# A verified generational garbage collector for CakeML

Downloaded from: https://research.chalmers.se, 2026-10-14 16:10 UTC

Citation for the original published paper (version of record):
Sandberg Eriksson, A., Myreen, M., Åman Pohjala, J. (2017). A verified generational garbage collector for CakeML. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 10499 LNCS: 444-461.
http://dx.doi.org/10.1007/978-3-319-66107-0_28

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)

# A Verified Generational Garbage Collector for CakeML

Adam Sandberg Ericsson, Magnus O. Myreen(✉), and Johannes Åman Pohjola

Chalmers University of Technology, Gothenburg, Sweden
myreen@chalmers.se

**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.

## 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, their performance is essential to make the memory abstraction seem worthwhile. As a result, there have been numerous improvements 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 [15]. The CakeML project has produced a formally verified compiler for an ML-like language called CakeML. The

© Springer International Publishing AG 2017
M. Ayala-Rincón and C.A. Muñoz (Eds.): ITP 2017, LNCS 10499, pp. 444–461, 2017.
DOI: 10.1007/978-3-319-66107-0_28

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.

## 2 Approach

In this section, we give a high-level overview of the work and our approach to it. Subsequent sections will cover some—but for lack of space, not all—of 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. 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, 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 one only has 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 shallow embedding and its verification proof is under `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

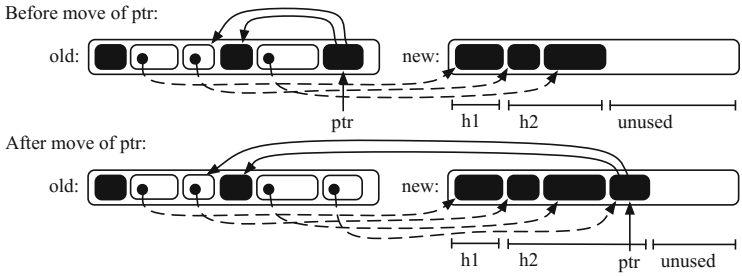
Intuitively, a Cheney copying garbage collector copies the live elements from the current heap into a new heap. We will call the heaps old and new. In its simplest form, the algorithm keeps track of two boundaries inside the new heap. These split the new heap into three parts, which we will call h1,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

Throughout execution, the heap segment h1 will only contain pointers to the new heap, and heap segment h2 will only contain pointers 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 forward pointer or not. A forward pointer is a heap element with a special tag to distinguish it from other heap elements. Forward pointers will only ever occur in the heap if the garbage collector puts them there; between collection cycles, they are never present nor created.

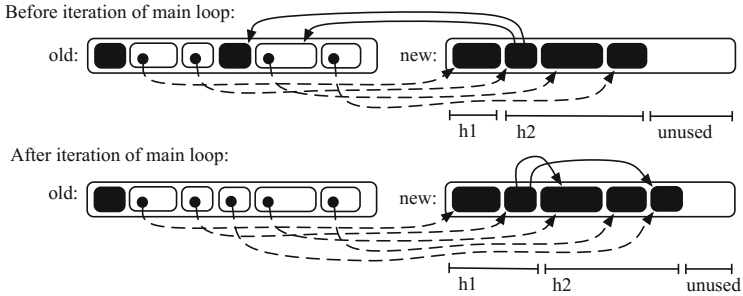
If d is not a forward pointer, then d will be copied to the end of heap segment h2, consuming some of the unused space, and ptr is updated to be the address of the new location of d. A forward pointer to the new location is inserted at the old location of d, namely at the original value of ptr. We draw forward 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 forward pointer, the move primitive knows that this element has been moved previously; it reads the new pointer value from the forward 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.

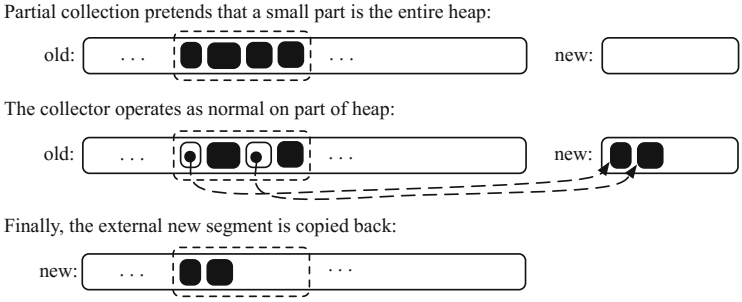



This process is repeated until h2 becomes empty, and the new heap contains no pointers to the old heap. The old heap can then be discarded, since it only contains data that is unreachable from the roots. The next time the garbage collector runs, the previous old heap is used as the new heap.

### 3.2 Intuition for Generational Algorithm

Generational garbage collectors attempt to 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.

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.

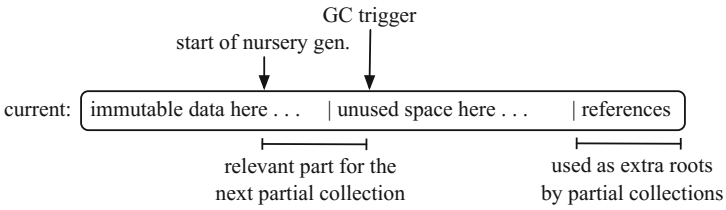



For the partial collection to work we need:

(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 is as follows. This diagram also shows that we use a GC trigger pointer, which causes a GC invocation whenever one attempts to allocate past the GC trigger pointer.



We modify the simple garbage collection algorithm described above to maintain this layout, and we make each run of the partial collection algorithm treat the references as roots that are not part of the heap. This way we can meet the two requirements (a) and (b) from above.

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.

### 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 somwewhat abstract with type variables. We use this flexiblity 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, a forward pointer, or actual data.

$$\alpha \text{ heap_address} = \text{Pointer num } \alpha \mid \text{Data } \alpha$$

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

Each heap element carries its concrete length, i.e. how many machine words the eventual memory representation will hold. The length function, el_length, returns $l$ plus one because we do not allow heap elements of length zero.

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

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; $s.ok$ 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.

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 forward pointer, it follows the forward 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 forward 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.

```
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
  if ptr < conf.gen_start ∨ conf.refs_start ≤ ptr then
   (Pointer ptr d,state with ok := ok)
  else
   case heap_lookup ptr state.heap of
    None ⇒ (Pointer ptr d,state with ok := F)
   | Some (Unused v9) ⇒ (Pointer ptr d,state with ok := F)
   | Some (ForwardPointer ptr' v11 l') ⇒ (Pointer ptr' d,state)
   | Some (DataElement xs l dd) ⇒
     let ok = ok ∧ l + 1 ≤ state.n ∧ ¬conf.isRef dd;
         n = state.n − (l + 1);
         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. 1.** The algorithm implementation of the move primitive for gen_gc_partial.

The HOL function gen_gc_full_move implements the move primitive for the full generational collection; its definition is elided for space reasons. It is similar

to gen_gc_partial_move, but differs in two main ways: first, it 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. Note that 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 $heap_2$; and that for every data element $d$ at address $a\ \in$ domain $f$, every address reachable from $d$ is also in the domain of $f$, and $f\ a$ points to a data element that is exactly $d$ with all its pointers updated according to f. Separately, we require that the roots are in domain $f$.

```
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 [10]. The main correctness theorem is as follows; we will not give further details of its proofs in this paper; for such proofs see [10].

$$\begin{array}{l}\vdash \mathsf{roots_ok}\ \mathit{roots}\ \mathit{heap} \wedge \mathsf{heap_ok}\ \mathit{heap}\ \mathit{conf}.\mathsf{limit} \Rightarrow \\ \quad \exists\, \mathit{state}\ f. \\ \quad\quad \mathsf{gen_gc_full}\ \mathit{conf}\ (\mathit{roots},\mathit{heap}) = (\mathsf{addr_map}\ (\mathsf{apply}\ f)\ \mathit{roots},\mathit{state}) \wedge \\ \quad\quad (\forall\, \mathit{ptr}\ u.\ \mathsf{mem}\ (\mathsf{Pointer}\ \mathit{ptr}\ u)\ \mathit{roots} \Rightarrow \mathit{ptr} \in \mathsf{domain}\ f) \wedge \\ \quad\quad \mathsf{gc_related}\ f\ \mathit{heap}\ (\mathit{state}.\mathsf{h1} +\!\!+ \mathsf{heap_expand}\ \mathit{state}.\mathsf{n} +\!\!+ \mathit{state}.\mathsf{r1})\end{array}$$

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-forward 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 includes the roots (and hence, by definition of gc_related, all live elements).

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.

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:

$$(\alpha,\ \beta)\ \mathsf{data_sort}\ =\ \mathsf{Protected}\ \alpha \mid \mathsf{Real}\ \beta$$

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:

$$\begin{array}{l}\mathsf{to_gen_heap_address}\ \mathit{conf}\ (\mathsf{Data}\ a) = \mathsf{Data}\ (\mathsf{Real}\ a) \\ \mathsf{to_gen_heap_address}\ \mathit{conf}\ (\mathsf{Pointer}\ \mathit{ptr}\ a) = \\ \quad \mathsf{if}\ \mathit{ptr} < \mathit{conf}.\mathsf{gen_start}\ \mathsf{then}\ \mathsf{Data}\ (\mathsf{Protected}\ (\mathsf{Pointer}\ \mathit{ptr}\ a)) \\ \quad \mathsf{else\ if}\ \mathit{conf}.\mathsf{refs_start} \leq \mathit{ptr}\ \mathsf{then}\ \mathsf{Data}\ (\mathsf{Protected}\ (\mathsf{Pointer}\ \mathit{ptr}\ a)) \\ \quad \mathsf{else}\ \mathsf{Pointer}\ (\mathit{ptr} - \mathit{conf}.\mathsf{gen_start})\ (\mathsf{Real}\ a)\end{array}$$

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 encoding than with semantic properties of the collector as such. 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 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 concrete finite memory (WordLang). Here the garbage collector's role is to automate memory deallocation and to implement the illusion of an unbounded memory.

This section sketches how the collector algorithm's types get instantiated, how the data refinement is specified, and how an implementation of the garbage collector algorithm is verified.

### 4.1 Instantiating the Algorithm's Types

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

$$\begin{aligned}\mathsf{v} \;=\; & \mathsf{Number\ int} \mid \mathsf{Word64}\ (64\ \mathsf{word}) \mid \mathsf{Block\ num}\ (\mathsf{v\ list}) \\ & \mid \mathsf{CodePtr\ num} \mid \mathsf{RefPtr\ num}\end{aligned}$$

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 $w$, or a code location Loc $l_1$ $l_2$.

$$\alpha\ \mathsf{word_loc} \;=\; \mathsf{Word}\ (\alpha\ \mathsf{word}) \mid \mathsf{Loc\ num\ num}$$

In what follows we will show through an example how an instance of $v$ is represented. We would have liked to provide more detail, but the definitions involved are simply too verbose to be included here. We will use the following DataLang value as our running example.

$$\mathsf{Block}\ 3\ [\mathsf{Number}\ 5;\ \mathsf{Number}\ 80000000000000]$$

The relation v_inv specifies how values of type v relate to the heap_addresses and heaps that the garbage collection algorithms operate on. Below is the Number case from the definition of v_inv. If integer $i$ 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 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 $i$.

$$\begin{aligned}&\mathsf{v_inv}\ \mathit{conf}\ (\mathsf{Number}\ i)\ (x,f,\mathit{heap}) \iff \\ &\quad \mathsf{if\ small_int}\ (:\alpha)\ i\ \mathsf{then}\ x = \mathsf{Data}\ (\mathsf{Word}\ (\mathsf{Smallnum}\ i)) \\ &\quad \mathsf{else} \\ &\qquad \exists\, \mathit{ptr}. \\ &\qquad\quad x = \mathsf{Pointer}\ \mathit{ptr}\ (\mathsf{Word}\ 0w) \wedge \\ &\qquad\quad \mathsf{heap_lookup}\ \mathit{ptr}\ \mathit{heap} = \mathsf{Some}\ (\mathsf{Bignum}\ i)\end{aligned}$$

$$\begin{aligned}&\mathsf{Bignum}\ i = \\ &\quad \mathsf{let}\ (\mathit{sign},\mathit{payload}) \;=\; \mathsf{sign_and_words_of_integer}\ i \\ &\quad \mathsf{in} \\ &\qquad \mathsf{DataElement}\ []\ (\mathsf{length}\ \mathit{payload})\ (\mathsf{NumTag}\ \mathit{sign},\mathsf{map\ Word}\ \mathit{payload})\end{aligned}$$

In the definition of v_inv, $f$ is a finite map that specifies how semantic location values for reference pointers (RefPtr) are to be represented as addresses.

$$\begin{aligned}&\mathsf{v_inv}\ \mathit{conf}\ (\mathsf{RefPtr}\ n)\ (x,f,\mathit{heap}) \iff \\ &\quad x = \mathsf{Pointer}\ (\mathsf{apply}\ f\ n)\ (\mathsf{Word}\ 0w) \wedge n \in \mathsf{domain}\ f\end{aligned}$$

The Block case below shows how constructors and tuples, Blocks, are represented.

```
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)
```

When v_inv is expanded for the case of our running example, we get the following 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 bignum 80000000000000. Here we assume that the architecture has 32-bit machine words. 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. Such information is used to speed up pattern matching. If the information fits into the lower bits of the pointer, then the pattern matcher does not need to follow the pointer to know whether there is a match.

```
⊢ v_inv conf (Block 3 [Number 5; Number 80000000000000]) (x,f,heap) ⟺
    ∃ ptr1 ptr2.
      x = Pointer ptr1 (Word (ptr_bits conf 3 2)) ∧
      heap_lookup ptr1 heap =
        Some
          (DataElement [Data (Word (Smallnum 5)); Pointer ptr2 (Word 0w)] 2
            (BlockTag 3,[])) ∧
      heap_lookup ptr2 heap = Some (Bignum 80000000000000)
```

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

```
⊢ v_inv conf (Block 3 [Number 5; Number 80000000000000])
    (Pointer 0 (Word (ptr_bits conf 3 2)),f,
     [DataElement [Data (Word (Smallnum 5)); Pointer 3 (Word 0w)] 2
       (BlockTag 3,[]); Bignum 80000000000000])
```

As we know, the garbage collector moves heap elements and changes the 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.

```
⊢ gc_related g heap1 heap2 ∧ (∀ ptr u. x = Pointer ptr u ⇒ ptr ∈ domain g) ∧
  v_inv conf w (x,f,heap1) ⇒
    v_inv conf w (addr_apply (apply g) x,g ∘ f,heap2)
```

Here addr_apply $f$ (Pointer $x$ $d$) = Pointer ($f$ $x$) $d$.

### 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, $\alpha$ word $\rightarrow \alpha$ word_loc, and an address domain set.

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

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

Using these, we define word_heap $a$ *heaf conf* to assert that a heap_element list *heap* is in memory, starting at address $a$, and word_el asserts the same thing about individual heap_elements. Figure 2 shows an expansion of the word_heap assertion applied to our running example.

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

**Fig. 2.** Running example expanded to concrete memory assertion

### 4.3 Implementing the Garbage Collector

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. The following is an excerpt of the theorem relating gen_gc_partial_move with its refinement word_gen_gc_partial_move. This 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.

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

## 5 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 [10]. The same verified algorithm was used for a verified Lisp implementation [11] 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 [16] and Ironclad Apps [7] 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 [8] and McCreight [9]; recent work by Gammie et al. [4] is also particularly impressive.

Hawblitzel and Petrank [8] 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 [9] 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. [4] verify a detailed model of a state-of-the-art concurrent collector in Isabelle/HOL, with respect to an x86-TSO memory model.

Pavlovic et al. [13] 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 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. [6,9,12,14]) and concurrent ones (e.g. [3–5,13]), but we believe that this paper is the first to report on a successful verification of a generational garbage collector.

## 6 Summary

This paper describes how a generational copying garbage collector has been proved correct and integrated into the verified CakeML compiler. The algorithm-level part of the 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.

*What We Did Not Do.* The current implementation lacks support for (a) nested nursery generations, and (b) the ability to switch garbage collector mode (e.g. from non-generational to generational, or adjust the size of the nursery) midway through execution of the application program. We expect both extensions to fit within the approach taken in this paper and neither to require modification of the algorithm-level proofs. For (a), one would keep track of multiple nursery starting points in the immutable part of the heap. These parts are left untouched by collections of the inner nursery generations. For (b), one could run a full

generational collection to introduce the special heap layout when necessary. This is possible since the correctness theorem for gen_gc_full does not assume that the references are at the top end of the heap when it starts.

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

## 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, J., Myreen, M.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. 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 (2015)
5. Gonthier, G.: Verifying the safety of a practical concurrent garbage collector. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 462–465. Springer, Heidelberg (1996). doi:10.1007/3-540-61474-5_103
6. Havelund, K.: Mechanical verification of a garbage collector. In: Rolim, J., et al. (eds.) IPPS 1999. LNCS, vol. 1586, pp. 1258–1283. Springer, Heidelberg (1999). doi:10.1007/BFb0098007
7. 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. USENIX Association, Broomfield (2014)
8. Hawblitzel, C., Petrank, E.: Automated verification of practical garbage collectors. In: ACM SIGPLAN Notices, vol. 44, no. 1, pp. 441–453 (2009). http://dl.acm.org/citation.cfm?id=1480935
9. McCreight, A.: The Mechanized Verification of Garbage Collector Implementations. Ph.D. thesis, Yale University, December 2008
10. Myreen, M.O.: Reusable verification of a copying collector. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 142–156. Springer, Heidelberg (2010). doi:10.1007/978-3-642-15057-9_10
11. Myreen, M.O., Davis, J.: A verified runtime for a verified theorem prover. In: Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 265–280. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22863-6_20
12. Nieto, L.P., Esparza, J.: Verifying single and multi-mutator garbage collectors with Owicki-Gries in Isabelle/HOL. In: Nielsen, M., Rovan, B. (eds.) MFCS 2000. LNCS, vol. 1893, pp. 619–628. Springer, Heidelberg (2000). doi:10.1007/3-540-44612-5_57
13. Pavlovic, D., Pepper, P., Smith, D.R.: Formal derivation of concurrent garbage collectors. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) MPC 2010. LNCS, vol. 6120, pp. 353–376. Springer, Heidelberg (2010). doi:10.1007/978-3-642-13321-3_20

14. Russinoff, D.M.: A mechanically verified incremental garbage collector. Formal Aspects Comput. **6**(4), 359–390 (1994)
15. 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)
16. Yang, J., Hawblitzel, C.: Safe to the last instruction: automated verification of a type-safe operating system. In: Programming Language Design and Implementation (PLDI), pp. 99–110. ACM, New York (2010)