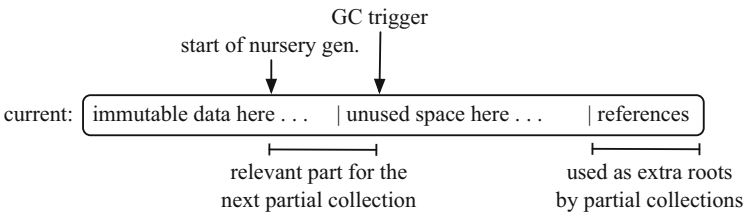
For the partial collection to work we need:

(a) the partial algorithm to treat all pointers to the outside (old data) as non-pointers, in order to avoid copying old data into its new memory region.
(b) that outside data does not point into the currently collected segment of the heap, because the partial collector should be free to move around and delete elements in the segment it is working on without looking at the heap outside.

In ML programs, most data is immutable, which means that old data cannot point at new data. However, ML programs also use references and arrays (henceforth both will be called references) that are mutable. References are usually used sparingly, but are dangerous for a generational garbage collector because they can point into the new data from old data.

Our pragmatic solution is to make sure immutable data is allocated from the bottom of the heap upwards, and references are allocated from the top downwards, i.e. the memory layout as in the diagram below. (The conventional solution, which does not impose such a layout on the heap, is described further down.)

The following diagram also shows our use of a *GC trigger pointer* which indicates the end of the current nursery generation. Any allocation that tries to grab memory past the GC trigger pointer causes the GC to run. By default, after a GC run, the GC trigger pointer is placed the distance of the nursery generation into the unused part of the heap. If the allocation asks for more space than the length of the nursery size, then the trigger pointer is placed further into the unused part of the heap in order to guarantee success of the allocation.



To satisfy requirement (a), full collection cycles must maintain this memory layout. Hence our full collection is the simple garbage collection algorithm described in the previous section, modified so that it copies references to the end of the new heap and immutable data to the start. The algorithm assumes that we have a way to distinguish references from other data

elements; the CakeML compiler delivers on this assumption by way of tag bits. To satisfy requirement (b), we make each run of the partial collection algorithm treat the references as roots that are not part of the heap.

Our approach means that references will never be collected by a partial collection. However, they will be collected when the full collection is run.

Full collections happen if there is a possibility that the partial collector might fail to free up enough space, i.e. if the amount of unused space prior to collection is less than the amount of new memory requested. Note that there is no heuristic involved here: if there is enough space for the allocation between the GC trigger pointer and the actual end of the heap, then a partial collection is performed since the partial collection will, in this case, always be able to move the GC trigger pointer a sufficient distance towards the beginning of the references for the requested allocation to be successful.

One could run a partial collection regardless of whether it might fail to find enough memory, and then run a full collection if it fails. We decided against this because scanning all of the roots twice would potentially be costly and if the complete heap is so close to running out of space that a partial collection might fail, then a full collection is likely to run very soon anyway.

Reconfiguring or Switching GC at Runtime With our approach, it is possible to reconfigure or switch GC at runtime. One can at any point switch from the generational to the non-generational because the non-generational version does not care about where the references are. Switching from the non-generational to the generational GC can be done by running a full collection cycle of the generational corrector on the heap. This works because the full collection cycle of the generational collector moves the references to the top of the heap regardless of where they were before.

The Conventional Solution to Mutable References Most implementations of ML do not impose a heap layout where references are at one end of a heap. Instead they use write barriers on reference updates. In the simplest form, an approach based on write barriers executes code at every reference update that conses the name of the updated reference to a list of references that have been updated since the last run of the generational garbage collector. With such a record of which references have been updated, the partial collector can use just a subset of the references (only the relevant ones) as extra roots. This is in contrast to our approach which treats all references as extra roots always.

We decided to go with the simple but unconventional approach of imposing a heap layout because it is simpler for the verification proofs, but also because we do not want to allocate write barriers on reference updates. Maintaining a list of recently updated references is not needed for the non-generational collector, and we want to have the same mutator code for both the generational and non-generational collectors in order to be able to switch between them.

3.3 Formalisation

The algorithm-level formalisation represents heaps abstractly as lists, where each element is of type heap_element. The definition of heap_element is intentionally somewhat abstract, with type variables α used for the type of data that can be attached to pointers and data elements, and β which represents tags carried by pointers. We use this flexibility to verify the partial collector for our generational version, in the next section.

Addresses are of type heap_address and can either be an actual pointer with some data attached, or a non-pointer Data. A heap element can be unused space (Unused), a forwarding pointer (ForwardPointer), or actual data (DataElement).

$$
\begin{aligned}
&\alpha \text{ heap_address } = \text{ Pointer num } \alpha \mid \text{Data } \alpha \\
&(\alpha,\ \beta) \text{ heap_element } = \\
&\quad \text{Unused num} \\
&\mid \text{ForwardPointer num } \alpha \text{ num} \\
&\mid \text{DataElement } (\alpha \text{ heap_address list}) \text{ num } \beta
\end{aligned}
$$

Each heap element carries its concrete length in machine words (minus one). The length (minus one) is part of each element for the convenience of defining a length function, el_length. No heap element has a zero length.

$$
\begin{aligned}
&\text{el_length (Unused } l) = l + 1 \\
&\text{el_length (ForwardPointer } n\ d\ l) = l + 1 \\
&\text{el_length (DataElement } xs\ l\ data) = l + 1
\end{aligned}
$$

The natural number (type num in HOL) in Pointer values is an offset from the start of the relevant heap. We define a lookup function heap_lookup that fetches the content of address a from a heap xs:

$$
\begin{aligned}
&\text{heap_lookup } a\ [] = \text{None} \\
&\text{heap_lookup } a\ (x{::}xs) = \\
&\quad \text{if } a = 0 \text{ then Some } x \\
&\quad \text{else if } a < \text{el_length } x \text{ then None} \\
&\quad \text{else heap_lookup } (a - \text{el_length } x)\ xs
\end{aligned}
$$

The generational garbage collector has two main routines: gen_gc_full which runs a collection on the entire heap including the references, and gen_gc_partial which runs only on part of the heap, treating the references as extra roots. Both use the record type gc_state to represent the heaps. In a state s, the old heap is in s.heap, and the new heap comprises the following fields: s.h1 and s.h2 are the heap segments h1 and h2 from before, s.n is the length of the unused space, and s.r2, s.r1 are for references what s.h1 and s.h2 are for immutable data [1]; $s.ok$ is a boolean representing whether s is a well-formed state that has been arrived at through a well-behaved execution. It has no impact on the behaviour of the garbage collector; its only use is in proofs, where it serves as a convenient trick to propagate invariants downwards in refinement proofs. Intuitively, adding a conjunct to $s.ok$ is similar in spirit to making an assert statement in a program.

Figure 1 shows the HOL function implementing the move primitive for the partial generational algorithm. It follows what was described informally in the section above: it does nothing when applied to a non-pointer, or to a pointer that points outside the current generation. When applied to a pointer to a forwarding pointer, it follows the forwarding pointer but leaves the heap unchanged. When applied to a pointer to some data element d, it inserts d at the end of h2, decrements the amount of unused space by the length of d, and inserts, at the old location of d, a forwarding pointer to its new location. When applied to an invalid pointer (i.e. to an invalid heap location, or to a location containing unused space) it does nothing except set the ok field of the resultant state to false; we prove later that this never happens.

[1] For technical reasons, the segments s.r1 and s.r2 each comprise two distinct heap segments that are treated slightly differently; the presentation here abstracts away from this detail.

```
gen_gc_partial_move conf state (Data d) = (Data d, state)
gen_gc_partial_move conf state (Pointer ptr d) =
  let ok = state.ok ∧ ptr < heap_length state.heap in   – Assert: ptr points into old heap
    if ptr < conf.gen_start ∨ conf.refs_start ≤ ptr then   – Is ptr not within generation?
      (Pointer ptr d, state with ok := ok)   – If so, no change to state or heap
    else
      case heap_lookup ptr state.heap of   – Follow ptr into old heap
        None ⇒ (Pointer ptr d, state with ok := F)   – Assert: ptr points at something...
      | Some (Unused _) ⇒ (Pointer ptr d, state with ok := F)   – ... that is not empty space
      | Some (ForwardPointer ptr' _ l') ⇒ (Pointer ptr' d, state)   – Follow forwarding pointers
      | Some (DataElement xs l dd) ⇒
          let ok = ok ∧ l + 1 ≤ state.n ∧ ¬conf.isRef dd;
              n = state.n − (l + 1);   – Decrement free space in new heap
              h2 = state.h2 ++ [DataElement xs l dd];   – Move data to the new heap
              (heap,ok) = write_forward_pointer ptr state.heap state.a d ok;
              a = state.a + l + 1 in   – Increment end of h2
            (Pointer state.a d,
             state with ⟪h2 := h2; n := n; a := a; heap := heap; ok := ok⟫)
```

Fig. 1 The algorithm implementation of the move primitive for gen_gc_partial

The with notation is for record update: for example, s with ⟪ok := T; h2 := l⟫ denotes a record that is as s but with the ok and h2 fields updated to the given values.

The HOL function gen_gc_full_move implements the move primitive for the full generational collection. Its definition, which is shown in Fig. 2, is similar to gen_gc_partial_move, but differs in two main ways. First, gen_gc_full_move does not consider generation boundaries. Second, in order to maintain the memory layout it must distinguish between pointers to references and pointers to immutable data, allocating references at the end of the new heap's unused space and immutable data at the beginning. This is implemented by the case split on $conf$.isRef, which is an oracle for determining whether a data element is a reference or not. It is kept abstract for the purposes of the algorithm-level verification; when we integrate our collector into the CakeML compiler, we instantiate $conf$.isRef with a function that inspects the tag bits of the data element.

gen_gc_partial_move does not need to consider pointers to references, since generations are entirely contained in the immutable part of the heap.

The algorithms for an entire collection cycle consist of several HOL functions in a similar style; the functions implementing the move primitive are the most interesting of these. The main responsibility of the others is to apply the move primitive to relevant roots and heap elements, following the informal explanations in previous sections.

3.4 Verification

For each collector (gen_gc_full and gen_gc_partial), we prove that they do not lose any live elements. We formalise this notion with the gc_related predicate shown below. If a collector can produce $heap_2$ from $heap_1$, there must be a map f such that gc_related f $heap_1$ $heap_2$. The intuition is that if there was a heap element at address a in $heap_1$ that was retained by the collector, the same heap element resides at address $f\ a$ in $heap_2$.

The conjuncts of the following definition state, respectively: that f must be an injective map into the set of valid addresses in $heap_2$; that its domain must be a subset of the valid addresses into $heap_1$; and that for every data element D at address $i \in$ domain f, every address reachable from D is also in the domain of f, and $f\ i$ points to a data element that is exactly D with all its pointers updated according to f.

```
gen_gc_full_move conf state (Data d) = (Data d, state)
gen_gc_full_move conf state (Pointer ptr d) =
 case heap_lookup ptr state.heap of
   None ⇒ (Pointer ptr d, state with ok := F)
 | Some (Unused _) ⇒ (Pointer ptr d, state with ok := F)
 | Some (ForwardPointer ptr' _ _) ⇒ (Pointer ptr' d, state)
 | Some (DataElement xs l dd) ⇒
   let ok = state.ok ∧ l + 1 ≤ state.n; n = state.n − (l + 1)
   in
    if conf.isRef dd then   – Does the data have a reference tag?
     let r2 = DataElement xs l dd::state.r2;   – References to the front of r2
     (heap,ok) = write_forward_pointer ptr state.heap (state.a + n) d ok
     in
      (Pointer (state.a + n) d,
       state with ⟪r2 := r2; n := n; heap := heap; ok := ok⟫)
    else   – Other data to the end of h2
     let h2 = state.h2 ++ [DataElement xs l dd];
     (heap,ok) = write_forward_pointer ptr state.heap state.a d ok;
     a = state.a + l + 1
     in
      (Pointer state.a d,
       state with ⟪h2 := h2; n := n; a := a; heap := heap; ok := ok⟫)
```

Fig. 2 The algorithm implementation of the move primitive for gen_gc_full

```
gc_related f heap1 heap2 ⟺
 injective (apply f) (domain f) { a | isSomeDataElement (heap_lookup a heap2) } ∧
 (∀ i. i ∈ domain f ⇒ isSomeDataElement (heap_lookup i heap1)) ∧
 ∀ i xs l d.
   i ∈ domain f ∧ heap_lookup i heap1 = Some (DataElement xs l d) ⇒
    heap_lookup (apply f i) heap2 =
     Some (DataElement (addr_map (apply f) xs) l d) ∧
    ∀ ptr u. mem (Pointer ptr u) xs ⇒ ptr ∈ domain f
```

Proving a gc_related-correctness result for gen_gc_full, as below, is a substantial task that requires a non-trivial invariant, similar to the one we presented in earlier work [12]. The main correctness theorem is as follows. We will not give further details of its proofs in this paper; for such proofs see [12].

```
⊢ roots_ok roots heap ∧ heap_ok heap conf.limit ⇒
   ∃ state f.
     gen_gc_full conf (roots,heap) = (addr_map (apply f) roots,state) ∧
     domain f = reachable_addresses roots heap ∧
     heap_length (state.h1 ++ state.r1) = heap_length (heap_filter (domain f) heap)
     ∧gc_related f heap (state.h1 ++ heap_expand state.n ++ state.r1)
```

The theorem above can be read as saying: if all roots are pointers to data elements in the heap (abbreviated roots_ok), if the heap has length *conf*.limit, and if all pointers in the heap are valid non-forwarding pointers back into the heap (abbreviated heap_ok), then a call to gen_gc_full results in a state that is gc_related via a mapping f whose domain is exactly all the addresses that are reachable from *roots* in the original *heap*. The theorem above, furthermore, states that the length of the used parts of the heap (i.e. *state*.h1 and *state*.r1) is the same as

the sum of the lengths of all reachable data elements in the original heap. The latter property means that a full collection cycle is complete in the sense that it collects all garbage.

The more interesting part is the verification of gen_gc_partial, which we conduct by drawing a formal analogy between how gen_gc_full operates and how gen_gc_partial operates on a small piece of the heap. The proof is structured in two steps:

1. We first prove a simulation result: running gen_gc_partial is the same as running gen_gc_full on a state that has been modified to pretend that part of the heap is not there and the references are extra roots.
2. We then show a gc_related result for gen_gc_partial by carrying over the same result for gen_gc_full via the simulation result (without the completeness conjunct, since partial cycles are not complete).

For the simulation result, we instantiate the type variables in the gen_gc_full algorithm so that we can embed pointers into Data blocks. The idea is that encoding pointers to locations outside the current generation as Data causes gen_gc_full to treat them as non-pointers, mimicking the fact that gen_gc_partial does not collect there. The type we use for this purpose is defined as follows:

$$(\alpha,\ \beta)\ \text{data_sort}\ =\ \text{Protected}\ \alpha\ |\ \text{Real}\ \beta$$

and the translation from gen_gc_partial's pointers to pointers on the pretend-heap used by gen_gc_full in the simulation argument is:

$$\begin{aligned}
&\text{to_gen_heap_address}\ conf\ (\text{Data}\ a) = \text{Data}\ (\text{Real}\ a)\\
&\text{to_gen_heap_address}\ conf\ (\text{Pointer}\ ptr\ a) =\\
&\quad \text{if}\ ptr < conf.\text{gen_start}\ \text{then Data}\ (\text{Protected}\ (\text{Pointer}\ ptr\ a))\\
&\quad \text{else if}\ conf.\text{refs_start} \le ptr\ \text{then Data}\ (\text{Protected}\ (\text{Pointer}\ ptr\ a))\\
&\quad \text{else Pointer}\ (ptr - conf.\text{gen_start})\ (\text{Real}\ a)
\end{aligned}$$

Similar to_gen functions, elided here, encode the roots, heap, state and configuration for a run of gen_gc_partial into those for a run of gen_gc_full. We prove that for every execution of gen_gc_partial starting from an ok state, and the corresponding execution of gen_gc_full starting from the encoding of the same state through the to_gen functions, encoding the results of the former with to_gen yields precisely the results of the latter.

Initially, we made an attempt to do the gc_related proof for gen_gc_partial using the obvious route of manually adapting all loop invariants and proofs for gen_gc_full into invariants and proofs for gen_gc_partial. This soon turned out to overly cumbersome; hence we switched to the current approach because it seemed more expedient and more interesting. As a result, the proofs for gen_gc_partial are more concerned with syntactic properties of the to_gen-encodings than with semantic properties of the collector. The syntactic arguments are occasionally quite tedious, but we believe this approach still leads to more understandable and less repetitive proofs.

Finally, note that gc_related is the same correctness property that we use for the previous copying collector; this makes it straightforward to prove that the top-level correctness theorem of the CakeML compiler remains true if we swap out the garbage collector.

3.5 Combining the Partial and Full Collectors

An implementation that uses the generational collector will mostly run the partial collector and occasionally the full one. At the algorithm level, we define a combined collector and leave

it up to the implementation to decide when a partial collection is to be run. The choice is made visible to the implementation by having a boolean input do_partial to the combined function. The combined function will produce a valid heap regardless of the value of do_partial.

Our CakeML implementation (next section) runs a partial collection if the allocation will succeed even if the collector does not manage to free up any space, i.e., if there is already enough space on the other side of the GC trigger pointer before the GC starts (Sect. 3.2).

4 Implementation and Integration into the CakeML Compiler

The concept of garbage collection is introduced in the CakeML compiler at the point where a language with unbounded memory (DataLang) is compiled into a language with a finite memory (WordLang). In this phase of the compiler, we have to prove that the garbage collector automates memory deallocation and implements the illusion of an unbounded memory.

A key lemma is the proof that running WordLang's allocation routine (which includes the GC) preserves all important invariants and that the resulting WordLang state relates to the same DataLang state extended with the requested new space (or alternatively giving up with a NotEnoughSpace exception). This theorem is shown in Fig. 9. It is used as a part in the correctness proof of the DataLang-to-WordLang phase of the compiler: theorem shown in Fig. 10.

Proving the key lemma about allocation requires several layers of invariants in the form of state- and value-relations and proofs about these. These invariants are the topic of the following subsections. The last part of this section will also briefly describe the required work in a language further down in the compiler (StackLang) where the GC primitive is implemented in concrete code.

4.1 Representing Values in the Abstract Heap

The language which comes immediately prior to the introduction of the garbage collector, DataLang, stores values of type v in its variables.

```
v = Number int | Word64 (64 word) | Block num (v list)
  | CodePtr num | RefPtr num
```

DataLang gets compiled into a language called WordLang where memory is finite and variables are of type word_loc. A word_loc is either a machine word Word w where the cardinality of α encodes the word width,[2] or a code location Loc l_1 l_2, where l_1 is the function name and l_2 is the label within that function.

```
α word_loc = Word (α word) | Loc num num
```

In what follows, we will provide some of the definitions that specify how values of type v are represented in WordLang's word_loc variables and memory. The definitions are multi-layered and somewhat verbose. In order to make sense of the definitions, we will use the following DataLang value as a running example.

```
Block 3 [Number 5; Number 80000000000000]
```

[2] For details on this representation of words and word widths, we refer to Harrison [7].

The relationship between values of type v and WordLang is split into layers. We first relate v to an instantiation of the data abstraction used by the algorithm-level verification of the garbage collector, and then separately in the next section relate that layer down to word_loc and concrete memory.

The relation v_inv, shown in Fig. 3, specifies how values of type v relate to the heap_addresses and heaps that the garbage collection algorithms operate on. The definition has a case for each value constructor in the type v. Note that list_rel r l_1 $l_2$is true iff l_1 and l_2 have equal length and their elements are pairwise related by r.

The Number case of v_inv is made complicated by the fact that DataLang allows integers of arbitrary size. If an integer is small enough to fit into a tagged machine word, then the head address x must be Data that carries the value of the small integer, and there is no requirement on the heap. If an integer i is too large to fit into a machine word, then the heap address must be a Pointer to a heap location containing the data for the bignum representing integer i.

The Word64 case of v_inv is simpler, because 64-bit words always need to be boxed. On 64-bit architectures, they are represented as a DataElement with a single word as payload. On 32-bit architectures (the only other alternative), they are represented as two words: one for each half of the 64-bit word. Here and throughout, dimindex $(:\alpha)$ is the width of the word type αword (64 for 64-bit words), and dimword $(:\alpha)$ is the size of the word type (2^{64} for 64-bit words). $(:\alpha)$ encodes the type α as a term.

The CodePtr case shows that DataLang's code pointers are represented directly as Loc-values wrapped in the Data constructor to signal that they are not pointers that the GC is to follow. The second element of the Loc is set to zero because DataLang's code pointers only point at the entry to functions.

The RefPtr case of v_inv makes use of the argument called f, which is a finite map that specifies how semantic location values for reference pointers are to be represented as addresses.

The Block case specifies how constructors and tuples from DataLang are represented. Values without a payload (e.g. those coming from source values such as `[]`, `NONE`, `()`) are represented in a word wrapped as Data. All other Block values are represented as DataElements that carry the name n of the constructor that it represents. Constructor names are numbers at this stage of compilation.

Note that pointers representing Block-values carry information about the constructor name and the length of the payload in the Pointer itself. This information is stored there in order to make primitives used for pattern matching faster: in many cases a pattern match can look at only the pointer bits rather than load the address in order to determine whether there is a match. The amount of information stored in Pointers is determined by the configuration *conf*. The ptr_bits function (definition omitted here) determines the encoding of the information based on *conf*.

For our running example, we can expand v_inv as follows to arrive at a constraint on the heap: the address x must be a pointer to a DataElement which contains Data representing integer 5, and a pointer to some memory location which contains the machine words representing

```
v_inv conf (Number i) (x,f,heap) ⟺
  if small_int (: α) i then x = Data (Word (Smallnum i))
  else
    ∃ ptr.
      x = Pointer ptr (Word 0w) ∧
      heap_lookup ptr heap = Some (Bignum i)

v_inv conf (Word64 w) (x,f,heap) ⟺
  ∃ ptr.
    x = Pointer ptr (Word 0w) ∧
    heap_lookup ptr heap = Some (Word64Rep (: α) w)

v_inv conf (CodePtr n) (x,f,heap) ⟺
  x = Data (Loc n 0)

v_inv conf (RefPtr n) (x,f,heap) ⟺
  x = Pointer (apply f n) (Word 0w) ∧ n ∈ domain f

v_inv conf (Block n vs) (x,f,heap) ⟺
  if vs = [] then
    x = Data (Word (BlockNil n)) ∧ n < dimword (: α) div 16
  else
    ∃ ptr xs.
      list_rel (λ v x'. v_inv conf v (x',f,heap)) vs xs ∧
      x = Pointer ptr (Word (ptr_bits conf n (length xs))) ∧
      heap_lookup ptr heap = Some (BlockRep n xs)

Bignum i =
  let (sign,payload) = sign_and_words_of_integer i
  in
    DataElement [] (length payload) (NumTag sign,map Word payload)

BlockNil n = n2w n << 4 + 2w

BlockRep tag xs = DataElement xs (length xs) (BlockTag tag,[])

Word64Rep (: α) w =
  if dimindex (: α) < 64 then
    DataElement [] 2 (Word64Tag,[Word ((63 >< 32) w); Word ((31 >< 0) w)])
  else DataElement [] 1 (Word64Tag,[Word ((63 >< 0) w)])
```

Fig. 3 Relation between values of type v and abstract heaps

bignum 80000000000000. Here we assume that we are talking about a 32-bit architecture. Below one can see that the first Pointer is given information, ptr_bits *conf* 3 2, about the length, 2, and tag, 3, of the Block that it points to.

$$
\begin{array}{l}
\vdash \mathsf{v_inv}\ conf\ (\mathsf{Block}\ 3\ [\mathsf{Number}\ 5;\ \mathsf{Number}\ 80000000000000])\ (x,f,heap) \iff \\
\quad \exists ptr_1\ ptr_2. \\
\qquad x = \mathsf{Pointer}\ ptr_1\ (\mathsf{Word}\ (\mathsf{ptr_bits}\ conf\ 3\ 2)) \wedge \\
\qquad \mathsf{heap_lookup}\ ptr_1\ heap = \\
\qquad\quad \mathsf{Some} \\
\qquad\qquad (\mathsf{DataElement}\ [\mathsf{Data}\ (\mathsf{Word}\ (\mathsf{Smallnum}\ 5));\ \mathsf{Pointer}\ ptr_2\ (\mathsf{Word}\ 0w)]\ 2 \\
\qquad\qquad\quad (\mathsf{BlockTag}\ 3,[])) \wedge \\
\qquad \mathsf{heap_lookup}\ ptr_2\ heap = \mathsf{Some}\ (\mathsf{Bignum}\ 80000000000000)
\end{array}
$$

The following is an instantiation of *heap* that satisfies the constraint set out by v_inv for representing our running example.

$$
\begin{array}{l}
\vdash \mathsf{v_inv}\ conf\ (\mathsf{Block}\ 3\ [\mathsf{Number}\ 5;\ \mathsf{Number}\ 80000000000000]) \\
\quad (\mathsf{Pointer}\ 0\ (\mathsf{Word}\ (\mathsf{ptr_bits}\ conf\ 3\ 2)),f, \\
\quad\ [\mathsf{DataElement}\ [\mathsf{Data}\ (\mathsf{Word}\ (\mathsf{Smallnum}\ 5));\ \mathsf{Pointer}\ 3\ (\mathsf{Word}\ 0w)]\ 2 \\
\quad\quad (\mathsf{BlockTag}\ 3,[]);\ \mathsf{Bignum}\ 80000000000000])
\end{array}
$$

As we know, the garbage collector moves heap elements and changes addresses. However, it will only transform heaps in a way that respects gc_related. We prove that v_inv properties can be transported from one heap to another if they are gc_related. In other words, execution of a garbage collector does not interfere with this data representation. Here $\mathsf{addr_apply}\ f\ (\mathsf{Pointer}\ x\ d) = \mathsf{Pointer}\ (f\ x)\ d$.

$$
\begin{array}{l}
\vdash \mathsf{gc_related}\ g\ heap_1\ heap_2 \wedge (\forall ptr\ u.\ x = \mathsf{Pointer}\ ptr\ u \Rightarrow ptr \in \mathsf{domain}\ g) \wedge \\
\quad \mathsf{v_inv}\ conf\ w\ (x,f,heap_1) \Rightarrow \\
\qquad \mathsf{v_inv}\ conf\ w\ (\mathsf{addr_apply}\ (\mathsf{apply}\ g)\ x,g \circ f,heap_2)
\end{array}
$$

In the formalisation, v_inv is used as part of abs_ml_inv (Fig. 4) which relates a list of values of type v and a reference mapping *refs* from DataLang to a state representation at the level of the collector algorithm's verification proof. The state is a list of *roots*, a *heap*, and some other components. For the relation to be true, the roots and the heap have to be well-formed (roots_ok and heap_ok as mentioned previously); furthermore, the heap layout mandated by the generational collector must be true if a generational collector is used (gc_kind_inv), and unused_space_inv specifies that $sp + sp_1$ slots of unused space exists at heap location a. The v_inv relation is used inside of bc_stack_ref_inv which specifies the relationship between *stack* and *roots*: these lists have to be pairwise (list_rel) related by v_inv.

The invariant on the layout of the heap is specified in gc_kind_inv in Fig. 5. The length of the available space is always given by $sp + sp_1$, where sp is the space available before the GC trigger pointer, see Sect. 3.2. If a non-generational GC is used, then sp_1 must be zero indicating that the trigger is always at the end of the available space. For the generational collector, it must be possible to split the heap (heap_split) at the end of the available space so that all heap elements prior to this cut off are not references, and all heap elements after this pointer are references. Furthermore, each generation boundary *gens* must be well-formed (gen_state_ok). Here gen_state_ok states that there must not be any pointers from old data to new data, or more precisely, that every pointer must point to a location that is before the generation boundary or point into the references at the end of the heap.

At the time of writing, the algorithm-level formalisation is proved correct for a version which supports having several nested generations, but the word-level implementation of the algorithm has only been set up to work for at most one nursery generation, i.e. the setting where $\mathsf{length}\ gens \leq 1$.

```
abs_ml_inv conf stack refs (roots,heap,be,a,sp,sp1,gens) limit ⟺
  roots_ok roots heap ∧ heap_ok heap limit ∧
  gc_kind_inv conf a sp sp1 gens heap ∧
  unused_space_inv a (sp + sp1) heap ∧
  bc_stack_ref_inv conf stack refs (roots,heap,be)

unused_space_inv ptr l heap ⟺
  (l ≠ 0 ⇒ heap_lookup ptr heap = Some (Unused (l − 1))) ∧
  ptr + l ≤ heap_length heap ∧ data_up_to ptr heap

bc_stack_ref_inv conf stack refs (roots,heap,be) ⟺
  ∃f.
    injective (apply f) (domain f) { a | isSomeDataElement (heap_lookup a heap) } ∧
    domain f ⊆ domain refs ∧ list_rel (λv x. v_inv conf v (x,f,heap)) stack roots ∧
    ∀n. reachable_refs stack refs n ⇒ bc_ref_inv conf n refs (f,heap,be)
```

Fig. 4 Invariants in the compiler proof

```
gc_kind_inv c a sp sp1 gens heap ⟺
  case c.gc_kind of
    None ⇒ sp1 = 0
  | Simple ⇒ sp1 = 0
  | Generational sizes ⇒
    gen_state_ok a (a + sp + sp1) heap gens ∧
    ∃h1 h2.
      heap_split (a + sp + sp1) heap = Some (h1,h2) ∧ every (λx. ¬isRef x) h1 ∧
      every isRef h2

every P [] ⟺ T
every P (h::t) ⟺ P h ∧ every P t

gen_state_ok a refs_start heap (GenState v0 starts) ⟺
  every (λs. s ≤ a) starts ∧ every (gen_start_ok heap refs_start) starts

gen_start_ok heap refs_start gen_start ⟺
  ∃h1 h2.
    heap_split gen_start heap = Some (h1,h2) ∧
    ∀xs l d p e.
      mem (DataElement xs l d) h1 ∧ mem (Pointer p e) xs ⇒
        p < gen_start ∨ refs_start ≤ p
```

Fig. 5 Invariants regarding the heap layout of the generational GC

4.2 Data Refinement Down to Concrete Memory

The relation provided by v_inv only gets us halfway down to WordLang's memory representation. In WordLang, values are of type word_loc, and memory is modelled as a function, α word $\rightarrow$ α word_loc, and an address domain set.

We use separation-logic formulas to specify how abstract heaps, i.e. lists of heap_elements, are represented in memory. We define separating conjunction *, and use fun2set to turn a memory function *m* and its domain set *dm* into something we can write separation logic assertions about. The relevant definitions are:

$$\begin{aligned}
&\mathsf{split}\ s\ (u,v) \iff u \cup v = s \wedge u \cap v = \emptyset \\
&p * q = (\lambda s.\ \exists u\ v.\ \mathsf{split}\ s\ (u,v) \wedge p\ u \wedge q\ v) \\
&a \mapsto x = (\lambda s.\ s = \{\ (a,x)\ \}) \\
&\mathsf{emp} = (\lambda s.\ s = \emptyset) \\
&\mathsf{cond}\ c = (\lambda s.\ s = \emptyset \wedge c) \\
&\mathsf{fun2set}\ (m,dm) = \{\ (a,m\ a) \mid a \in dm\ \}
\end{aligned}$$

Using this separation logic set up and a number of auxiliary functions, we define word_heap *a heap conf* to assert that a heap_element list *heap* is in memory, starting at address *a*. The definition of word_heap, which is partially shown in Fig. 6, uses word_el to assert that individual heap_elements are correctly represented. Here and throughout, n2w is a function which turns a natural number into the corresponding word modulo the word size. Numerals written with a *w*-suffix indicate that it is a word literal, i.e. $2w$ is the same as n2w 2. Here $w \ll n$ shifts word w by n bits left, and $(m - n)\ w$ zeros bit k of word w if $k > m$ or $k < n$.

Figure 7 shows an expansion of the word_heap assertion applied to our running example from the previous section.

4.3 Implementing the Garbage Collector

WordLang *Implementation* The garbage collector is used in the WordLang semantics as a function that the semantics of Alloc applies to memory when the allocation primitive runs out of memory. At this level, the garbage collector is essentially a function from a list of roots and a concrete memory to a new list of roots and concrete memory.

To implement the new garbage collector, we define a HOL function at the level of a concrete memory, and prove that it correctly mimics the operations performed by the algorithm-level implementation from Sect. 3.

In Fig. 8 we show the definition of word_gen_gc_partial_move, which is the refinement of gen_gc_partial_move. A side-by-side comparison with the latter (shown in Fig. 1) reveals that it's essentially the same function, recast in more concrete terms: for example, pattern matching on the constructor of the heap element is concretised by inspection of tag bits, and we must be explicit about converting between pointers (relative to the base address of the current heap) and (absolute) memory addresses. Note that it is still a specification. It is never executed: its only use is to define the semantics of WordLang's garbage collection primitive.

To formally relate word_gen_gc_partial_move and gen_gc_partial_move we prove the following theorem, which states that the concrete memory is kept faithful to the algorithm's operations over the heaps. We prove similar theorems about the other components of the garbage collectors.

$$\begin{aligned}
\vdash\ & \mathsf{gen_gc_partial_move}\ gc_conf\ s\ x = (x_1,s_1) \wedge \\
& \mathsf{word_gen_gc_partial_move}\ conf\ (\mathsf{word_addr}\ conf\ x,\ldots) = (w,\ldots) \wedge \ldots \wedge \\
& (\mathsf{word_heap}\ a\ s.\mathsf{heap}\ conf\ *\ \mathsf{word_heap}\ p\ s.\mathsf{h2}\ conf\ *\ \ldots)\ (\mathsf{fun2set}\ (m,dm)) \Rightarrow \\
& \quad w = \mathsf{word_addr}\ conf\ x_1 \wedge \ldots \wedge \\
& \quad (\mathsf{word_heap}\ a\ s_1.\mathsf{heap}\ conf\ *\ \mathsf{word_heap}\ p_1\ s_1.\mathsf{h2}\ conf\ *\ \ldots)\ (\mathsf{fun2set}\ (m_1,dm))
\end{aligned}$$

As a corollary of these theorems, we can lift the result from Sect. 3.4 that the generational garbage collector does not lose any live elements, to the same property about WordLang's garbage collection primitive and the allocation function.

For the allocation primitive, we prove a key lemma shown in Fig. 9: if the state relation state_rel between DataLang state *s* and WordLang state *t* holds, then running the allocation

```
word_heap a [] conf = emp
word_heap a (x::xs) conf =
  word_el a x conf * word_heap (a + bytes_in_word * n2w (el_length x)) xs conf

bytes_in_word = n2w (dimindex (: α) div 8)

word_list a [] = emp
word_list a (x::xs) = a ↦ x * word_list (a + bytes_in_word) xs

word_list_exists a n = ∃ xs. word_list a xs * cond (length xs = n)

word_el a (Unused l) conf = word_list_exists a (l + 1)
word_el a (ForwardPointer n d l) conf =
  a ↦ (Word (n2w n << 2)) * word_list_exists (a + bytes_in_word) l
word_el a (DataElement ys l (tag,qs)) conf =
  let (h,ts,c) = word_payload ys l tag qs conf
  in
    word_list a (Word h::ts) *
    cond
      (length ts < 2 ** (dimindex (: α) − 4) ∧
       decode_length conf h = n2w (length ts) ∧ c)

word_payload ys l (BlockTag n) qs conf =
  (make_header conf (n2w n << 2) (length ys),
   map (word_addr conf) ys,
   qs = [] ∧ length ys = l ∧
   encode_header conf (n * 4) (length ys) =
     Some (make_header conf (n2w n << 2) (length ys)))
...

make_header conf tag len =
  (n2w len << (dimindex (: α) − conf.len_size) || tag << 2 || 3w)

word_addr conf (Data (Loc l1 l2)) = Loc l1 l2
word_addr conf (Data (Word v)) = Word (v && ¬1w)
word_addr conf (Pointer n w) = Word (get_addr conf n w)

get_addr conf n w = (n2w n << shift_length conf || get_lowerbits conf w)

get_lowerbits conf (Word w) = ((small_shift_length conf − 1 − 0) w || 1w)
get_lowerbits conf (Loc v3 v4) = 1w
```

Fig. 6 Some of the definitions for word_heap

routine on input k will either result in an abort with NotEnoughSpace (and an unchanged foreign-function interface state ffi) or in success (indicated by *res* = None) and space for k more slots in a new WordLang state *new_t* which is state_rel-related to a modified version of the original DataLang state. Here *names* is the set of local variable names that need to be stored on the stack in case the garbage collector is called, i.e. it is the set of local variables that survive a call to the allocation routine.

With the help of such properties about the WordLang's allocation routine, we can prove a compiler correctness theorem for the compiler phase from DataLang to WordLang. Compiler correctness theorems are, in the context of the CakeML compiler [18], simulation relations as shown in Fig. 10.

```
 word_heap a
  [DataElement [Data (Word (Smallnum 5)); Pointer 3 (Word 0w)] 2
    (BlockTag 3,[]); Bignum 80000000000000] conf (fun2set (m,dm))
⟺
 (word_el a
   (DataElement [Data (Word (Smallnum 5)); Pointer 3 (Word 0w)] 2
     (BlockTag 3,[])) conf *
  word_el (a + 12w) (Bignum 80000000000000) conf) (fun2set (m,dm))
⟺
 (a ↦ (Word (make_header conf 12w 2)) *
  (a + 4w) ↦ (word_addr conf (Data (Word (Smallnum 5)))) *
  (a + 8w) ↦ (word_addr conf (Pointer 3 (Word 0w))) *
  (a + 12w) ↦ (Word (make_header conf 3w 2)) *
  (a + 16w) ↦ (Word 1939144704w) * (a + 20w) ↦ (Word 18626w))
  (fun2set (m,dm))
⟺
 m a = Word (make_header conf 12w 2) ∧ m (a + 4w) = Word 20w ∧
 m (a + 8w) = Word (get_addr conf 3 (Word 0w)) ∧
 m (a + 12w) = Word (make_header conf 3w 2) ∧
 m (a + 16w) = Word 1939144704w ∧ m (a + 20w) = Word 18626w ∧
 dm = { a; a + 4w; a + 8w; a + 12w; a + 16w; a + 20w } ∧
 all_distinct [a; a + 4w; a + 8w; a + 12w; a + 16w; a + 20w]
```

Fig. 7 Running example expanded to concrete memory assertion

```
word_gen_gc_partial_move conf (Loc l1 l2,i,pa,old,m,dm,gs,rs) =
 (Loc l1 l2,i,pa,m,T)
word_gen_gc_partial_move conf (Word w,i,pa,old,m,dm,gs,rs) =
 if (w && 1w) = 0w then (Word w,i,pa,m,T)
 else
  let header_addr = ptr_to_addr conf old w; tmp = header_addr − old
  in
   if tmp <+ gs ∨ rs ≤+ tmp then (Word w,i,pa,m,T)
   else
    let c1 = ptr_to_addr conf old w ∈ dm; v = m (ptr_to_addr conf old w)
    in
     if is_fwd_ptr v then
      (Word (update_addr conf (theWord v >>> 2) w),i,pa,m,c1)
     else
      let c2 = c1 ∧ header_addr ∈ dm ∧ isWord (m header_addr);
      len = decode_length conf (theWord (m header_addr));
      v = i + len + 1w;
      (pa1,m1,c3) = memcpy (len + 1w) header_addr pa m dm;
      c4 = c2 ∧ header_addr ∈ dm ∧ c3;
      m1 = (header_addr =+ Word (i << 2)) m1
      in
       (Word (update_addr conf i w),v,pa1,m1,c4)
```

Fig. 8 The WORDLANG shallow embedding of the move primitive for partial collection

Note that the compiler correctness theorem implies that the garbage collector must terminate on every cycle. This is because the theorem does not allow a terminating execution (e.g. one that terminates returning a value Rval *v*) to be simulated by a divergent execution. Here a divergent execution is one that results in Rtimeout_error for every value *ck* of the semantic clock. In the case of diverging source-language programs, the GC cannot diverge, since diverging executions must be simulated by diverging executions that exhibit the same observable events (i.e. the same FFI calls); hence if the GC diverged the STACKLANG program

```
⊢ state_rel c l1 l2 s t [] locs ∧ cut_env names s.locals = Some new_locals ∧
  alloc k names t = (res,new_t) ⇒
    res = Some NotEnoughSpace ∧ new_t.ffi = t.ffi ∨
    res = None ∧ state_rel c l1 l2 (s with ⟪locals := new_locals; space := k⟫) new_t [] locs
```

Fig. 9 Key lemma: the allocation primitive of WordLang either raises a NotEnoughSpace-exception or returns normally with a *new_t* state that relates to the original DataLang state *s* updated to have *k* slots of space available

```
⊢ evaluate_data (Call None (Some start) [] None,s) = (res,s1) ∧
  res ≠ Some (Rerr (Rabort Rtype_error)) ∧ state_rel_ext x l1 l2 s t ⇒
  ∃ ck t1 res1.
    evaluate_word (Call None (Some start) [0] None,inc_clock ck t) = (res1,t1) ∧
    (res1 = Some NotEnoughSpace ⇒ t1.ffi.io_events ≼ s1.ffi.io_events) ∧
    (res1 ≠ Some NotEnoughSpace ⇒
      t1.ffi = s1.ffi ∧
      case res of
        None ⇒ res1 = None
      | Some (Rval v) ⇒ ∃ w. res1 = Some (Result (Loc l1 l2) w)
      | Some (Rerr (Rraise v')) ⇒ ∃ v w. res1 = Some (Exception v w)
      | Some (Rerr (Rabort Rtype_error)) ⇒ res1 = Some TimeOut
      | Some (Rerr (Rabort Rtimeout_error)) ⇒ res1 = Some TimeOut
      | Some (Rerr (Rabort (Rffi_error f))) ⇒ res1 = Some (FinalFFI f))
```

Fig. 10 Correctness theorem relating DataLang evaluation with WordLang evaluation. Here state_rel_ext relates DataLang states *s* with WordLang states *t* and includes the requirements: (1) the code in state *t* is the compilation of the code in state *s*; (2) the GC is present in state *t*; and (3) all data from *s* is correctly represented in state *t*

would emit fewer FFI calls.[3] For more details on this technique for treating divergence we refer to Owens et al. [15].

StackLang *Implementation:* As mentioned earlier, the WordLang garbage collection primitive needs to be implemented by a deep embedding that can be compiled with the rest of the code. This happens in the next intermediate language, StackLang, which uses the same data representation but concretises the stack and adds primitives for inspecting and manipulating it. These primitives are used to implement root scanning in the StackLang implementation of the GC.

The GC implementation is tedious: the StackLang programmer does not have the luxury of variables, and so must manually juggle data between registers and memory locations. To give the flavour, here is a pretty printed version of the StackLang code for a memory copying procedure. This code snippet copies *n* machine words from the memory location pointed to by register 2, to the memory location pointed to by register 3, where *n* is the contents of register 0. Here `BYTES_IN_WORD` is a target-specific constant specifying the number of bytes in a machine word: for 32-bit architectures it is 4 and 64-bit architectures it is 8. In HOL, the corresponding constant, bytes_in_word, is a constant whose value depends on its type, e.g. bytes_in_word:32 word = 4 and bytes_in_word:64 word = 8.

```
while (reg0 <> 0) {
     reg1 := mem[reg2];
     reg2 := reg2 + BYTES_IN_WORD;
     reg0 := reg0 - 1;
     mem[reg3] := reg1;
     reg3 := reg3 + BYTES_IN_WORD;
   }
```

[3] In principle, the theorem would allow for a GC that may diverge iff the source program diverges silently. This would be unproblematic since we would then simulate silent divergence with silent divergence.

The actual deep embedding as StackLang code is the following:

```
memcpy_code =
 While NotEqual 0 (Imm 0w)
  (Seq (Inst (Mem Load 1 (Addr 2 0w)))
   (Seq (Inst (Arith (Binop Add 2 2 (Imm bytes_in_word))))
    (Seq (Inst (Arith (Binop Sub 0 0 (Imm 1w))))
     (Seq (Inst (Mem Store 1 (Addr 3 0w)))
      (Inst (Arith (Binop Add 3 3 (Imm bytes_in_word)))))))))
```

Here $0w$ denotes the machine word where all bits are 0, and $1w$ the machine word where all bits are 0 save for the LSB.

We prove that according to StackLang's big-step semantics, evaluating this program computes the same function and has the same effect on memory as the corresponding shallow embedding memcpy does:

```
⊢ memcpy (n2w n) a b m dm = (b1,m1,T) ∧ n < dimword (: α) ∧ s.memory = m ∧
  s.mdomain = dm ∧ get_var 0 s = Some (Word (n2w n)) ∧ 1 ∈ domain s.regs ∧
  get_var 2 s = Some (Word a) ∧ get_var 3 s = Some (Word b) ⇒
   ∃r1.
     evaluate (memcpy_code,s with clock := s.clock + n) =
      (None,
       s with
       ⟪memory := m1;
        regs :=
         s.regs |++
         [(0,Word 0w); (1,r1); (2,Word (a + n2w n * bytes_in_word));
          (3,Word b1)]⟫)
```

Here |++ updates a finite map with the key-value pairs in the RHS. We prove similar theorems about all the constituent parts of the GC implementation, allowing us to lift the result that the WordLang garbage collection primitive does not lose live elements to the same result about its StackLang implementation.

Root scanning is made explicit and implemented in the StackLang implementation. The root scanning code has to find and process all current roots in the program stack. This is made tedious because not all slots in the stack contain active data or data that is relevant to the GC. Each stack frame is marked with an identifier which is a pointer into a separate data structure where the GC can at runtime find a compact representation of a description of the structure of the stack frame. Verification of the root scanning code is not particularly difficult, but the proofs are long due to the low-level nature of the compact stack frame descriptions, which is described in previous work [18].

The StackLang implementation of the GC is injected into the program to be compiled as part of the StackLang phases of the compiler. At this point all uses of the allocation primitive have been replaced by calls to the GC implementation. From this point onwards, the code for the GC is just another part of the program to be compiled, and there is no need for the remaining passes to even be aware of whether the code contains a garbage collector.

5 Timing the GC

In this section, we evaluate how the performance of our new generational garbage collector compares to CakeML's pre-existing copying collector on a number of benchmarks. We have several objectives here. We would like to discover whether the new generational collector is performant enough to be a useful addition to the CakeML ecosystem, and if so, to give users some hints about what kind of programs our collector might be useful for. Moreover, we are interested in the performance penalty that maintaining our heap layout incurs on non-generational copying collection.

Benchmark Suite
Our first seven benchmarks are taken from the MLton repository.[4] They exclude benchmarks that use features currently unsupported by CakeML, and features that are treated in substantially different ways by MLton and CakeML (records, system calls, machine words, floating point numbers). Moreover, we exclude benchmarks whose memory footprint is too small to trigger the copying collector. For the remaining benchmarks, we modified minor details like syntax and currying to make the benchmarks compatible with CakeML's parser and basis library.

The remaining benchmarks repeatedly create short-lived binary trees of various depths, simulating programs that require different amounts of short-lived live data in addition to a fixed amount of long-lived data allocated at the start of the program. The depths considered go from 5 (157 machine words in memory) to 16 (327,677 machine words in memory). We would expect the generational collector to perform better with smaller trees that fit comfortably in a generation, but worse on bigger trees.

Setup We compiled each benchmark program with the heap size set to 10 MB—the small size makes sure we trigger plenty of collection cycles—and with a variety of different garbage collection settings:

1. (*copying*): Copying GC
2. (*no-gen*): Generational GC, with generation size > heap size (hence partial collection never triggers).
3. (*small-gen*): Generational GC, with 100,000 word generation size
4. (*large-gen*): Generational GC, with 200,000 word generation size

All benchmarks were run on a 4 GHz Intel(R) Core(tm) i7-6700K CPU and 32 GB of memory, running Debian 4.9.82-1. Presented results are the average over 100 runs. In order to allow us to distinguish GC time from other time, we run the compiler in a debug mode which injects snippets of timing code at GC entry and exit.

Results Table 1 shows our benchmark results for the MLton benchmarks. We see that the relative performance of our collectors varies wildly: from `imp-for` where the generational GC performs worse by two orders of magnitude, to `smith-normal-form` where the generational GC performs approximately 4 times better.

The benchmark `imp-for` consists of 7 nested for-style loops that allocate around 100 million references in total, and uses very little immutable data. That a program with such an allocation pattern does not benefit from our generational garbage collection scheme is hardly surprising: when most data is mutable, running collection cycles on only the immutable data is a waste of time. The other example where generational collection underperforms is `pidigits`, which is based on an encoding of lazy lists as a function for producing the tail;

[4] https://github.com/MLton/mlton/tree/master/benchmark.

Table 1 MLton benchmark results, measured as total garbage collection time in milliseconds with different collector settings (and as percentage of the benchmark's total run-time)

	copying	*no-gen*	*small-gen*	*large-gen*
`merge`	7520 (63%)	6830 **(61%)**	11,300 (74%)	**6700** (62%)
`imp-for`	**8.7 (0.08%)**	9.7 (0.08%)	5810 (34%)	2360 (17%)
`life`	12.5 (0.08%)	13 (0.08%)	33 (0.2%)	**17 (0.1%)**
`pidigits`	**10 (0.09%)**	11 (0.1%)	75 (0.7%)	37 (0.34%)
`logic`	7570 (33%)	**5700 (27%)**	10,300 (41%)	5700 (28%)
`mpuz`	**50.4 (0.37%)**	56.3 (0.41%)	98.3 (0.74%)	51.6 (0.39%)
`smith-normal-form`	69.0 (0.47%)	69.0 (0.47%)	22.4 (0.15%)	**19.6 (0.13%)**

Best-performing collector on each benchmark indicated in boldface

Table 2 Results for tree allocation benchmarks

	copying	*no-gen*	*small-gen*	*large-gen*
`depth5`	1840 (21%)	1920 (22%)	30.8 (0.46%)	**16.9 (0.25%)**
`depth6`	1810 (21%)	1930 (22%)	39.4 (0.59%)	**20.5 (0.31%)**
`depth7`	1820 (21%)	1930 (22%)	54.5 (0.81%)	**29.0 (0.44%)**
`depth8`	1820 (21%)	1910 (22%)	86.5 (1.29%)	**45.4 (0.68%)**
`depth9`	1820 (21%)	1920 (22%)	146 (2.1%)	**78.5 (1.2%)**
`depth10`	1820 (21%)	1910 (22%)	266 (3.9%)	**145 (2.1%)**
`depth11`	1820 (21%)	1910 (22%)	504 (7.1%)	**278 (4.0%)**
`depth12`	1980 (23%)	2090 (24%)	988 (13%)	**543 (7.6%)**
`depth13`	1970 (23%)	2090 (24%)	2020 (23%)	**1130 (14%)**
`depth14`	**1970 (23%)**	2080 (24%)	4020 (38%)	2270 (25%)
`depth15`	**1970 (22%)**	2080 (24%)	7630 (53%)	4970 (43%)
`depth16`	**6420 (49%)**	6470 (49%)	12,600 (65%)	10,600 (61%)

Total garbage collection time in milliseconds with different collector settings (and as percentage of the benchmark's total run-time). Best-performing collector on each benchmark indicated in boldface

more research is needed to determine why the generational GC would be bad for such an application.

The `smith-normal-form` has several features that seem a good fit for generational collection. It uses references, albeit more sparingly than `imp-for`: it represents 35*35 matrices as `int arrays`, so collecting the references themselves would be of relatively little use. The fact that the references point at integers that change frequently means high turnover of data and a small footprint of live data at all times. Hence partial cycles are likely to be quick (not much more to see after following each reference to an integer) and free up plenty of space (1225 integers is very little live data).

The difference between the columns *copying* and *no-gen* shows the overhead incurred by maintaining our heap layout, which turns out to be negligible. The difference between *small-gen* and *large-gen* suggest that all else being equal, larger generation sizes are preferable.

Table 2 shows our results for the tree allocation benchmarks. We see that the performance profile of our generational collector vs the copying collector is as expected: when the amount of live non-persistent data at any one time is much smaller than the generation size, the generational collector outperforms the copying collector by two orders of magnitude. The collection time for the generational collector grows linearly in the size of the non-persistent live memory (i.e. exponentially in the tree depth), until it starts performing worse than the copying collector when the trees no longer fit comfortably in a generation. Meanwhile, the performance profile of copying collection is more flat. At depth 16, both collectors start

exhibiting degenerate performance as the size of the trees approaches the heap size. Trees of depth 17 are too big to fit in the heap regardless of which collection scheme is used.

In conclusion, we find that the generational collector is indeed a useful addition to the CakeML ecosystem that can improve the performance of programs where most new data has small live ranges. Of course, neither the generational nor the copying collector is a clear winner for all use cases, and the optimal choice of collector will depend heavily on the allocation pattern of the user program under consideration. We encourage performance-conscious compiler users to perform their own experiments to determine which settings fit their programs. However, it is worth stressing that when program responsiveness matters, the generational GC is often preferable when the performance difference is small, because the same total GC time is spread out over many shorter collection cycles.

6 Discussion of Related Work

Anand et al. [1] reports that the CertiCoq project has a "high-performance generational garbage collector" and a project is underway to verify this using Verifiable C in Coq. Their setting is simpler than ours in that their programs are purely functional, i.e. they can avoid dealing with the added complexity of mutable state. The text also suggests that their garbage collector is specific to a fixed data representation. In contrast, the CakeML compiler allows a highly configurable data representation, which is likely to become more configurable in the future. The CakeML compiler generates a new garbage collector implementation for each configuration of the data representation.

CakeML's original non-generational copying collector has its origin in the verified collector described in Myreen [12]. The same verified algorithm was used for a verified Lisp implementation [13] which in turn was used underneath the proved-to-be-sound Milawa prover [2]. These Lisp and ML implementations are amongst the very few systems that use verified garbage collectors as mere components of much larger verified implementations. Verve OS [19] and Ironclad Apps [9] are verified stacks that use verified garbage collectors internally.

Numerous abstract garbage collector algorithms have been mechanically verified before. However, most of these only verify the correctness at the algorithm-level implementation and only consider mark-and-sweep algorithms. Noteworthy exceptions include Hawblitzel and Petrank [10], McCreight [11], and Gammie et al. [5].

Hawblitzel and Petrank [10] show that performant verified x86 code for simple mark-and-sweep and Cheney copying collectors can be developed using the Boogie verification condition generator and the Z3 automated theorem prover. Their method requires the user to write extensive annotations in the code to be verified. These annotations are automatically checked by the tools. Their collector implementations are realistic enough to show good results on off-the-shelf C# benchmarks. This required them to support complicated features such as interior pointers, which CakeML's collector does not support. We decided to not support interior pointers in CakeML because they are not strictly needed and they would make the inner loop of the collector a bit more complicated, which would probably cause the inner loop to run a little slower.

McCreight [11] verifies copying and incremental collectors implemented in MIPS-like assembly. The development is done in Coq, and casts his verification efforts in a common framework based on ADTs that all the collectors refine.

Gammie et al. [5] verify a detailed model of a state-of-the-art concurrent mark-and-sweep collector in Isabelle/HOL, with respect to an x86-TSO memory model. A related effort by Zakowski et al. [20] uses Coq to verify a concurrent mark-and-sweep collector expressed in a purpose-built compiler intermediate representation rather than the pseudocode of Gammie et al., although Zakowski et al. verifies theirs with respect to an interleaving semantics.

Pavlovic et al. [16] focus on an earlier step, namely the synthesis of concurrent collection algorithms from abstract specifications. The algorithms thus obtained are at a similar level of abstraction to the algorithm-level implementation that we start from. The specifications are cast in lattice-theoretic terms, so e.g. computing the set of live nodes is fixpoint iteration over a function that follows pointers from an element. A main contribution is an adaptation of the classic fixpoint theorems to a setting where the monotone function under consideration may change, which can be thought of as representing interference by mutators.

This paper started by listing incremental, generational, and concurrent as variations on the basic garbage collection algorithms. There have been prior verifications of incremental algorithms (e.g. [8,11,14,17]) and concurrent ones (e.g. [3,5,6,16]), but we believe that this paper is the first to report on a successful verification of a generational garbage collector.

7 Summary

This paper verifies a generational copying garbage collector and integrates it into the verified CakeML compiler. The algorithm-level part of our proof is structured to follow the usual informal argument for a generational collector's correctness: a partial collection is the same as running a full collection on part of the heap if pointers to old data are treated as non-pointers. To the best of our knowledge, this paper is the first to report on a completed formal verification of a generational garbage collector.

Acknowledgements We thank Ramana Kumar and the anonymous reviewers for many helpful comments on drafts of this text. This work was partly supported by the Swedish Research Council and the Swedish Foundation for Strategic Research.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Anand, A., Appel, A., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Belanger, O.S., Sozeau, M., Weaver, M.: CertiCoq: A verified compiler for Coq. In: Coq for Programming Languages (CoqPL) (2017)
2. Davis, Jared, Myreen, Magnus O.: The reflective Milawa theorem prover is sound (down to the machine code that runs it). J. Autom. Reason. **55**(2), 117–183 (2015)
3. Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S., Steffens, E.F.M.: On-the-fly garbage collection: an exercise in cooperation. Commun. ACM **21**(11), 966–975 (1978)
4. Ericsson, A.S., Myreen, M.O., Åman Pohjola, J.: A verified generational garbage collector for CakeML. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) Interactive Theorem Proving (ITP), vol. 10499 of LNCS, pp. 444–461. Springer, Berlin (2017)
5. Gammie, P., Hosking, A.L., Engelhardt, K.: Relaxing safely: verified on-the-fly garbage collection for x86-TSO. In: Grove, D., Blackburn, S. (eds.) Programming Language Design and Implementation (PLDI). ACM , pp. 99–109 (2015)

6. Gonthier, G.: Verifying the safety of a practical concurrent garbage collector. In: Alur, R., Henzinger, T.A. (eds) Computer Aided Verification (CAV), vol. 1102 of Lecture Notes in Computer Science. Springer, Berlin (1996)
7. Harrison, J.: A HOL theory of euclidean space. In: Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, August 22–25, 2005, Proceedings, pp. 114–129 (2005)
8. Havelund, K.: Mechanical verification of a garbage collector. In: Parallel and Distributed Processing, 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, pp. 1258–1283 (1999)
9. Hawblitzel, C., Howell, J., Lorch, J.R., Narayan, A., Parno, B., Zhang, D., Zill, B.: Ironclad apps: End-to-end security via automated full-system verification. In: Operating Systems Design and Implementation (OSDI), pp. 165–181, Broomfield, CO (2014). USENIX Association
10. Hawblitzel, C., Petrank, E.: Automated verification of practical garbage collectors. Log. Methods Comput. Sci. **6**(3), 441–453 (2010)
11. McCreight, A.: The mechanized verification of garbage collector implementations. Ph.D. thesis, Yale University, December (2008)
12. Myreen, M.O.: Reusable verification of a copying collector. In: Leavens, G.T., O'Hearn, P.W., Rajamani, S.K. (eds) Verified Software: Theories, Tools, Experiments (VSTTE), vol. 6217 of Lecture Notes in Computer Science. Springer, Berlin (2010)
13. Myreen, M.O., Davis, J.: A verified runtime for a verified theorem prover. In: van Eekelen, M.C.J.D., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds) Interactive Theorem Proving (ITP) , pp. 265–280 (2011)
14. Nieto, L.P., Esparza, J.: Verifying single and multi-mutator garbage collectors with owicki-gries in isabelle/hol. In: International Symposium on Mathematical Foundations of Computer Science, pp. 619–628. Springer, Berlin (2000)
15. Owens, S., Myreen, M.O., Kumar, R., Tan, Y.K.: Functional big-step semantics. In: Programming Languages and Systems—25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings, pp. 589–615 (2016)
16. Pavlovic, D., Pepper, P., Smith, D.R.: Formal derivation of concurrent garbage collectors. In: Bolduc, C., Desharnais, J., Ktari, B. (eds) Mathematics of Program Construction. MPC 2010. Lecture Notes in Computer Science, vol. 6120, pp. 353–376. Springer, Berlin, Heidelberg (2010)
17. Russinoff, David M.: A mechanically verified incremental garbage collector. Formal Aspects Comput. **6**(4), 359–390 (1994)
18. Tan, Y.K., Myreen, M.O., Kumar, R., Fox, A., Owens, S., Norrish, M.: A new verified compiler backend for CakeML. In: Garrigue, J., Keller, G., Sumii, E. (eds) International Conference on Functional Programming (ICFP). ACM (2016)
19. Yang, J., Hawblitzel, C.: Safe to the last instruction: automated verification of a type-safe operating system. In: Zorn, B.G., Aiken, A. (eds) Programming Language Design and Implementation (PLDI), pp. 99–110. New York, NY, ACM (2010)
20. Zakowski, Y., Cachera, D., Demange, D., Petri, G., Pichardie, D., Jagannathan, S., Vitek, J.: Verifying a concurrent garbage collector using a rely-guarantee methodology. In: Interactive Theorem Proving—8th International Conference, ITP 2017, Brasília, Brazil, September 26–29, 2017, Proceedings, pp. 496–513 (2017)