Journal of Automated Reasoning (2025) 69:11
https://doi.org/10.1007/s10817-025-09721-0

A Mechanically Verified Garbage Collector for OCaml

Sheera Shamsu[1] · Dipesh Kafle[2] · Dhruv Maroo[1] · Kartik Nagar[1] · Karthikeyan Bhargavan[3] · KC Sivaramakrishnan[1,4]

Received: 2 July 2024 / Accepted: 24 February 2025

Abstract

The OCaml programming language finds application across diverse domains, including systems programming, web development, scientific computing, formal verification, and symbolic mathematics. OCaml is a memory-safe programming language that uses a garbage collector (GC) to free unreachable memory. It features a low-latency, high-performance GC, tuned for functional programming. The GC has two generations—a minor heap collected using a copying collector and a major heap collected using an incremental mark-and-sweep collector. Alongside the intricacies of an efficient GC design, OCaml compiler uses efficient object representations for some object classes, such as interior pointers for supporting mutually recursive functions, which further complicates the GC design. The GC is a critical component of the OCaml runtime system, and its correctness is essential for the safety of OCaml programs. In this paper, we propose a strategy for crafting a correct, proof-oriented GC from scratch, designed to evolve over time with additional language features. Our approach neatly separates abstract GC correctness from OCaml-specific GC correctness, offering the ability to integrate further GC optimizations, while preserving core abstract GC correctness. As an initial step to demonstrate the viability of our approach, we have developed a verified stop-the-world mark-and-sweep GC for OCaml. The approach is fully mechanized in F* and its low-level subset Low*. We use the KaRaMel compiler to compile Low* to C, and

✉ Sheera Shamsu
sheera.shms@gmail.com

Dipesh Kafle
dipesh.kaphle111@gmail.com

Dhruv Maroo
dhruvsmaroo@gmail.com

Kartik Nagar
nagark@cse.iitm.ac.in

Karthikeyan Bhargavan
karthik.bhargavan@gmail.com

KC Sivaramakrishnan
kcsrk@cse.iitm.ac.in

1 IIT Madras, Chennai 600036, India

2 NIT Trichy, Trichy 620015, India

3 Inria, 75014 Paris, France

4 Tarides, Chennai, India

Published online: 14 May 2025

integrate the verified GC with the OCaml runtime. Our GC is evaluated against off-the-shelf OCaml GC and Boehm–Demers–Weiser conservative GC, and the experimental results show that verified OCaml GC is competitive with the standard OCaml GC.

Keywords Formal verification · Mark and sweep garbage collection · F* · Low* · Mechanized formal proofs · Graph traversal proofs

1 Introduction

Many contemporary programming languages, including OCaml, utilize a garbage collector (GC) to manage memory automatically. This reliance on automatic memory management ensures memory safety, effectively preventing the occurrence of many security vulnerabilities [26, 35]. However, it is worth noting that the GC itself is often implemented in a language like C, which lacks inherent memory safety guarantees. Additionally, memory managers for modern languages often feature complex functionalities such as multiple generations, diverse memory layout for supporting different language features, incremental collection, and concurrency. These complexities make it challenging to ascertain the correctness of GC implementations, often resulting in the introduction of memory safety bugs.

The GC used in OCaml version 4 is generational and features two heap generations: the minor and major heaps. The minor heap employs copying collection, while the major heap utilizes an incremental mark and sweep GC to automatically reclaim memory. Both the minor and the major GC is implemented in C. Given that the memory safety of OCaml depends on the correctness of the GC, we wondered whether we could formally verify the correctness of the OCaml GC. Some previous works [10, 39] have verified the correctness of abstract GC models, which risk missing out on subtle bugs due to the air gap between the abstract model and the GC implementation. Our goal in this work is to develop a verified GC for OCaml, through a proof-oriented approach, such that executable code compatible with the OCaml compiler can be extracted directly from the verification artifact.

Rather than undertake the daunting task of verifying the full functional correctness of the existing OCaml GC in C, we have chosen to develop the verified GC from scratch in a proof-oriented language. We start from a feature complete GC that can run OCaml programs, but one which lacks the optimizations and features of the existing OCaml GC, and aim to incrementally enhance this GC with more features. To support this evolution, we have structured our verification approach such that the core correctness conditions for the GC need minimal changes throughout the enhancements.

At its core, garbage collection relies on accurately identifying objects designated as garbage, regardless of the specific GC algorithm employed. In a tracing GC, the allocated objects and their interconnections form a graph, transforming the task of identifying garbage objects into a graph traversal problem. Starting from the root sets of program variables (stack, heap, and globals), solving the graph traversal problem essentially involves identifying all objects transitively reachable from the root set. These reachable objects are termed as *live* objects. In terms of garbage collection, it is imperative that a GC does not free any live objects, a requirement known as the *safety* or *soundness* property of a GC. Allocated objects which are unreachable are considered as garbage objects, and it is the responsibility of the GC to free them. This aspect is referred to as the *liveness* or *completeness* property of the GC.

In light of these observations, our GC correctness specifications are founded on abstract graph reachability, enabling us to specify the GC correctness without including the specifics of the GC implementation. This ensures that the GC can evolve to provide additional optimizations and incorporate more features without necessitating alterations to the core correctness specifications. There is a clear distinction between abstract GC correctness and OCaml-specific GC correctness, where the requirements can be managed in separate layers. Setting aside the functional aspects of the GC, it is crucial to ensure that the C implementation of the GC itself does not introduce any memory safety bugs. This mandates a third layer of separation focusing exclusively on the memory safety of the GC implementation, all the while maintaining the functional properties of the GC.

To manage the verification demands of each layer and to generate the C code corresponding to the verified GC implementation, our preferred tool is F* [23, 34]. F* is a proof-oriented, solver-assisted programming language, along with its low-level subset Low* [28]. F* enables the co-development of programs and their proofs of correctness with the help of a rich type system and offering facilities for type refinements. Low* streamlines the verification of low-level code by providing libraries that support machine integers, heap and stack allocated arrays, and the C memory model.

In summary, we have three distinct layers, each addressing a specific aspect essential for ensuring the overall correctness of the GC implementation as follows:

1. An abstract graph interface and a formally verified depth-first search layer (DFS) in F*, wherein the correctness of DFS is specified through inductively defined graph reachability.
2. A system-specific layer in F* that addresses the intricacies of the OCaml GC algorithm, such as the tricolor invariant [18], utilized for reasoning about the correctness of mark-and-sweep GCs. This *functional GC layer* serves to bridge the gap between the abstract graph-based specification and its practical implementation in C. In this layer, the GC is implemented to operate on OCaml-style object layout, which is crucial to integrate the GC with the rest of the OCaml runtime. Within this layer, we have illustrated the progression of a practical GC by commencing with a basic GC implementation and systematically integrating diverse memory layouts supporting different OCaml features.
3. A low-level layer in Low* responsible for verifying memory safety of the GC implementation. The GC code within this layer is extracted to C using a compiler known as KaRaMel [28].

To the best of our knowledge, ours is the first work to formally verify a complete end-to-end mark-and-sweep GC extractable to C for a full-fledged industrial-strength programming language. We have integrated the verified GC with the OCaml 4.14.1 compiler and the integrated GC is capable of running non-trivial OCaml programs. Our experimental results demonstrate that the verified GC is competitive with the existing OCaml GC in terms of performance.

While numerous previous works [3, 5, 11, 12, 15, 17, 32] have addressed the problem of GC verification, most have tended to focus exclusively on verifying abstract models of GC, instead of actual implementations. A comparison with the related works that are verified practical GC implementations or close to practical GC implementations are summarized in Table 1.

A notable example of a stop-the-world (STW) mark-and-sweep GC verification is the work of Hawblitzel et al. [16], who verify an assembly-level x86 implementation. However, their work lacks the portability offered by a C implementation, and it cannot address the intricacies emerging due to the OCaml memory layout and integration with the OCaml runtime system. Moreover, their specification is based on the invariants of the GC algorithm,

Table 1 Comparison with the related work

Author	Mode	Algo	Spec	Code	Heap ayout
Hawblitzel et al. [16]	stw	mark & sweep	algo. specific	assembly	C#
Hawblitzel et al. [16]	stw	copying	algo. specific	assembly	C#
Ericsson et al. [6]	generational	copying	reachability	assembly	CakeML
McCreight [25]	incremental	copying	reachability	assembly	–
Gammie et al. [10]	concurrent	mark & sweep	reachability	model only	–
Zakowski et al. [39]	concurrent	mark & sweep	reachability	model only	–
Our work	stw	mark & sweep	reachability	C	OCaml

whereas our specification is based on abstract graph reachability. As mentioned earlier, the specifications based on abstract reachability gives us more flexibility to extend the GC correctness conditions to other GC algorithms. The verified copying collector by Ericsson et al. [6], tied to the CakeML compiler, is another notable work due to their integration of the verified GC with the rest of the CakeML runtime. However, mark-and-sweep GCs require a completely different form of reasoning as compared to copying collectors. One of the main highlights of our work is that it deals with the verification of a mark and sweep GC operating on OCaml-style objects, as the alignment with OCaml object layout is an essential factor for integrating the GC with the rest of the OCaml runtime. McCreight et al. [25] verify incremental copying collectors implemented in MIPS-like assembly language. The verification is through a common framework based on ADTs, which are later refined by various collectors. Gammie et al. [10] and Zakowski et al. [39] verify a concurrent mark-and-sweep GC over a detailed execution model, but they do not generate a verified executable code which can be integrated with the rest of the runtime. A more detailed discussion of the related work is presented in Sect. 9. While our work utilizes many of the ideas proposed in previous works, this is the first end to end verified and portable GC implementation integrated with the OCaml runtime environment. We view this work as a significant milestone in the journey towards establishing a highly performant, verified, robust GC for OCaml.

The rest of the paper is structured as follows. Sect. 2 offers an overview of OCaml memory management, tricolor mark and sweep garbage collection, and provides an introduction to F* and Low*. Sect. 3 outlines the abstract GC correctness specifications, and Sect. 4 describes the path towards a verified OCaml GC. Sect. 5 is dedicated to OCaml-specific GC correctness specifications. Sect. 6 elaborates on the layered design of our specification framework and the proof strategies employed in each layer. The benchmarks and experimental evaluation are presented in Sect. 7. In Sect. 8, we discuss how our approach can be extended for copying and incremental collection, thus laying a roadmap towards extensions of our verified GC. Sect. 9 examines related work, while Sect. 10 summarizes the conclusions drawn from our work and outlines potential future research directions.

2 Background

In this section, we present some background information on the OCaml object layout and memory manager, and the F* and Low* programming languages. The memory manager that we describe corresponds to the GC in OCaml version 4.14.1. OCaml 5 has introduced a

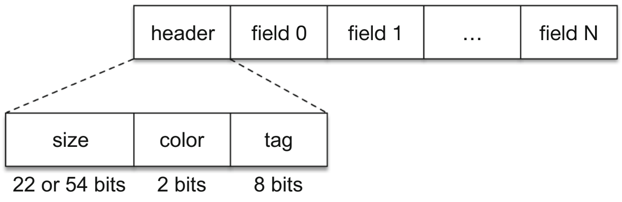

Fig. 1 Layout of an OCaml object. The header and each field of an OCaml object occupy one word

concurrent and a parallel GC [33], the details of which we omit as it is not in the scope of the current work.

The OCaml uses a uniform memory representation for OCaml values. A value is a single memory word that either represents an immediate integer or a pointer to some other memory. The OCaml runtime, written in C, manages the OCaml heap. The heap is a collection of memory regions obtained from the operating system in which OCaml objects reside. OCaml uses a *generational* GC with a small, fixed-size *minor* heap into which new objects are allocated. When the minor heap becomes full, it is evacuated with a copying collector to a large *major* heap. The major heap is collected with an *incremental mark-and-sweep* GC.

Directly verifying the correctness of the existing OCaml GC would be a difficult task due to the complexities of the existing codebase. Our aim is to develop a correct-by-construction GC from scratch that would act as an alternate GC for OCaml. For that, we need to develop a verified GC that operates on a heap compatible with the OCaml object layout. We have adopted an incremental approach in the development of the GC, starting from a bare-bones stop-the-world mark and sweep GC that operates on OCaml style objects, and then incrementally adding enough features to be able to run OCaml programs. We now describe the OCaml object layout.

2.1 OCaml Object Layout

Every object in OCaml has a word-sized header in which meta-data about the object is stored [22]. A typical OCaml object is represented as a *block* in the OCaml heap, which has a header followed by variable number of fields. Fig. 1 shows the layout of an OCaml object. The header includes an 8-bit tag, 2 bits for the object color (encoding the four colors blue, white, grey, and black), with the rest of the most significant bits representing the object size in words. Every field of the object is also word-sized, which ensures that the pointers to objects are always word-aligned. Immediate values such as integers and booleans are also word-sized. Immediate values are encoded with their least significant bit (LSB) to be 1, with the rest of the bits encoding the value of the data type. Thus, OCaml integers are 31-bits and 63-bits long on 32-bit and 64-bit platforms. Pointers are always guaranteed to be word-aligned and have 0 as their LSB. While the representation is not compact, it simplifies the GC; by examining the LSB, the GC can decide whether the value is a pointer or an immediate.

Many OCaml language constructs are represented as objects in the heap. For example, variants with parameters, records, arrays, polymorphic variants, closures, floating-point numbers, etc. are all represented as objects in the heap. The tag bits in the header of an OCaml object is used, among other things, to determine whether the fields of the objects may contain pointers. In particular, for objects with tag greater or equal to No_scan_tag (251), the fields are all opaque bytes, and are not scanned by the GC. For example, OCaml strings have a tag of String_tag (252) and contain opaque bytes and never contain pointers.

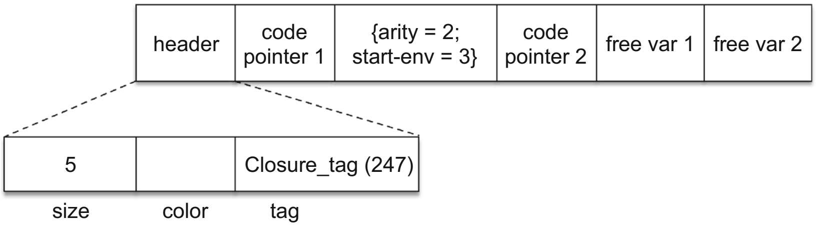

Fig. 2 The layout of a closure object with arity 2 and environment size 2

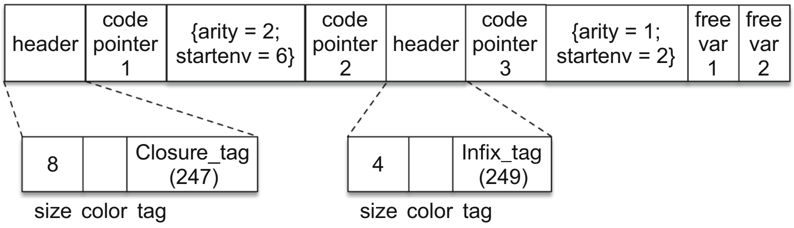

Fig. 3 The layout of a mutually-recursive closure object layout with arities 2 and 1 and environment size 2

If an object's tag is less than No_scan_tag (251), then the fields of the objects may be pointers. Among these, apart from Closure_tag (247) and Infix_tag (249) objects, the GC scans each field of the object to determine if it is a pointer or an immediate value and takes appropriate action.

A closure for a function or a set of mutually-recursive functions is a heap block with the following structure:

```
closure      ::= entrypoint (infix-header entrypoint)* value*
entrypoint  ::= code-pointer closure-info                // (with arity = 1)
              | code-pointer closure-info code-pointer // (with arity > 1)
closure-info ::=  arity (8 bits) . start-of-environment (wordsize - 9 bits) . 1
```

The values are the *environment* of the closure, which are the values of the free variables. Each entrypoint is either a 2- or 3- word record with the code pointer, closure information and, in the case of a closure with arity > 1, another code pointer. The closure information contains the arity of the closure. Importantly, the start of the environment information encodes the offset to the environment from the start of the closure. As an example, a closure with arity 2 and an environment of size 2 would have the following layout shown in Fig. 2. The start of environment information says that the environment starts from the field index 3 in the closure object. The GC only needs to scan the environment and uses the start of environment information to locate the environment in the closure.

Mutually recursive functions are represented as a closure object with one or more infix objects *within* the closure. Importantly, all the mutually recursive functions share the same environment. As an example, Fig. 3 shows a closure object with two mutually recursive functions of arities 2 and 1 and an environment of size 2. There are a few interesting things to note in this layout. First, the size of the closure object is 8, and it includes the infix object. While objects may point to the infix object, the infix object color is not used by the GC. Instead, the GC marks the parent closure object. The size of the infix object is 4, and it represents the offset (in words) of this object to the parent closure object. The GC uses this offset to locate the parent closure object.

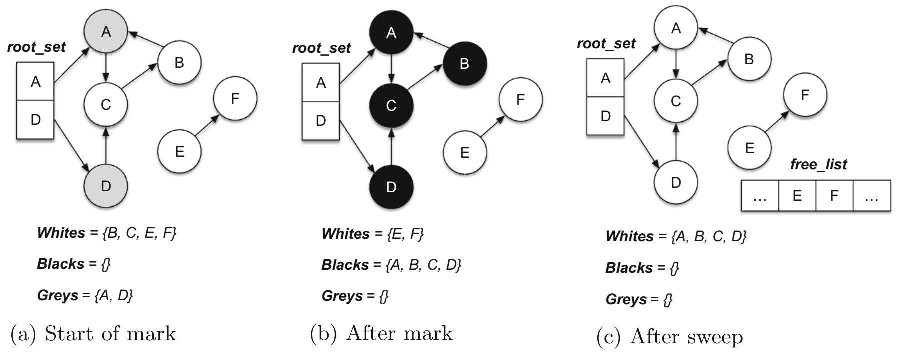

Fig. 4 Tricolor marking. Initially the objects that are directly pointed by the root pointers are grey in the heap. The stack is populated with such objects first. (Color figure online)

2.2 Tricolor Mark and Sweep GC

We now describe the details of a tricolor *stop-the-world* mark and sweep GC algorithm. As mentioned previously, our verified GC is based on this algorithm. Fig. 4 shows a run of the mark and sweep GC. The GC runs in two phases—mark and sweep. The mark phase performs a depth-first traversal of the object graph reachable from the *root set* of pointers—globals, stack and registers. At the start of the mark phase, all objects directly pointed from the root set are colored grey and are added to the *mark stack* (Fig. 4a).The mark phase uses a mark stack to store the objects that are discovered, but not yet fully explored. Objects which are free in the heap are colored blue and are maintained in a *free list*, a linked-list of free objects. Every other object is white. We have the invariant that all objects in the mark stack are grey and every live object is reachable from a grey object transitively through a sequence of white objects.

The mark phase proceeds by popping a grey object from the mark stack. All of the white successors of the popped object are marked grey and pushed onto the mark stack. Finally, the popped object is marked black. Thus, we also have the invariant that a black object never points to a white object. When the mark stack is empty, the mark phase ends (Fig. 4b). We are guaranteed that all the reachable objects are marked black, all unreachable objects remain white and there are no grey objects.

The sweep phase performs a linear traversal of the heap from the low address to the high address and examines each object. If the object is black, it is live. Sweep changes its color to white. If the object is white, it is dead. Sweep changes its color to blue and adds it to the free list. During sweep, we have the invariant that an object whose address is less than the traversal pointer is either white (live) or blue (free) and is on the free list. After sweep (Fig. 4c), we are guaranteed that all live objects are white and all unreachable objects are in the free list with color blue.

2.3 F* and Low*

Our correct-by-construction GC is implemented and verified in F* and its low-level subset Low*. F* is a general-purpose proof-oriented programming language, that supports both purely functional and effectful programs. In F*, the expressive power of dependent types is combined with proof automation based on SMT solving and tactic-based interactive theorem

```
module HA = FStar.HyperStack.All
module ST = FStar.HyperStack.ST
module B = LowStar.Buffer

let swap (r0 r1 : B.buffer UInt8.t)
  : HA.Stack (unit)
  (* PRE-CONDITIONS *)
  (* B.live predicate ensures that buffers must be allocated before
     their use *)
  (λ m -> B.live m r0 ∧ B.live m r1 ∧
     (* Unit length buffers *)
     B.length r0 == 1 ∧ B.length r1 == 1 ∧
     (* Buffer memory locations are not aliased *)
     B.loc_disjoint (B.loc_buffer r0) (B.loc_buffer r1))

  (* POST-CONDITIONS *)
  (λ m0 _ m1 ->  B.live m0 r0 ∧ B.live m1 r1 ∧
     (* Explicitly specify which memory locations is modified *)
     (B.modifies (B.loc_union (B.loc_buffer r0)
                  (B.loc_buffer r1)) m0 m1) ∧
     (* Encode functional correctness *)
     Seq.index (B.as_seq m1 r0) 0 == Seq.index (B.as_seq m0 r1) 0 ∧
     Seq.index (B.as_seq m1 r1) 0 == Seq.index (B.as_seq m0 r0) 0) =
      (* Initial memory m0 *)
       let m0 = ST.get() in
      (* Initial values in the single length buffers r0 and r1 *)
       let r0_val = !*r0 in
       let r1_val = !*r1 in
      (* Asserts that, if we convert the buffer to its funtional seq
         data type counter part, the value at index 0 is r0_val and
         similarly r1_val *)
      assert (Seq.index (B.as_seq m0 r0) 0 == r0_val);
      assert (Seq.index (B.as_seq m0 r1) 0 == r1_val);
      (* Updates r0 at index 0 as r1_val and r1 at index 0 as r0_val *)
       r0.(0ul) <- r1_val;
       r1.(0ul) <- r0_val;
      (* Get the new memory state after the swap *)
       let m1 = ST.get() in
      (* Asserts that the values are swapped *)
      assert (Seq.index (B.as_seq m1 r1) 0 == r0_val);
      assert (Seq.index (B.as_seq m1 r0) 0 == r1_val);
      (* Return type is unit, equivalent to C void *)
      ()
```

Fig. 5 A Low* code to swap the contents of two memory locations r_0 and r_1

proving. After verification, F* programs are usually extracted to OCaml or F#. The keyword val is used to define a function signature, whereas functions are defined using the keyword let (let rec for recursive functions). A variable x is declared in the form x:t, which means x has type t. F* provides support for refinement types, which helps to express more properties on the type of the variable. For instance, the type of non-negative integers, nat, is defined as n:int$\{n \geq 0\}$.

Fig. 6 Extracted C code from the Low* code in Fig. 5

```
void swap(uint8_t *r0, uint8_t *r1)
{
  uint8_t r0_val = *r0;
  uint8_t r1_val = *r1;
  r0[0U] = r1_val;
  r1[0U] = r0_val;
}
```

Low* [28] is a subset of F* with restricted features that allows a programmer to write verified low-level code that can be extracted to C. In Low*, the full expressiveness of F* can be used in proofs and specifications, while also exposing low-level details such as the memory layout which facilitates the development of verified, low-level code. The C code that is extracted from Low* after verification is free from low-level memory errors such as buffer overflows, use-after-free, etc. as these properties are formally verified as part of Low* pre-conditions and post-conditions.

To illustrate some of the features of Low* that we will use later in the paper, a sample Low* code to swap the contents of two memory locations r0 and r1 is shown in Fig. 5, along with its specification in the form of pre and post-conditions. We explain much of the Low* syntax through comments in the code. Low* operates on a C-like memory model with explicit heap and stack memory management, which is captured in the module FStar.HyperStack. FStar.HyperStack.ST.get() is used to obtain the contents of the heap memory at any program point. In Low*, C arrays are modeled using buffers, whose interface is defined in LowStar.Buffer module. Additionally, Low* provides support for machine integers of type 8, 32 or 64 bits. UInt8.t is the type of 8 bit machine integers.

The swap program takes as input, the buffers r0 and r1 of length 1, with the pre-conditions asserting that these buffers have been allocated space in memory and they are not aliases. Notice that the pre-condition takes as input the initial heap state as the argument m. The post-condition is specified over both the initial heap state m0 and the final heap state m1. For specifying functional correctness in the post-condition, we use the function LowStar.Buffer.as_seq which converts the buffer to its sequence counterpart (a sequence is just a functional list). We use the function Seq.index to obtain the element of a sequence at a given index. The post-condition asserts that the value stored at index 0 in the location r1 in the final heap is the value stored at index 0 in location r0 in the initial heap, and vice versa. The extracted C code from the Low* code is shown in Fig. 6.

3 Abstract GC Correctness

From the discussion in Sect. 2.2, it is evident that mark and sweep GC is primarily a graph algorithm. In particular, mark is a depth first traversal on the heap. Therefore, the correctness specification of garbage collection is most naturally expressed using graph theoretic terminology, rather than relying on the GC implementation details. The prime consideration for any GC is *soundness*—that is it only collects unreachable objects. A GC is said to be *complete* if it collects all the unreachable objects. We first formally define GC correctness in graph theoretic terms without any reference to the underlying implementation. We first define the construction of the object graph abstractly, without appealing to the details of the GC implementation. Later, we will instantiate these definitions for our verified OCaml GC.

Let h denote the heap and |h| represents the length of the heap in bytes. Let objs(h) to be the set of all objects in the heap identified by their unique ids. The id of an object depends on the implementation. For example, in OCaml, the id of an object is the address of the first field. Let allocs(h) denote the set of allocated (not free) objects represented by their ids in h. Let ptrs(x,h) be the set of ids of the objects pointed to by x. Let data(x,h) be the set of non-pointer, opaque data fields of x.

Definition 1 (*Well-formed heap*) A heap h is said to be well-formed, denoted by ω(h) iff ($\forall$ x, y. x $\in$ allocs(h) $\wedge$ y $\in$ ptrs(x,h) $\implies$ y $\in$ allocs(h))

Definition 2 (*Object graph*) An object graph G(h) = (V,E) is constructed from a well-formed heap h as follows: the vertex set V = allocs(h), and edge set E = {(x,y) | x $\in$ V $\wedge$ y $\in$ ptrs(x,h)}. The object graph is represented as G(h).

Definition 3 (*Accessibility relation*) Given x, y $\in$ allocs(h), x and y are related through the accessibility relation (denoted as x $\rightsquigarrow$ y) if and only if either (1) x = y or (2) $\exists$ z. z $\in$ allocs(h) $\wedge$ x $\rightsquigarrow$ z $\wedge$ y $\in$ ptrs(z,h).

Definition 4 (*Reachable sub-graph*) The reachable sub-graph RG(h,r) = (V_{RG}, E_{RG}) is formed from a well-formed heap h and a root-set r. Let G(h) = (V,E) be the graph constructed out of the heap h. Then,

- ($\forall$ x. x $\in$ V_{RG} $\iff$ x $\in$ V $\wedge$ ($\exists$ y. y $\in$ r $\wedge$ y $\rightsquigarrow$ x))
- ($\forall$ x y. (x,y) $\in$ E_{RG} $\iff$ x $\in$ V_{RG} $\wedge$ (x,y) $\in$ E)

That is, RG(h,r) only contains the accessible objects from r in h as vertices and the edges between accessible objects in h are preserved in RG(h,r).

Definition 5 (*GC correctness*) Let h_0 be the initial state of the heap on which the GC operates, such that $\omega(h_0)$ holds, and let r be the set of *roots*, which are pointers to objects into h_0. Let h_1 be the heap after the GC terminates and let V be the vertex set of G(h_1). Then, the GC is said to be correct if:

1. $\omega(h_1)$ holds.
2. G(h_1) = RG(h_0,r)
3. ($\forall$ x. x $\in$ V $\implies$ data(x,h_0) = data(x,h_1))

The GC correctness definition says that, after the GC, the heap remains *well-formed*. The object graph after the GC is equal to the sub-graph of accessible objects from r in h_0. This ensures that only the accessible objects are part of the object graph after the GC terminates, thereby ensuring completeness. Soundness is ensured as RG(h_0,r) retains all the reachable objects and their interconnections. Additionally, the third correctness property ensures that the non-pointer fields of accessible objects remain the same.

We note that this definition of GC correctness is generic and applicable across different types of GCs. For example, in a copying collector, while the data fields and the object graph remains the same, the object themselves are moved. The generic GC correctness definition is able to accommodate this since it does not claim to preserve value of the pointer fields across the GC. We note that the main correctness theorem of [6], which is a verified copying collector for CakeML also captures GC correctness similar to Definition 5. In Sect. 8, we present the abstract correctness specifications for a copying collector as well as an incremental mark and sweep GC that uses a snapshot-at-the-beginning deletion barrier [38].

With the generic GC correctness specification in place, let us now move on to the implementation of an actual mark and sweep GC for OCaml in the next section. With the help of the implementation, we show how the generic specifications are adapted specifically for OCaml and the mark and sweep GC (Sect. 5).

```
void mark_and_sweep_GC (uint8_t *hp, uint64_t *st, uint64_t *tp,
                        uint64_t *r, uint64_t r_len, uint64_t *sw,
                        uint64_t *fp) {
  // GC initialization phase starts with pushing of roots
  // into the mark stack
  darken_roots (hp, st, tp, r, r_len);
  // GC mark phase is dfs that operates on different OCaml objects
  mark (hp, st, tp);
  // GC sweep phase frees unreachable objects and updates the free list
  sweep (hp, sw, fp);
}
```

Fig. 7 Extract C code for the top-level stop-the-world mark-and-sweep GC function. (Color figure online)

4 Towards a Verified OCaml GC

In this section, we present our approach to verify a practical GC for OCaml. As mentioned in Sect. 2, OCaml uses a generational and incremental GC, aimed at supporting high allocation rates and low latency. Given that verifying such a GC implementation is a challenging task, we develop a verified stop-the-world mark-and-sweep GC in a proof-oriented manner. We show in Sect. 8 how this GC may be extended to support copying and incremental mark-and-sweep collection.

Our task involves connecting the abstract graph reachability specification defined in the previous section with performant C code, that involves low-level operations such as pointer arithmetic and bitwise operations. Since our aim is to integrate the verified GC with the rest of the OCaml compiler, our verified GC must be made aware of the different object layouts used by the compiler. We adopt a layered approach for verification similar to [6, 27]. The layered approach allows us to cleanly separate the abstract graph-based correctness from the low-level operations and language-specific features. We present the evolution of a proof-oriented practical GC for OCaml, by starting with a base GC model and then progressively adding essential features until it is sufficient to integrate the GC with the rest of the OCaml compiler.

As we mentioned in Sect. 1, the verified GC code written in Low* can be extracted to C. Fig. 7 shows the top-level GC function extracted from the verified GC code on a 64-bit platform. No change has been made apart from renaming the functions for readability. Our heap hp is a single, contiguous, fixed-size byte buffer of size heap_size. The GC takes as inputs an array of roots r of length r_len, a mark stack array st with a stack top pointer tp that indexes into the stack, a sweep pointer sw and a free list pointer fp. Note that tp, sw and fp are singleton buffers containing the stack top pointer, sweep pointer and free list pointer respectively. The free list is a singly linked list of free objects in the heap, which is implicitly stored through the fields of the objects. Recall that free objects are colored blue. Like OCaml, we assume that zero length objects are not on the heap. This implies that each object has at least one field, which we use to store the next pointer for the free list. Initially, st is empty, tp points to the stack base address, sw points to the start of the heap (i.e. hp), while fp points to the first blue object. The GC first calls darken_roots to grey all the roots in r and pushes them onto the mark stack st and suitably updates the stack top pointer tp.

Next is the mark function. We start with a base implementation first (Fig. 8), where there is no distinction between different types of objects. The implementation here is to enable us to establish the base invariants necessary to do the verification. Then we extend the base

```
void mark(uint8_t *hp, uint64_t *st, uint64_t *tp) {
  while (*tp > (uint64_t)0U)
    mark_body(hp, st, tp);
}

void mark_body(uint8_t *hp, uint64_t *st, uint64_t *tp) {
  tp[0U] = *tp - (uint64_t)1U; // Decrement tp
  uint64_t x = st[*tp];
  uint64_t h_x = hd_address (x);
  colorHeader (hp, h_x, black);
  uint64_t wz = wosize (h_x, hp);
  darken (hp, st, tp, h_x, (uint64_t)1U);
}

void darken (uint8_t *hp, uint64_t *st,
             uint64_t *tp, uint64_t h_addr, uint64_t j) {
  uint64_t wz = wosize (h_addr, hp);
  for (uint32_t i = j; i < (wz + (uint64_t)1U)); i++) {
    darken_body(hp, st, tp, h_addr, i);
  }
}

void darken_body (uint8_t *hp, uint64_t *st, uint64_t *tp,
                  uint64_t h_addr, uint64_t i) {
  uint64_t succ_indx = h_addr + i * mword;
  uint64_t succ = load64 (hp + succ_indx);
  uint64_t c = color (hd_address (succ), hp);
  if (isPointer (succ_indx, hp)) {
    if (c == white) {
      push_to_stack(hp, st, tp, succ);
    }
  }
}
```

Fig. 8 Base version of `mark` and `darken` function

implementation to handle objects with No_scan_tag (Fig. 9) and finally closure and infix objects (Fig. 10).

Let us first discuss the base version of the mark function as shown in Fig. 8. The mark function repeatedly calls mark_body until the stack is empty. mark_body pops the object x from the top of the stack and finds its header address h_x using the function hd_address. Then the color bits of the value pointed by h_x are made black through colorHeader. wosize returns the object size in words stored at h_x and after which darken iterates through all the fields of x, calling darken_body on the fields. darken_body darkens the white objects (i.e. turning them grey) and pushes the field pointers onto the mark stack as necessary.

The code snippets in Fig. 8 hints at the verification challenge in front of us. Given that we are in C, we have to ensure the accesses are memory safe, i.e., all memory accesses are

Fig. 9 Version of mark function that deals with no_scan objects

```
void mark_body(..omitted...) {
  // Code omitted, same as before...
  uint64_t tg = tag (h_x, hp);
  if (tg < (uint64_t)251U) {
    darken (hp, st, tp, h_x, (uint64_t)1U);
  }
}
```

to valid memory. Observe that implementation works by coloring the header words with bitwise operations. Hence, we need to reason about the correctness of bitwise arithmetic, also ensuring that the change in color bits does not affect wosize and tag of the object, which are also stored in the same header word.

The version of mark function as shown in Fig. 9, incorporates the usage of the tag bits which are part of the OCaml object layout. Here, tag is used to determine whether the newly popped out object from mark stack needs to be scanned by the GC. Recall from Sect. 2.1 that any object with a tag greater than or equal to no_scan (value 251) is not scanned by the GC. In this case, the mark skips scanning this object and moves on to next object from the mark stack.

The third version of the mark function, shown in Fig. 10, further extends the marking process for objects with tag less than no_scan. In particular, it checks whether the object under consideration is a closure object or an infix object. mark_body calls darken_wrapper instead of darken, which decides the starting address of fields of the particular object under consideration. In the case of closure objects, as explained in Sect. 2, the offset of the environment need to be extracted first. The details of the extraction is not shown for brevity. Another change is in darken_body, where, if a field points to an infix_object, then the *parent* closure object is determined. This parent closure is the one that is darkened by the GC. We note that this goes beyond just a simple DFS traversal, and the details of these operations are necessary to reason about the correctness of the GC.

For simplifying the exposition of our verification process, we will use base version of mark throughout the rest of the paper. We note that our verified GC deals with closure and infix objects, and integrates with the rest of the OCaml compiler and the runtime. In Sect. 6.5, we expand upon the changes required to verify the implementation in Fig. 10.

After mark finishes, sweep scans the objects stored in the heap, starting from sw to the end of the heap. The extracted code for sweep is shown in Fig. 11. While scanning, sweep examines the color of the object. If the object is black, it is colored white and if the object is white or blue, the color is changed to blue and the object is added to the free list by making the first field of fp point to this object. Additionally, the current object pointed by sw is made to be the new fp. sweep remains the same across the different variants of mark.

5 OCaml GC Specification

We now instantiate the abstract GC correctness definition from Sect. 3 for our GC compatible with OCaml. We express this specification in F* as is done in our artifact.

```
void mark_body (..omitted...) {
  // Omitted....
  if (tg < (uint64_t)251U) {
    // Wrapper function for darken
    darken_wrapper(hp, st, tp, h_x);
  }
}

void darken_wrapper (..omitted...) {
  // If the object is closure objs
  if (tag(h_x, hp) == (uint64_t)247U) {
    uint64_t x = f_address(h_x);
    // Start of environment has to be extracted for closure objects
    uint64_t start_env = start_env_clos_info (hp, x);
    darken (hp, st, tp, h_x, start_env + (uint64_t)1U);
  } else {
    darken (hp, st, tp, h_x, (uint64_t)1U);
  }
}

// Darken remains the same as that in base version
void darken_body(...omitted...) {
  // Omitted...
  if (isPointer(succ_indx, hp)) {
    uint64_t h_addr_succ = hd_address(succ);
    uint64_t tg = tag (h_addr_succ,hp);
    // If the field points to an infix object
    if (tg == (uint64_t)249U) {
      // Finds the parent closure
      uint64_t parent_hdr = parent (hp, h_addr, i);
      darken_helper (hp, st, tp, parent_hdr);
    } else {
      darken_helper (hp, st, tp, h_addr_succ);
    }
 }
}

void darken_helper(...omitted...) {
  if (color(hdr_id, hp) == white) {
    push_to_stack (hp, st, tp, hdr_id);
  }
}
```

Fig. 10 Version of `mark` and `darken` function that deals with `closure` and `infix` objects

```
void sweep (uint8_t *g, uint64_t *sw, uint64_t *fp,
            uint64_t limit, uint64_t mword) {
  while (*sw < limit) {
    uint64_t curr_obj_ptr = *sw;
    uint64_t curr_header = hd_address(curr_obj_ptr);
    uint64_t wz = wosize_of_block(curr_header, g);
    uint64_t next_header =  curr_header + (wz + 1ULL) * mword;
    uint64_t next_obj_ptr = next_header + mword;
    sweep_body (g, sw, fp);
    sw[0U] = next_obj_ptr;
  }
}

void sweep_body (uint8_t *g, uint64_t *sw, uint64_t *fp) {
  uint64_t curr_obj_ptr = *sw;
  uint64_t curr_header = hd_address(curr_obj_ptr);
  uint64_t c = color_of_block(curr_header, g);
  uint64_t wz = wosize_of_block(curr_header, g);

  if (c == white || c == blue) {
    colorHeader(g, curr_header, blue);
    uint64_t fp_val = *fp;
    uint32_t x1 = fp_val;
    store64_le(g + x1, curr_obj_ptr);
    fp[0U] = curr_obj_ptr;
  } else {
    colorHeader(g, curr_header, white);
  }
}
```

Fig. 11 Extracted C code of `sweep` function implemented as an iterative function invoking the `sweep_body`

```
noeq type graph (#a:eqtype) = {
  vertices : v: vertex_set #a;
  (* [vertices] are a sequence of type a with no duplicates *)
  edges : e: edge_set #a {edge_ends_are_vertices vertices e}
  (* [edges] are a sequence of type (a,a) with no duplicates *)
}
```

Fig. 12 The `graph` type

5.1 Basic Definitions

We first define the object graph in F* in Fig. 12. The graph is defined as a record type in F* with two fields and is parametric over type a. Note that the prefix # before type a indicates that a is an *implicit* argument. The first field vertices has type vertex_set a and is a type alias of seq a with a type refinement that does not allow duplicates. seq is an unbounded array like data structure available in the F* standard library. The second field edges is defined to have

```
type reach: (g:graph) → (x:vertex) → (y:vertex) → Type =
   (* reachability is reflexive *)
   | ReachRefl  : (g:graph) → (x:vertex) → (reach g x x)
   (* reachability is transitive *)
   | ReachTrans : (g:graph) → (x:vertex) → (z:vertex) →
                  (reach g x z) →
                  (* [edge g z y] is a type refinement which mandates
                     that [(z,y)] is an edge in [g] *)
                  (y:vertex {edge g z y}) → (reach g x y)
```

Fig. 13 Graph reachability as an inductive predicate reach

type edge_set a, where edge_set a is a type alias for seq (a,a) with no duplicates. The type refinement on the edges which enforces that both the members of an edge should belong to the vertices of the graph.

In Fig. 13, we define the accessibility relation (Definition 3) as an inductive predicate reach. Note that vertex is any type a, and edge is a type alias for (vertex & vertex) (& is product type operator in F*). The reach g x y predicate encodes a proof of reachability from vertex x to vertex y in graph g. There are two ways to construct this proof: either through ReachRefl which encodes that every vertex is reachable from itself, or through ReachTrans, which requires a proof of reachability from x to z, and an edge in g from z to y, captured by the type refinement edge g z y.

Our basic definitions in F*, which are related to the OCaml heap, are shown in Fig. 14. We assume a 64-bit architecture. However, note that our framework is parametric over the machine word size. mword indicates the word size in bytes. We define heap_size to be an integer n such that n is a multiple of mword. The heap has enough space to store at least 1 object. Since the smallest object on the heap has one word header and one field, the smallest heap size is 16 bytes. We also need an upper bound on the heap size to prevent overflow when we perform arithmetic operations on the heap addresses. We choose the upper bound to be 1 TiB ($2^{40} = 1099511627776$ bytes), which is a pragmatic upper bound for OCaml programs.

We assume that the heap is densely packed with objects of any colour. Recall from Sect. 2.1 that the OCaml object header includes two bits in the header for color. Blue color represents a free object, whereas white, black or grey object represents an allocated object. Objects can have arbitrary sizes, encoded in the wosize bits of its header block. For example, a completely empty heap (devoid of any allocated objects) may have one blue object that spans the entire heap or may have successive blue objects that span the entire heap.

The heap type is defined as a sequence of 8-bit unsigned machine integers of length heap_size. A valid heap address hp_addr is defined as a 64-bit unsigned machine integer. The heap address is word aligned and points to a location within the heap. In OCaml, every object is represented by the address of its first field. Therefore, obj_addr represents an object address which has an additional restriction that the valid heap address should start from mword or greater indicated by the type refinements inside the curly brackets. Similar refinement is applied to the type hdr_addr which denotes a header address of the object. Since the objects on the heap have at least one field, the header address should be at least mword less than the heap size. The function hd_address takes the address of an object o and returns the header address of o.

We now describe a few definitions, which are not shown in the code. wosize_t, color_t and tag_t define the type of wosize, color and tag of an object, respectively. The functions

```
//Machine integers
module U64 = FStar.UInt64
module U8 = FStar.UInt8
let mword = 8UL
val heap_size : n:int{n `mod` U64.v mword == 0 ∧ n >=16 ∧
                      n < 1099511627776}
(* heap is a sequence of 8 bit unsigned machine integers *)
type heap = h:seq U8.t{length h == heap_size}
(* A valid heap address *)
type hp_addr = addr:U64.t {U64.v addr < heap_size  ∧
                           is_multiple_of_mword addr}
(* object address *)
type obj_addr = x:hp_addr {U64.v x >= mword})
(* header address *)
type hdr_addr = x:hp_addr {U64.v x + mword < heap_size})

(* header address from object address *)
let hd_address (o:obj_addr) = U64.sub o mword

(* object address from header address *)
let f_address (h:hdr_addr) = U64.add h mword

let valid_field_number (i:U64.t) (h:heap) (x:obj_addr) =
    i >= 1 ∧ i <= wosize(hd_address x, h)

let field_addr (x:obj_addr) (h:heap)
               (i:U64.t{valid_field_number i h x}) =
    U64.add (hd_address x) (U64.mul i mword)

(* Field reads of ith field of object x *)
let field (x:obj_addr) (h:heap)
          (i:U64.t{valid_field_number i h x}) =
    r_word h (field_addr x h i)

(* allocs(h) is the set of allocated objects in the heap *)
let well_formed_heap h =
    (∀ x. seq.mem x (allocs (h)) ⟹
       (∀ (i:U64.t{valid_field_number i h x}).
         isPointer (field_addr x h i) h ⟹
           (field x h i) ∈ allocs(h)))
```

Fig. 14 Basic definitions in F*

wosize h_x h, color h_x h and tag h_x h returns the wosize, color and tag respectively of the object x with header address h_x in heap h. The value stored at a heap address x in a heap h is read using a function r_word h x. Similarly, w_word h x writes to h in location specified by x.

Unlike the OCaml runtime, in the formalisation, for convenience, we address the fields from the header address of an object. Hence, the first field will have the offset 1. The function valid_field_number (shown in Fig. 14), checks whether a field number is valid. A field number i is valid only if it lies within the range of 1 and the wosize of the object. The function field reads the i^{th} field of object x in h, if i is a valid field number for the object x. We define a boolean predicate isPointer i h = U64.logand (r_word h i) 1UL = 0, which holds when the value at address i holds a pointer (the least-significant bit is 0). The predicate well_formed_heap is the instantiation of the abstract well-formed heap (Definition 1) defined in Sect. 3.

Using the above basic definitions, we now define some auxiliary functions. Given a heap h, objs h returns the sequence of object addresses in the heap h. It essentially scans the heap from the beginning, using the wosize of each object to move to the next object. allocs h returns the allocated object addresses in h (i.e. objects with a non-blue color). h_objs h is a sequence of the header addresses of all objects in h. Similarly, h_allocs h is the sequence of header addresses of allocated objects of h. Additionally we define blacks h, whites h, greys h and blues h to represent the sequence of header addresses of black, white, grey and blue objects respectively. The function valid_hdr takes a header address of an object and the heap and checks whether the header address is a part of h_allocs h.

5.2 Specification for GC Functions

With these definitions in place, let us see how we can specify the correctness of the GC (Definition 5) based on the correctness of the constituent functions darken_roots, mark and sweep introduced in Fig. 7. Certain useful algebraic properties of these functions as defined in F* are shown in Fig. 15. The type st_hp is a pair of seq obj_addr (representing the mark stack) and heap. The F* library functions fst and snd returns the first and second member of a pair respectively. Note that the heap before and after the GC contains only white and blue objects.

The function darken_roots recursively fills the stack with all object addresses specified in the root list r_list. It maintains the invariant that all objects in the stack are colored grey (pre-condition 3 and post-condition 4). darken_roots returns the modified stack and heap pair. In addition, the heap remains well-formed, and all fields of every object remains the same (post-conditions 1 and 3). The mark function also ensures the above properties, and additionally ensures that there are no grey objects after it finishes, essentially coloring all reachable objects black. The type of the return value of sweep is hp_fp, which is a pair of heap type and free list pointer type (obj_addr type). The sweep function ensures that there are no more black objects after it completes. Note that all these functions ensure that the heap remains well-formed, and there is no change to the object fields, effectively ensuring properties (1) and (3) in the definition of GC correctness (Definition 5). For proving property (2), we need to consider the reachability of objects in the underlying object graph.

```
(* Product type *)
type st_hp = seq obj_addr & heap
type hp_fp = heap & obj_addr

val darken_roots (h:heap) (st:seq obj_addr) (r_list:seq obj_addr)
  : Pure (st_hp)
  (requires (* Only core conditions shown *)
  (*1*) well_formed_heap (h) ∧
  (*2*) (∀ x. x ∈ h_objs(h)  ⟹  (x ∈ whites(h) ∨ (x ∈ blues(h)) ∧
  (*3*) (∀ x. x ∈ st  ⟺  hd_address x ∈ greys(h)))
  (ensures  (* Only core conditions shown *)
  (*1*) (λ res → well_formed_heap (snd res) ∧
  (*2*) (∀ x. x ∈ r_list  ⟹  x ∈ (fst res)) ∧
  (*3*) (∀ x i. (hd_address x) ∈ h_objs(h)  ⟹
                    field x h i = field x (snd res) i)) ∧
  (*4*) (∀ x. x ∈ (fst res)  ⟺  hd_address x ∈ greys(snd res)))

val mark (h:heap) (st:seq obj_addr)
  : Pure (heap)
  (requires (* Only core conditions shown *)
  (*1*) well_formed_heap (h) ∧
  (*2*) (∀ x.x ∈ st  ⟺  hd_address x ∈ greys(h)))
  (ensures  (* Only core conditions shown *)
  (*1*) (λ h₁ → well_formed_heap (h₁) ∧
  (*2*) (∀ x i. (hd_address x) ∈ h_objs(h)  ⟹
                    field x h i = field x h₁ i) ∧
  (*3*) (∀ x.x ∈ h_objs(h₁)  ⟹  (color (hd_address x h)₁ ≠ grey)))

val sweep (h:heap) (curr_ptr:obj_addr) (fp:obj_addr)
  : Pure (hp_fp)
  (requires (* Only core conditions shown *)
  (*1*) well_formed_heap (h) ∧
  (*2*) (∀ x.x ∈ objs(h)  ⟹  (color (hd_address x h) ≠ grey)) ∧
  (ensures  (* Only core conditions shown *)
  (*1*) (λ h₁, fp₁ → well_formed_heap (h₁) ∧
  (*2*) (∀ x. x ∈ blacks(h)  ⟺  x ∈ whites(h₁)) ∧
  (*3*) (∀ x. x ∈ whites(h) ∨ blues(h)  ⟺  x ∈ blues(h₁)) ∧
  (*4*) (∀ x i. (hd_address x) ∈ h_allocs(h)  ⟹
                    field x h i = field x h₁ i) ∧
  (*5*) (∀ x. x ∈ h_objs(h₁)  ⟹  x ∈ whites(h₁) ∨ x ∈ blues(h₁)))
```

Fig. 15 Algebraic properties of the constituent functions of our verified GC

5.3 Object Graph Construction

The construction of object graph from the heap crucially depends on the well-formedness of the heap (well_formed_heap defined in Fig. 14). Well-formed heap requires that pointers from allocated objects should only refer to other allocated objects. We assume that the allocator and the mutator (the OCaml program) maintain this invariant.

```
module G = Spec.Graph
val edges_of_graph (s:seq obj_addr) (h:heap{well_formed_heap (h)})
      : Tot (e:G.edge_set{∀ x y. mem x s  ⟹   mem (x,y) e  ⟺
                                (∃ i. valid_field_number i h x ∧
                                      isPointer (field_addr x h i) h ∧
                                      y = field x h i)})
val graph_from_heap (h:heap)
      : Pure (G.graph)
        (requires  well_formed_heap (h))
        (ensures  λ g→ g.vertices = allocs h ∧
                       g.edges = edges_of_graph allocs h)
```

Fig. 16 Constructing graph from the heap

```
val mark_reachability_lemma (h:heap) (st:seq obj_addr)
                            (r_list:seq obj_addr)
   : Lemma
     (requires
     (*1*) well_formed_heap (h)
     (*2*) (∀ x. x ∈ st  ⟺  hd_address x ∈ greys(h)) ∧
     (*3*) well_formed_heap (mark h st))

     (ensures
     (*1*) (graph_from_heap (mark h st) = graph_from_heap h) ∧
     (*2*) (∀ x y.y ∈ r_list ∧
              reach (graph_from_heap h) y x  ⟺  x ∈ blacks(mark h st)))


val sweep_subgraph_lemma (h:heap) (r_list:seq obj_addr)
                         (curr_ptr:obj_addr) (fp:obj_addr)
   : Lemma
     (requires
     (*1*) well_formed_heap (h) ∧
     (*2*) (∀ x.x ∈ h_objs(h)  ⟹  (color (hd_address x h) ≠ grey)) ∧
     (*3*) well_formed_heap (sweep h curr_ptr fp))

     (ensures
     (*1*) (∀ x. x ∈ graph_from_heap (sweep h curr_ptr fp).vertices  ⟺
                x ∈ graph_from_heap (h).vertices ∧
                (∃ y. y ∈ r_list ∧ reach (graph_from_heap h) y x)) ∧

     (*2*) (∀ x y.
             (x,y) ∈ graph_from_heap (sweep h curr_ptr fp).edges  ⟺
                x ∈ graph_from_heap (sweep h curr_ptr fp).vertices ∧
                y ∈ graph_from_heap (sweep h curr_ptr fp).vertices ∧
                (x,y) ∈ graph_from_heap (h).edges))
```

Fig. 17 Mark reachability and reachable subgraph preserving lemma

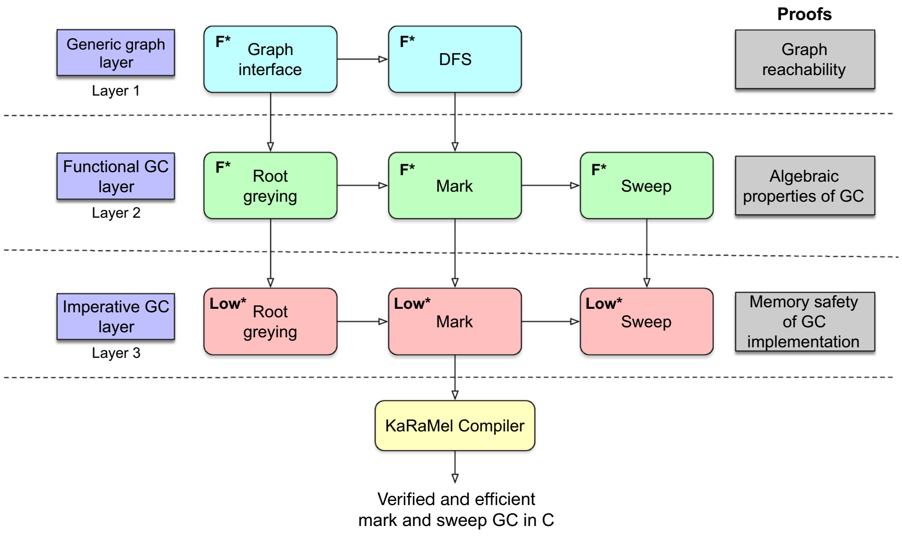

Fig. 18 The layering approach for GC verification

OCaml features such as closure and infix objects also affect graph construction, as these objects have a different layouts to regular objects and influence how the GC scans the objects. For simplifying the presentation, the graph construction of Fig. 16 ignores closure and infix objects and objects which only have opaque bytes. The details on how to incorporate the additional features is described in Sect. 6.5. As shown in Fig. 16, the vertices of the graph are simply the set of allocated objects allocs h, while the edges are constructed using a function edges_of_graph, that takes as inputs allocs h and h and creates pairs (x,y) for all x in allocs h such that y is a field pointer of x in h.

Using the graph definitions, we can now specify the additional properties required for proving property (2) in the abstract GC correctness definition. As shown in Fig. 17, the mark_reachability_lemma ensures that the object graph constructed from the heap remains the same before and after mark. In addition, all the objects and only the objects that are reachable from the root list r_list in the object graph will be colored black in the output heap after mark. Note the use of the inductive reach predicate from Fig. 13 in this specification. Notice also that the precondition (2) requires that all objects in the stack are colored grey in the input heap, which is essentially a post-condition of darken_roots in Fig. 15. The sweep correctness conditions are listed in sweep_subgraph_lemma. The post-conditions of the lemma ensure that the graph formed after sweep has only reachable objects and their interconnections, that is the reachable sub-graph of the original graph before the GC. The fact that sweep is the last operation of the GC ensures that the final graph after the GC is the reachable subgraph of the graph before the GC, thus proving property (2) of Definition 5.

As evident from the specifications, there are three different dimensions of reasoning required for verifying correctness: (i) first, we must relate the coloring logic with reachability in the object graph, (ii) next, we need to ensure the algebraic properties related to well-formedness and preservation of the object graph for the bit-wise manipulations performed during the GC, (iii) and finally, while the above specifications talk about a functional heap, the C implementation performs in-place mutations, and hence we need to reason about aliasing and memory safety. This naturally points to the need for a layered strategy for verification, which is the focus of the next section.

```
(* dfs calls dfs_body until stack empty.
   Inputs are graph, stack and visited set.  *)
let rec dfs (g:graph) (st:seq U64.t) (vl:seq U64.t)
 : Pure (seq U64.t)
   (requires ...)
   (ensures (λ res → ...)
   (decreases (length g.vertices - length vl; length st)) =
   if length st = 0 then vl
   else
     let st₁ ,vl₁ = dfs_body g st vl in
     dfs g st₁ vl₁

let dfs_body g st vl
 : Pure ...
   (requires ...)
   (ensures (λ res → ...) =
      let x   = stack_top st in
      let xs  = stack_rest st in
      let s   = successors g x in
      let vl₁ = set_insert x vl in
      let st₁ = push_unvisited s xs vl1 in
      (st₁, vl₁)
```

Fig. 19 Functional dfs

6 Verification Framework and Correctness Proofs

With the implementation and the specifications in place, we now deep dive into the details of our verification framework and how we have carried out the main correctness proofs. As mentioned earlier, the challenge here is that we need to reason about complex graph theoretic specifications for an optimized and efficient implementation, while ensuring memory-safe behavior of the GC implementation itself. We have designed our layered verification methodology to cleanly separate various proof obligations involving the graph reachability based specification, the correctness of bitwise arithmetic and the correctness of concrete memory changes carried out by the GC. As shown in Fig. 18, the first layer deals with the verification of reachability properties of DFS, the second layer is for proving algebraic properties related to the bitwise arithmetic operations as well as to prove the abstract graph related properties of the GC, and the third layer proves that the GC does not violate memory safety. The third layer also acts as the layer from which the actual C code of the GC is extracted.

6.1 Layer 1—A Verified Depth First Search Implementation in F*

We know that mark performs a depth-first traversal of the OCaml heap, but it takes advantage of the OCaml object layout to efficiently perform operations such as finding successors, maintaining the visited vertices, etc. Directly proving the mark_reachability_lemma would be hard, especially for F*, as inductively defined properties such as reach do not work well with SMT-based verifiers. Mixing this reasoning with the OCaml object layout and the

```
(* mark calls mark body until stack empty.
   Inputs are heap and stack *)
let rec mark (h:heap) (st:seq obj_addr)
 : Pure (heap)
  (requires ...)
  (ensures (λ res → ...)
  (decreases (length allocs h - length blacks h;
              length st)) =
  if length st = 0 then h
  else
    let st₁, h₁ = mark_body h st in
    mark h₁ st₁

let mark_body (h:heap) (st:seq obj_addr)
  : Pure ...
    (requires ...)
    (ensures (λ res → ...) =
  let x   = stack_top st in
  let xs  = stack_rest st in
  let h₁  = colorHeader h x black in
  let st1 = darken h₁ xs x 1UL in
  (st₁, h₁)
```

Fig. 20 `mark` implementation

bit-wise arithmetic operations occurring in mark would make the problem even harder. To simplify this proof, we instead focus on proving the reachability properties for a bare bones dfs implementation. In the second layer, we establish the functional equivalence between mark and dfs by proving that mark colors all and only those objects reached by dfs.

The dfs implementation, shown in Fig. 19, directly takes as input the object graph (whose type defined in Sect. 5). It explicitly maintains a *visited list*, corresponding to the set of vertices which have been fully explored. We design the dfs implementation to closely resemble the mark implementation, shown in Fig. 20. The dfs implementation is functional and recursive, and in each recursive call, it removes the vertex at the top of the stack, pushes it into the visited list, and then pushes all the unvisited successors into the stack. At the end, dfs returns the set of all vertices reachable from the root set.

The correctness specification of dfs is defined using the reach predicate in Fig. 21. Let us first focus on the ensures clause, which is the required guarantee that we need in terms of reachability. The forward direction says that every vertex present in the return value of dfs must be reachable from some vertex in the root set. The backward direction says that if a vertex is reachable from the root set, then it must be present in the return value of dfs.

To prove the forward direction, we assert the pre-condition (F2,F3) that all vertices in both the stack and the visited list are always reachable from some vertex in the root set. For the backward direction, our pre-condition B1 asserts that every vertex reachable from the root set should either already be part of the visited list, or it should be reachable from some vertex in the stack. However, this property by itself is not inductive, and hence we also require that every vertex reachable from the visited list should either already be part of the visited list, or

```
(* r_list is the root set, stack is filled with r_list initially *)
val dfs_reachability_lemma (g:graph) (st:seq obj_addr)
                           (vl:seq obj_addr) (r_list:seq obj_addr)
  : Lemma
   (requires
     (* Pre-conditions required to prove forward direction *)
     (*F1*) mutually_exclusive_sets st vl ∧
     (*F2*) (∀ y.y ∈ st ⟹ (∃ x.x ∈ r_list ∧ reach g x y) ∧
     (*F3*) (∀ y.y ∈ vl  ⟹ (∃ x.x ∈ r_list ∧ reach g x y) ∧

     (* Pre-conditions required to show the backward direction *)
     (*B1*) (∀ x y.x ∈ r_list ∧ reach g x y  ⟹
            (∃ z.z ∈ st ∧ reach g z y) ∨ y ∈ vl)
     (*B2*) (∀ x y.x ∈ vl ∧ reach g x y  ⟹
            (∃ z.z ∈ st ∧ reach g z y) ∨ y ∈ vl)

   (ensures (∀ y.y ∈ (dfs g st vl) ⟺ (∃ x.x ∈ r_list ∧ reach g x y))
```

Fig. 21 Correctness specification of dfs

reachable from some vertex in the set (B2). We show that these pre-conditions are ensured by the initial call to dfs, and they are also maintained for every recursive call.

6.2 Layer 2—Functional Mark and Sweep in F*

We now consider proving the correctness of the mark and sweep implementations. Towards this end, we first consider their functional implementations in F*, which operate on the OCaml heap representation. Here, the input h will be a sequence of memory words (using the seq type in F*), that contains OCaml objects following the format of Fig. 1. Since the mark and sweep implementations use various *colors* to indicate different phases of an object, these functions depend heavily on a correct implementation of the colorHeader function, which sets the color of an object. The next section focuses on the specifications of colorHeader, which is crucial to ensure the correctness of the GC.

6.2.1 Specification of colorHeader

The specifications of colorHeader are shown in Fig. 22. colorHeader takes as input the heap h, an address hdr into h (which will be the address of the object header) and a color c that has to be updated in the color bits of the value stored at hdr. The function returns the modified heap. makeHeader is a bit manipulation function that is used to create a header value. The operations U64.shift_right and U64.logand are 64-bit right shift and logical AND operations available as part of the F* standard library. The specifications ensure that, only the color bits changes when colorHeader is applied. Especially the predicate heap_same_except ensures that except hdr in h, everything else remains the same in the resultant heap res. Since hdr is a valid_hdr, this ensures that, all the fields of all objects remains the same. Using the specifications of colorHeader, we show the proof outlines for proving the algebraic properties of the GC functions as shown in Fig. 15.

```
  val makeHeader (wz:wosize_t) (c:color_t) (t:tag_t)
      : Pure (U64.t)
        (requires True)
        (ensures (λ res →
                  (U64.shift_right res 10UL = wz) ∧
                  (U64.logand (U64.shift_right res 8) 3UL = c) ∧
                  (U64.logand res 255UL = t)))

   val colorHeader (h:heap) (hdr:hdr_addr) (c:color_t)
      : Pure (heap)
        (requires (* Only core conditions shown *)
                   well_formed_heap (h) ∧
                   valid_hdr hdr h)
        (ensures λ res →
                  well_formed_heap (res) ∧
                  valid_hdr hdr res ∧
                  objs h = objs res ∧
                  heap_same_except hdr h res ∧
                  color hdr res  = c ∧
                  wosize hdr res = wosize hdr h ∧
                  tag hdr res = tag hdr h ∧
                  r_word res hdr =
                   makeHeader (wosize hdr h) (c) (tag hdr h))
```

Fig. 22 Specifications of `colorHeader`

6.2.2 Proof Outline for the Sub-functions of the GC

The specifications are shown in Fig. 15. The darken_roots function pushes all the root pointers in h_list to an empty stack and then colors them as grey. Let h_0 be the initial heap before the GC and let h_1 be the heap resulted after darken_roots. Since the GC starts with well_formed_heap h_0 that contains only white and blue objects and the fact that h_list contents are members of allocs(h_0) implies that well_formed_heap h_1 is preserved as the only change to heap is coloring of h_list members from white to grey. Since both white and grey are considered as allocated objects, changing the color from white to grey still preserves the membership in allocated set. Therefore, allocs(h_1) = allocs(h_0). Therefore the vertex set of both graphs constructed from h_0 and h_1 remains the same. Since the specification of colorHeader ensures that except the color bits of the address hdr in the heap, nothing else in the heap changes, the edge set of the graphs from h_0 and h_1 also remains the same. Therefore, both the graphs are the same. Since darken_roots starts with an empty stack, when the objects in h_list are pushed into the stack and colored them grey in the heap, the only grey objects in the heap are the ones that are pushed onto the stack.

Let h_2 be the heap formed after `mark`. All the algebraic properties of mark as in Fig. 15 can be proved by using the same approach as darken_roots and the fact that mark starts with a stack that contains all the grey and only the grey objects of the heap. When mark terminates after the stack becomes empty, there are no grey objects in the resultant heap after mark.

```
val dfs_mark_equivalence_lemma (h:heap) (st:seq obj_addr)
                               (vl:seq obj_addr) (h_list:seq obj_addr)
      : Lemma
        (* Only important properties shown *)
        (requires (*1*) mutually_exclusive_sets st vl ∧
                  (*2*) well_formed_heap(h) ∧
                  (* stack invariant *)
                  (*3*) (∀ x.x ∈ st  ⟺  (hd_address x) ∈ greys(h))
                  (* visited-list invariant *)
                  (*4*) (∀ x.x ∈ vl  ⟺  (hd_address x) ∈ blacks(h))

        (ensures (∀ x. x ∈ (dfs (graph_from_heap h) st vl) ⟺
                           (hd_address x) ∈ blacks(mark h st)))
```

Fig. 23 Behavioral equivalence between `mark` and `dfs`

Similarly the algebraic properties of sweep can also be directly proven using the specifications of colorHeader. Recall that, sweep changes the color of white objects to blue and black objects to white. As well as the first field of the free list pointer is changed to point to the newly created blue block during a sweep invocation. Since the well-formedness property only affects the allocated objects and the fact that free list pointer points to a blue object, with the help of some extra lemmas related to the sweep properties, F* can prove that the heap resulted after sweep remains as well-formed.

6.2.3 Proof Outline for `mark_reachability_lemma`

As the direct proof of reachability of mark requires reasoning about bit-wise arithmetic and graph reachability together, we have avoided that path due to the inherent complexities to handle such proofs with an SMT solver. Instead, we use the reachability proofs of dfs for mark by proving the program equivalence between dfs and mark using the lemma in Fig. 23.

Proof outline for `dfs_mark_equivalence_lemma`: Both dfs (in Fig. 19) and mark (in Fig. 20) are tail-recursive functions, whose outputs are exactly same as the outputs of the recursive calls at the end. Hence, we use the post-condition of dfs_mark_equivalence_lemma as a form of inductive invariant, following the classical modular verification technique for recursive functions. We also require two invariants to be obeyed relating the input arguments to dfs and mark. The stack invariant says that the grey objects of the input heap h can only be found in the stack st and the visited-list invariant ensures that the black objects of the heap are only present in the visited list vl. This way membership checks in st and vl can be avoided by replacing it with two color bits check, which is more efficient than the membership checks in dfs. We show that if the mark and dfs methods are called with the same input stack st which satisfies the stack invariant, and a visited set that satisfies visited-set invariant as well as a heap that satisfies well_formed_heap (h), then their outputs should be equivalent, i.e., all objects in the output of dfs should be colored *black* in the output heap of mark.

Due to the tail-recursive nature of both dfs and mark, we can use the equivalence of outputs of the recursive calls to infer the required result. Suppose h_1 and st_1 are the results of one invocation of mark_body and st_2 and vl_2 are the outputs obtained after dfs_body. These outputs are passed as inputs to the recursive calls of mark and dfs. Then, we need to ensure that the input arguments to the recursive calls satisfy the pre-conditions mentioned below:

- well_formed_heap h_1

```
(* push_unvisited calls push_unvisited_body until successors is empty *)
let push_unvisited_body s st vl j
  : Pure ...
    (requires j < length s...)
    (ensures (λ res → ...) =
  if (not (mem (index s j) (set_union st vl))) then
    let st₂ = push (head s) st in
    st'

(* darken calls darken_body until i = wosize + 1 *)
let darken_body h st h_addr i
   : Pure ...
    (requires ...)
    (ensures (λ res → ...) =
  let succ = r_word h (h_addr + i * mword) in
  if not (isPointer succ) then (st, h)
  else
   let c = color h (hd_address succ) in
   if not (c = white) then (st, h)
   else
     let h₁, st₁ = push_to_stack h st succ in
     (st₁, h₁)
```

Fig. 24 A comparison of push_unvisited_body and darken_body functions

- $(\forall x.\, x \in vl_2 \iff (\text{hd\_address } x) \in \text{blacks}(h_1))$
- $(\forall x.\, x \in st_1 \iff (\text{hd\_address } x) \in \text{greys}(h_1))$
- $st_2 = st_1$

Since well_formed_heap h holds for the input heap h, the first property follows as the color changes to the heap during one invocation of mark_body only involves darkening. That is, white objects becomes grey and grey objects become black. That means, the allocs h and allocs h_1 remains the same. Combining this property with the fact that the fields remain unchanged due to the coloring operation, the well-formedness of h_1 can be proved. This also ensures that the graphs formed from both h and h_1 remains the same. A careful inspection of dfs_body and mark_body reveals that, each of the functions removes the top of the stack and mark_body colors it as black whereas dfs_body adds it to the visited list. Since the input stacks are the same, this operation ensures that the visited-list invariant is maintained with vl_2 and h_1. While pushing the field pointers of the top of the stack, mark_body colors them grey, which ensures the stack invariant of st_1 with respect to h_1.

For proving the last property, which is the stack equivalence between the stacks produced by dfs_body and mark_body, let us understand the behavior of the two tail recursive functions push_unvisited and darken, used in dfs_body and mark_body to obtain st_2 and st_1 respectively. A comparison between these functions are shown in Fig. 24.

The push_unvisited function scans through the list of successors of x (which was at the top of the stack) in the object graph, while darken scans the fields of the object x in the heap. Both functions starts with the same stack st and populate the stack with unvisited (white) field pointers of x into the stack. push_unvisited and darken perform the bulk of the work by calling push_unvisited_body and darken_body respectively.

The parameter j in push_unvisited_body indicates the index of the successor in s to be examined. Similarly, the parameter i in darken_body indicates the field number of x to be scanned in h. Recall that st and vl are mutually exclusive. The function push_unvisited_body decides whether an element is unvisited by checking the membership in st and in vl. Due to the invariants on st and vl, such an object will also be colored white in the heap. The same action is being performed by the darken_body as well. But the difficulty here is that s is already a filtered list of field pointers (i.e. successors), while the field scan in fields by darken may encounter non-pointer fields as well. Hence, there may not exist a direct one-to-one correspondence between the invocations of push_unvisited_body and darken_body. To get around this issue, we can make use of the observation that if the field_slice which starts at the i^{th} field of x in the heap returns the same set of field pointers as that of successor_slice that starts at j^{th} index of the successors list in s, then the stacks produced by darken that starts at field index i of x in h and push_unvisited that starts at index j of s are the same. Formally in F* we prove the below lemma,

```
val darken_push_unvisited_produces_same_stack (h:heap) (st:seq obj_addr)
                                              (vl:seq obj_addr)
                                              (curr:hdr_addr)
                                              (i:U64.t) (j:U64.t)
   : Lemma
    (requires mutually_exclusive_sets st vl ∧
              well_formed_heap (h) ∧
              successor_slice s j = field_slice h hdr_addr i)
    (ensures  darken h st hdr_addr i = push_unvisited s st vl j)
```

The above lemma is invoked with i = 1 and j = 0 to prove the stack equivalence (i.e., $st' = st_1$). Thus, through the maintenance of pre-conditions, the induction hypothesis through the recursive invocation of the dfs_mark_equivalence_lemma, we are able to complete the proof of dfs_mark_equivalence_lemma. This way, dfs_reachability_lemma and dfs_mark_equivalence_lemma is sufficient to complete the proof of mark_reachability_lemma in Fig. 17.

6.2.4 Proof Outline for `sweep_subgraph_lemma`

Let h_0 be the heap state before the GC, let h be the heap after mark that satisfies dfs_mark_equivalence_lemma and let h_1 be the heap after sweep, and thus the heap after the GC. Therefore, from the algebraic properties of sweep in Fig. 15, if h_list is the root set, curr_ptr is the sweep head and fp is the free list pointer then the following property holds:

```
(∀ x y.y ∈ h_list ∧ reach (graph_from_heap h) y x  ⟺
                 (hd_address x) ∈ whites (sweep h curr_ptr fp))
```

Since the graph before and after darken_roots and mark remains the same, this means all the white objects that results after the sweep are the only reachable objects in h_1. Again from the algebraic properties of sweep, the only allocated objects after the sweep are white objects. We have already shown that coloring operation and modification of the first field of free list does not alter the contents of any allocated object fields. These properties ensure that the objects in the graph formed from h_1 contains all the reachable and only the reachable objects in h_0 and the edges between them remains the same in h_1 as that in h_0. This completes the proof of sweep_subgraph_lemma. Thus we can see that all the abstract GC correctness properties as mentioned in Definition 5 is fulfilled by our functional GC. Now, all it remains is to show that an imperative implementation of such a GC does not violate any of the functional correctness properties of the GC. For that, in the next section, we show how to implement the imperative GC in the third and the final layer, and how to prove its program equivalence with that of the functional GC.

6.3 Layer 3—Imperative Mark and Sweep in Low*

As discussed earlier, the verification focus of this layer is to prove that the imperative GC implementation itself does not cause any memory safety bugs. There are a number of challenges to be dealt with in this layer: the heap and stack are now modeled as fixed-length buffers, thus requiring proofs of absence of buffer overflows, the heap/stack mutations are now in-place instead of functional, thus requiring anti-aliasing proofs. This layer uses Low*, which allows extraction to verified C code. To understand how Low* ensures memory safety of the C implementations, let us first see the code listing of mark in Low* as an example as shown in Fig. 25 and its pre- and post-conditions in Fig. 26. To differentiate from a functional implementation, imperative mark is qualified with a suffix imp. The function takes as input a buffer hp to store the heap, another buffer st to store the stack and tp to store the top pointer of st.

Each of the Low* functions takes a pre-condition (requires clause) on the initial memory state m and a post-condition (ensures clause) about the initial and the final memory states m_0 and m_1 respectively. For example, the mark_imp requires the pre and post conditions about the memory state as shown in Fig. 26.[1]

The term live m hp states that hp is a live buffer in memory state m, where the location is specified by loc_buffer hp. The condition that the buffers hp and st should be disjoint in memory is captured in disjoint (loc_hp) (loc_st). In the post-condition, a modifies clause is used which ensures that the buffers hp, st and tp got modified between the memory states m_0 and m_1. The as_seq function takes as input a memory state m and a buffer and creates the functional seq equivalent of the buffer in m. We need to reason about the stack contents upto stack top only. Therefore, we take a slice or portion of the stack from the start of the stack upto the stack top. This is captured in (slice seq_st_0 0 (index (seq_tp_0) 0). The final clause ensures that the functional equivalent of buffer hp in the final memory state is equivalent to running a functional `mark` with the functional equivalent of hp in the initial memory m_0 and the slice of the functional equivalent of st upto tp in m_0. This clause ensures the output equivalence of functional mark and imperative mark. This way the Low* specification ensures the memory safety of the GC implementation as well as the functional equivalence with the functional implementation. The inv and body functions are used to specify the loop invariants and the body of the loop respectively.

But there is one more hurdle, because of the size limitations of concrete buffers. The allocated stack has a fixed-size. Therefore, there is a probability that the stack might overflow during mark. Low* rightfully captures this caveat and fails to typecheck if no conditions are provided that prevents stack overflow. Hence, to work around this, we set the stack size equals to the heap size and prove a lemma that states that when there is a non-grey object in the heap, the stack top is less than the heap size. The maximum size required to store all the objects in the heap is heap size in the worst case. Since the stack preserves the stack invariant, existence of one non-grey object means the stack top is less than the heap size, and thus less than the stack size. Thus, there is room in the stack to store this non-grey object, which will be converted to grey once it enters the stack.

Now, let us see, how we can establish the functional equivalence between the functional GC, where all algebraic GC properties and the equivalence with a dfs traversal is proved, and the imperative GC. In Low*, we need to prove the program equivalence between the F* and Low* GC intrinsically, that is along with the implementation of the function. The specification is shown in Fig. 27. As explained earlier, hp, st, tp are buffers representing

[1] Some details have been elided. The complete specification can be found in the supplemental material.

Fig. 25 A Low* implementation of mark

```
let mark_imp hp st tp
 :Stack unit
 (requires λ m → ...)
 (ensures λ m₀ _ m₁ →  ...) =
  let inv m = ...(*loop invariants*)
  let guard (t: bool) m = inv m ∧
   (t = true  ⟹ B.get m tp 0) > 0) ∧
   (t = false ⟹ B.get m tp 0) = 0) in
  let test ()
   :Stack bool
    (requires λ m → inv m)
    (ensures λ _ ret m₁ → guard ret m₁)
     = (!*tp) >^ 0UL in
  let body ()
   : Stack unit
    (requires λ m → guard true m))
    (ensures λ _ _ m₁ → inv m₁)
     = mark_heap_body_imp hp st tp in
  C.Loops.while #(inv) #(guard) test body
```

```
requires λ m →
  let loc_hp = loc_buffer hp in
  let loc_st = loc_buffer st in
  let loc_tp = loc_buffer tp in
  live m hp ∧ live m st ∧ live m tp ∧
  disjoint (loc_hp) (loc_st) ∧ disjoint (loc_st) (loc_tp) ∧
  disjoint (loc_hp) (loc_tp)

ensures λ m₀ _ m₁ →
  let union = loc_union (loc_buffer hp)
                (loc_union (loc_buffer st) (loc_buffer tp))
  in
  let seq_st₀ = as_seq m₀ st in
  let seq_hp₀ = as_seq m₀ hp in
  let seq_tp₀ = as_seq m₀ tp in
  let seq_hp₁ = as_seq m₁ hp in
  let slice_st₀ = slice seq_st₀ 0 (index (seq_tp₀) 0) in
  live m₁ hp ∧ live m₁ st ∧ live m₁ tp ∧
  (* Same disjoint clause as above *) ∧
  (modifies union m₀ m₁) ∧
  seq_hp₁  = mark seq_hp₀ slice_st₀
```

Fig. 26 Pre- and post-conditions of the Low* `mark` implementation in Fig. 25

```
val mark_sweep_gc_imp hp st tp rlist rlist_len sw fp
 :Stack unit
 (* Similar conditions for mark...omitted *)
 (requires λ m → ...)
 (ensures λ m₀ _ m₁ →
   let seq_st₀ = as_seq m₀ st in
   let seq_hp₀ = as_seq m₀ hp in
   let seq_r_list = as_seq m₀ r_list in
   let seq_tp₀ = as_seq m₀ tp in
   let seq_rlist_len₀ = as_seq m₀ rlist_len in
   let seq_hp₁ = as_seq m₁ hp in
   let slice_rlist₀ = slice seq_r_list₀ 0 (index (seq_rlist_len₀) 0) in
   let slice_st₀ = slice seq_st₀ 0 (index (seq_tp₀) 0) in
     seq_hp₁ = mark_sweep_gc seq_hp₀ slice_st₀ slice_rlist₀ sw fp)
```

Fig. 27 Pre- and post-conditions of the Low* `mark_sweep_gc` implementation

the heap, stack and the stack pointer respectively. Similarly, rlist, rlist_len, sw and fp are all buffers that carries roots, the last location of rlist up to which the roots are stored, the sweep pointer and the free-list pointer respectively. This specification establishes the functional correctness of the GC implementation with that of the algebraic properties of the functional GC implementation. Note that, the functional GC implementation acts as the middle layer of specifications, which aids in the final verification of abstract GC correctness defined in Definition 5.

How to prove the functional equivalence between the imperative and the functional GC? Here, we need to prove the output equivalence of each of the sub-functions that make up the imperative GC with their functional counter parts. For functions without loops such as darken_body and colorHeader, the equivalence proof is straightforward, as the operations in these functions are almost identical in both functional and imperative world, with the only difference being that of the underlying data structure (sequences as opposed to buffers). Functions with loops require a suitable inductive loop invariant. However, since the functional mark and sweep implementation was designed to have only tail-recursive functions, which correspond to a tight while-loop, the loop invariants establishing equivalence are quite straightforward. They essentially capture equivalence between an iteration of the loop in the imperative implementation and an invocation of the tail-recursive method in the functional implementation.

6.4 End-to-end GC Correctness

Our end-to-end correctness condition is captured in Fig. 28. The end-to-end correctness theorem is written using Layer 2 primitives. The specification takes in as arguments a well-formed heap h_init, a root set roots, a mark stack st that contains all the roots with the refinement that all the roots are grey, and a free-list pointer fp. We elide additional pre-conditions in the requires clause that relate the arguments to the heap.

The ensures clause captures the post-conditions after the execution of mark and sweep. The ensures clause is a conjunction of five properties. The first property states that the final heap after sweep h_sweep is well-formed. This corresponds to the first property in abstract GC correctness definition (Definition 5). The second property states that graph formed out of the final heap h_sweep will only have those objects that are reachable from the roots in the

```
val end_to_end_correctenss_theorem
        (* Initial heap *)
        (h_init:heap{well_formed_heap h_init})
        (* mark stack - contains grey objects *)
        (st: seq Usize.t {pre_conditions_on_stack h_init st })
        (* root set *)
        (roots : seq Usize.t{pre_conditions_on_root_set h_init roots})
        (* free list pointer *)
        (fp:hp_addr{pre_conditions_on_free_list h_init fp})
: Lemma
( requires
  (* Pre-conditions elided for brevity. Important ones are:
     + The mark stack [st] contains all the [roots].
     + All the grey objects in the heap are in the mark stack [st].
   *) )
( ensures
   (* heap after mark *)
   let h_mark = mark h_init st in
   (* heap after sweep *)
   let h_sweep = fst (sweep h_mark mword fp) in
   (* graph at init *)
   let g_init = graph_from_heap h_init in
   (* graph after sweep *)
   let g_sweep = graph_from_heap h_sweep in
   (* GC preserves well-formedness of the heap *)
   (* 1 *) well_formed_heap h_sweep ∧

   (* GC preserves reachable object set *)
   (* 2 *) ((∀ x. x ∈ g_sweep.vertices ⟺
             (∃ o. mem o roots ∧ reach g_init o x))) ∧

   (* GC preserves pointers between objects *)
   (* 3 *) ((∀ x. mem x (g_sweep.vertices) ⟹
             (successors g_init x) ==
             (successors g_sweep x))) ∧

   (* The resultant heap objects are either white or blue only *)
   (* 4 *) (∀ x. mem x (h_objs h_sweep) ⟹
             color x h_sweep == white ∨
             color x h_sweep == blue) ∧

   (* No object field (either pointer or immediate) is modified *)
   (* 5 *) field_reads_equal h_init h_sweep )
```

Fig. 28 Overall correctness theorem for the mark and sweep GC

initial heap, and every reachable object in the initial heap is present in the final graph. The third property states that the edges between the reachable objects are preserved by the GC. The second and the third properties together correspond to the second property in Definition 5. The fourth property states that the resultant heap only has white and blue objects. The abstract GC correctness does not refer to object colours as the notion of GC colour is an implementation detail of the GC. However, it is an important implementation detail that ensures that the GC leaves the heap in a consistent state for the next cycle. Finally, the fifth property states that fields (both immediate and pointer) of the non-blue objects are unmodified by the GC. This property is stronger than the third property in Definition 5, which only says that the data fields remain the same. Since the mark and sweep GC does not move objects, the pointer fields are also preserved.

One might wonder why the end to end correctness theorem is defined in layer 2 (F*) and not in layer 3 (Low*). As described in Sect. 6.3, our approach is to have the proofs of the GC correctness in Layer 2 and prove the equivalence of the imperative GC with the functional GC in Sect. 3 only focussing on the memory safety properties.

6.5 Extending the GC Functionality

In all our previous discussions as mentioned in Sect. 4, we have used the base version of mark function. But as we change the GC variant to deal with different types of OCaml objects (i.e., closure and infix objects), both the object graph construction, and the scanning of fields performed by mark needs to change in sync with each other. The graph construction acts as the bridge between the abstract graph world and the functional GC world and hence the graph construction should be carefully done to connect the two worlds together.

In the case of the second version of mark (Fig. 9), the edge set of an object with No_scan_tag is made empty. For closure objects, the edge set is constructed by scanning the fields that are stored from the start of the environment (See Sect. 2.1). During the edge set construction of an object, if the field point to an infix object, then the parent closure of that infix object is added as the successor, instead of the infix object. Also the definitions of well-formedness have to capture the property that the return sequence of h_objs should never have an infix object, as the infix object pointers are *interior* pointers. There are additional properties related to the closure info field of closure objects such as the minimum number of fields for a closure object should be at least 2 (See Sect. 2), which are also part of the well-formedness property. By adapting the graph construction, and with the help of additional lemmas, we have verified the third version of the mark function as mentioned in Sect. 4 (Fig. 10). We have also proved that the GC with the third variant of mark follows all the correctness properties as described in Fig. 28.

7 Evaluation

In this section, we report on the verification effort of building a correct-by-construction GC for OCaml, and evaluate the performance of the verified GC on a variety of benchmarks.

7.1 Verification Effort

The verification effort is summarized in Table 2. We calculate the effort in terms of different metrics such as the lines of code, the number of definitions and lemmas, the time required by

Table 2 Verification effort

Modules	#Lines	#Defns	#Lemmas	Verif time	Dev effort
Graph	4653	72	81	2 m 3 s	3
DFS	657	1	9	2 m 5 s	9
Functional GC	18,401	65	218	120 m	12
Imperative GC	2734	19	19	27 m 43 s	3

Development effort (Dev effort) is in person-months

F*/Low* to discharge the VCs, and the human development time. The development time is in terms of number of person-months. We note that the F*/Low* code/proofs were developed by a PhD student who was new to verification and F*/Low*. We divide the verification effort across the different layers of our approach. Since we co-develop programs and proofs, the total number of lines of code include both the programs and proofs together.

The Graph module contains the mathematical graph and reachability definitions, and several functions and lemmas on paths in the graph. These are needed to implement and prove the correctness of the DFS module. Proving the reachability property of DFS (Fig. 21) was particularly tricky as we needed to discover complex inductive invariants involving the `reach` predicate.

The Functional GC module incorporates multiple proofs that assert the correctness of the functional mark and sweep implementation, alongside graph construction and demonstrations of equivalence between the `mark` and `dfs` functions. This requires development of inductive invariants to show equivalence between corresponding methods of DFS and functional mark, as well as proving algebraic properties of the various bit manipulation operations performed by the GC. Once the functional GC was developed, implementing and verifying the imperative GC in Low* is more straightforward. The various verification tasks associated with the imperative GC module include establishing suitable loop invariants to prove equivalence between the functional and imperative GC, proving various memory safety properties such as lack of aliasing, ensuring allocation of memory before access, no use-after-free bugs, etc.

Throughout the layers, we also adopted an incremental approach in adding complex GC features, such as closures and infix objects. Initially, we proved the GC correctness over a basic version that does not distinguish between different types of OCaml objects. Subsequently, we introduced modifications in both the implementations and proofs to accommodate more complex GC variants.

Note that our trusted code base now includes F*/Low* and the KaRaMel compiler which translates Low* to C. It has to be noted that these tools also serve as the trusted code bases for previous projects such as Everest [1] and Everparse [29], which focus on verifying cryptographic implementations and parsers.

The verification through F*/Low* was challenging due to several reasons. F* uses Z3 SMT solver. Our verification conditions (VCs) are not restricted to decidable logics. While Z3 does best-effort reasoning, it may take a long time to prove some VCs or the proof does not terminate. We had to tune the timeouts for individual lemmas to get them to discharge. In addition, Z3 is also non-deterministic. The same VC may be discharged in one run and not in another depending upon the solver state. F* provides some support for dealing with proof instability, such as running in `quake` mode or using `proof-recovery` mode to recover from proof failures. However, the fundamental issue of non-determinism remains.

Table 3 Benchmark characteristics

Benchmark	OCaml 4.14.1					Verified GC	
	Alloc	Promote	# Minor	# Major	MaxRSS	GCs	MaxRSS
BinaryTrees	15,206	7900	7647	87	516	42	515
CountChange	905	140	458	11	145	5	260
FannkuchRedux	0.03	0	0	0	2.68	0	3.5
Fasta	3171	0.03	1569	4	44	72	67
Quicksort	19	0.02	1	0	22	0	22
Nbodies	808	0.04	405	2	4.66	3819	3.63
Mandelbrot	3009	0.13	1508	9	4.3	34,201	3.65
Spectralnorm	3052	0.06	1529	8	4.7	47	67
Knucleotide	140	17	52	6	57	2	67
Cpdf	512	200	254	11	140	1	517
Yojson	129	14	45	16	17	48	14

Alloc, Promote and maxRSS are in MB

7.2 Integration with OCaml

Our goal in this work is to develop a practical verified GC for OCaml that can serve as a replacement for the unverified GC. We have successfully extracted the verified C code for the GC functionality from Low* using the KaRaMel compiler [28]. We have integrated the extracted code into the OCaml 4.14.1 runtime system, replacing the existing GC in the bytecode runtime.

Unlike OCaml 4.14.1 GC, our verified GC is stop-the-world and non-generational. We use an unverified next-fit allocator written in Rust that allocates objects in the verified heap. As mentioned before, our heap is a single, contiguous block of memory (encoded as an F* buffer), into which the objects are allocated. The verification of the allocator is orthogonal to the focus of the work. There is a recent work on StarMalloc [30] which provides a verified, hardened memory allocator written in F*/Low*. We plan to investigate integrating StarMalloc with our verified GC in the future. When the heap is full, the verified GC is triggered. We use the existing root marking procedure in the OCaml runtime to darken the roots and push them to the verified mark stack. This is followed by the call to the verified mark and sweep function.

We made the following small modifications to the extracted code to facilitate integration with the OCaml runtime. The first modification is in the sweep code, where we have implemented coalescing of consecutive free blocks. This is done to reduce fragmentation. The second modification is necessitated by the fact that infix objects do not appear in the mark stack in the verified GC, whereas they do (during root marking) in the OCaml runtime. Since the root marking is done by the OCaml runtime, we have added a wrapper function that inserts the parent closure of an infix object into the mark stack if an infix object appears as a GC root.

7.3 GC Evaluation

We evaluate the performance of the verified GC on a variety of benchmark programs from the Computer Language Benchmarks Game [13] as well as larger programs—cpdf (an

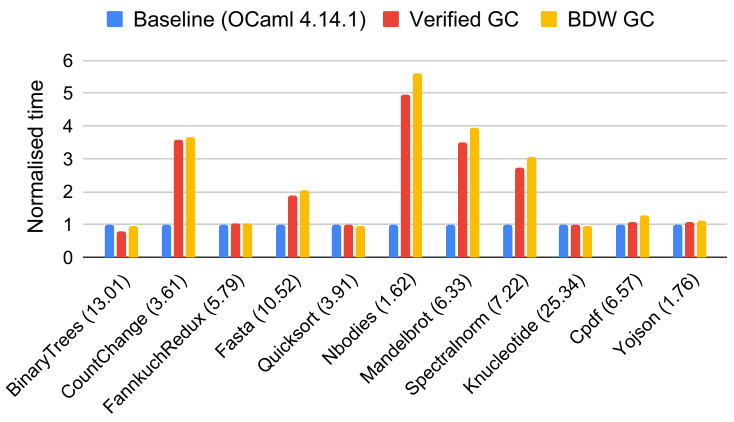

Fig. 29 Normalized running time of different OCaml GCs. The numbers in the parenthesis next to the benchmark names are the running time in seconds for the baseline OCaml 4.14.1 GC

industrial-strength pdf processing tool) and yojson (JSON processing library)—from the OCaml ecosystem. The larger programs have a deep dependency graph of other packages from the OCaml ecosystem. The performance evaluation was performed on a 2-socket, Intel® Xeon® Gold 5120 CPU x86-64 server, with 28 physical cores (14 cores on each socket), and 2 hardware threads per core. Each core runs at 2.20GHz and has 32 KB of L1 data cache, 32 KB of L1 instruction cache and 1MB of L2 cache. The cores on a socket share a 19 MB L3 cache. The server has 64GB of main memory and runs Ubuntu 20.04 LTS.

The benchmark characteristics are given in Table 3. Alloc, Promote and maxRSS indicate the allocated memory, memory promoted from minor to major heap and the maximum resident set size in MB. Since the running times for OCaml programs are a function of the heap size, for a fair comparison, we have chosen the heap size of the verified GC such that the maximum resident set sizes in both the cases are similar, except in cases where the verified GC runs out of memory with small heap sizes. The exceptional case occurs since the verified GC can waste space due to fragmentation. From the table, we can see that the verified GC is able to run fairly large programs using similar maxRSS. Some of the programs also allocate a lot of memory, triggering many GCs.

Fig. 29 shows the running time of the benchmarks run using different GCs normalized against the default OCaml 4.14.1 GC. The comparison also includes OCaml equipped with Boehm–Demers–Weiser (BDW) GC [2]. BDW GC is widely-known, pragmatic GC for uncooperative environments. This means that unlike the other GCs used in the comparison, BDW GC does not have access to precise root set information. It operates in a conservative fashion, and may over-approximate the actual set of accessible objects. On many programs, the verified GC performs on par with the baseline GC and never worse than the BDW GC. On benchmarks where the verified GC and BDW GC are slower, we can attribute the slowdown to the lack of a generational collector. For example, on the Nbodies benchmark, the verified GC is almost 6$\times$ slower than the baseline. We can see in Table 3 that almost none of the memory is promoted to the major heap. Without a generational collector, the verified GC spends a lot of time sweeping garbage, whereas a copying minor collector in the baseline only needs to copy live objects to the major heap. The results show that the verified GC is pragmatic.

8 Extending the Verified GC

In this section, we shows how we can extend our verified GC to collectors that are different from our stop-the-world mark-and-sweep GC. For this exercise, we pick two collectors that are used in the current OCaml runtime system, namely (1) a copying collector and (2) an incremental version of the mark-and-sweep collector. Our aim in this section is not to develop a full-fledged verified versions of these collectors, but rather show that we can model their correctness specifications by extending the specifications and the proofs that we have used in the stop-the-world mark-and-sweep collector. As we will show, we can extend the abstract GC correctness specification from Sect. 3 to cover these collectors as well.

8.1 Incremental Mark and Sweep GC

An incremental mark and sweep GC, as the name suggests performs the GC work in *slices*. During each slice, the GC performs a part of the mark or sweep work and these slices are interleaved with the mutator actions, i.e., the execution of the OCaml program. The main advantage of an incremental GC is that it can reduce the pause times of the application. In a stop-the-world GC, the GC actions are performed in a single shot, which means that the application is paused for the entire duration of the GC. With an incremental GC, the program is paused only for the duration of a slice.

OCaml uses an incremental mark and sweep collector for the major heap. OCaml uses snapshot-at-the-beginning variant [38] of the incremental GC, which ensures that the objects that are reachable at the beginning of the cycle are reachable at the end of the cycle as well. This is achieved by using a deletion write barrier, which, on a field update, marks the old value at that field. As the result, the old value, if unmarked, gets marked before it is overwritten by the new value. This ensures that the old value and the objects transitively reachable from it are reachable at the end of the cycle. As a result, despite the updates to the object graph, all the objects that were reachable at the start of the cycle remain reachable at the end of the cycle (which is the snapshot-at-the-beginning property). Note that the write barrier is the means by which the mutator coordinates with the collector.

How do we reason about the correctness of an incremental mark and sweep GC? We can still reason about the overall correctness of the GC cycle similar to the abstract GC correctness (Definition 5) from Sect. 3. However, we need the mutator to provide us a summary of the changes that it has made to the heap during the GC cycle. Let new_allocs be the set of objects that are allocated by the mutator during a GC cycle. Whenever the mutator allocates a new object, the object id is added to the new_allocs set. Let added_edges and deleted_edges be the set of edges that are added and deleted by the mutator during the GC cycle. Assume that new_allocs, added_edges and deleted_edges are empty at the start of the cycle. The added_edges and deleted_edges are computed in the write barrier. The write barrier is a function that gets called before a write x:= y is performed. The sets added_edges and deleted_edges are computed in the write barrier as follows:

```
(∗ called before [x := y] ∗)
let write_barrier (x,y) = (∗ assuming that [y] is a heap object ∗)
    let old = !x in (∗ assuming that [old] is a heap object ∗)
    added_edges := (added_edges \ {(x,old)}) ∪ {(x,y)};
    deleted_edges := (deleted_edges ∪ {(x,old)}) \ {(x,y)};
    (∗ ...other write barrier actions... ∗)
```

Note that the above write barrier ensures that $\mathsf{added\_edges} \cap \mathsf{deleted\_edges} = \emptyset$. An edge that is added and then deleted will only appear in deleted_edges set. Similarly, an edge that is deleted and then added will only appear in added_edges set.

With this information, we can define the correctness of an incremental mark and sweep GC. Let h_0 be the initial state of the heap on which the GC operates, such that $\omega(\mathsf{h}_0)$ holds, and let r be the set of *roots*, which are pointers to objects into h_0. Let $\mathsf{G}(\mathsf{h}_0).\mathsf{V}$ be the vertex-set and $\mathsf{G}(\mathsf{h}_0).\mathsf{E}$ be the edge-set of $\mathsf{G}(\mathsf{h}_0)$. Let h_1 be the heap after a full cycle of the incremental mark and sweep GC. Let $\mathsf{G}(\mathsf{h}_1).\mathsf{V}$ be the vertex-set and $\mathsf{G}(\mathsf{h}_1).\mathsf{E}$ be the edge-set of $\mathsf{G}(\mathsf{h}_1)$. Let $\mathsf{RG}(\mathsf{h}_0,\mathsf{r})$ be the reachable sub-graph residing in $\mathsf{G}(\mathsf{h}_0)$.

Definition 6 (*GC correctness*) An incremental mark and sweep GC is said to be correct if the following conditions hold:

1. $\omega(\mathsf{h}_1)$
2. (a) $\mathsf{G}(\mathsf{h}_1).\mathsf{V} = \mathsf{RG}(\mathsf{h}_0,\mathsf{r}).\mathsf{V} \cup \mathsf{new\_allocs}$
   (b) $\mathsf{G}(\mathsf{h}_1).\mathsf{E} = (\mathsf{RG}(\mathsf{h}_0,\mathsf{r}).\mathsf{E} \cup \mathsf{added\_edges}) \setminus \mathsf{deleted\_edges}$
3. $(\forall\, \mathsf{x}.\ \mathsf{x} \in \mathsf{G}(\mathsf{h}_1.\mathsf{V}) \implies \mathsf{data}(\mathsf{x},\mathsf{h}_0) = \mathsf{data}(\mathsf{x},\mathsf{h}_1))$

Observe that the correctness specification of the incremental mark and sweep GC is the same as Definition 5, except for the change summary from the mutator.

In addition to the correctness specification, we also observe that the incremental mark and sweep GC can use the correctness proofs of the stop-the-world mark and sweep GC. The intuition is that each slice of the incremental mark and sweep GC is a sequence of atomic mark and sweep steps. The atomic mark and sweep steps are exactly the definitions in mark_body (in Fig. 10) and sweep_body (in Fig. 11), respectively. Given that the new_allocs, added_edges and deleted_edges are not modified during a GC slice, we conjecture that the proofs of mark_body and sweep_body can be reused for the incremental mark and sweep GC without any significant change. As a result, we anticipate that the incremental mark and sweep GC will reuse significant parts of the proofs of the stop-the-world mark and sweep GC.

8.2 Copying Collector

In the context of a copying collector, the heap is divided into two disjoint spaces, namely from_space and to_space. The goal of the copying collector is to copy all the reachable and only the reachable objects from the from_space to the to_space. Earlier works [27] have proved the correctness of a standalone copying collector (albeit with a different object layout than OCaml), and we simply adapt their correctness specifications in our framework.

Let h_0 be the state of from_space of the heap on which the collector operates, such that $\omega(\mathsf{h}_0)$ holds, and let r be the set of *roots*, which are pointers to objects into h_0. Let $\mathsf{G}(\mathsf{h}_0).\mathsf{V}$ be the vertex-set and $\mathsf{G}(\mathsf{h}_0).\mathsf{E}$ be the edge-set of $\mathsf{G}(\mathsf{h}_0)$. Let h_1 be the state of the to_space of the heap after the GC terminates and let $\mathsf{G}(\mathsf{h}_1).\mathsf{V}$ be the vertex-set and $\mathsf{G}(\mathsf{h}_1).\mathsf{E}$ be the edge-set of $\mathsf{G}(\mathsf{h}_1)$. Note that in a copying collector, the to_space is empty at the beginning of the GC.

Let $\mathsf{RG}(\mathsf{h}_0,\mathsf{r})$ be the reachable sub-graph residing in $\mathsf{G}(\mathsf{h}_0)$. Let f be a one-to-one mapping function which maps objects in the from_space to objects in the to_space. For a set s, we define $f_{\mathcal{S}}(s)$ as the set obtained by applying f to every element in s. For a graph g, we define $f_{\mathcal{G}}(g)$ as the graph obtained by applying f to every object in vertex set $g.V$ and to the components of the pair in the edge set $g.E$.

Definition 7 (*Correctness of copying collector*) The copying collector is said to be correct if the following conditions hold:

1. $\omega(\mathsf{h}_1)$
2. $\mathsf{G}(\mathsf{h}_1) = f_{\mathcal{G}}(\mathsf{RG}(\mathsf{h}_0,\mathsf{r}))$, where $\mathsf{G}(\mathsf{h}_1).\mathsf{V} = f_{\mathcal{S}}(\mathsf{RG}(\mathsf{h}_0).\mathsf{V})$ and
   $(\forall x_1, y_1.\, x_1 \in \mathsf{G}(\mathsf{h}_1).\mathsf{V} \wedge y_1 \in \mathsf{G}(\mathsf{h}_1).\mathsf{V} \wedge (x_1, y_1) \in \mathsf{G}(\mathsf{h}_1).\mathsf{E}) \iff$
   $(\exists x_0, y_0.\, x_0 \in \mathsf{G}(\mathsf{h}_0).\mathsf{V} \wedge y_0 \in \mathsf{G}(\mathsf{h}_0).\mathsf{V} \wedge x_1 = f(x_0) \wedge\ y_1 = f(y_0) \wedge$
   $(x_0, y_0) \in \mathsf{G}(\mathsf{h}_0).\mathsf{E})$
3. $(\forall x_1.\, x_1 \in \mathsf{G}(\mathsf{h}_1).\mathsf{V} \iff$
   $(\exists x_0.\, x_0 \in \mathsf{G}(\mathsf{h}_0).\mathsf{V} \wedge x_1 = f(x_0) \wedge \mathsf{data}(x_0,\mathsf{h}_0) = \mathsf{data}(x_1,\mathsf{h}_1)))$

Notice that this definition is almost identical to the correctness specification of the mark-and-sweep GC defined in Sect. 3. In particular, it uses the object reachability predicate to define the reachable subgraph RG in the from_space heap, which needs to be preserved by the heap in the to_space, along with well-formedness of the to_space heap and preserving the data values.

In the context of OCaml 4 runtime system, the copying collector is used for collecting the minor heap. The from_space will be the minor heap, while the to_space would be the major heap. Since we already have a verified major collector, a verified copying collector for the minor heap can be incorporated fairly independently. The only subtlety is that for the minor collection, the pointers from the major heap to the minor heap must be included in the root set of the minor collection. This is ensured by the mutator, which maintains a remembered set of pointers from the major heap to the minor heap. Crucially, this is an expectation on the mutator and not the collector. However, unlike the incremental mark-and-sweep GC, where significant parts of the layer 2 proofs may be reused, we anticipate that the copying collector will require significant re-engineering in layer 2 as the copying collector algorithm is quite different from a mark and sweep GC.

9 Related Work

Previous works on verifying GCs have either used pen-and-paper proofs or mechanization using theorem provers. Mechanized verification has an advantage over pen and paper proofs, so our discussion mainly focuses on mechanically verified GCs. Hawblitzel et al. [16] verified a mark and sweep collector, similar to ours, and a copying collector implemented in x86 assembly. They extensively annotated code with specifications, using Boogie and Z3 to discharge proof obligations. Their verification does not define GC correctness based on object reachability. Instead, the verification relies on object color invariants of the GC implementation specifically tied to the Bartok compiler. In contrast, our GC correctness specifications, based on explicit reachability at an abstract graph theoretic level, are suitable to specify the correctness of diverse GCs. As we discussed in Sect. 8, our specification can be extended and can be used to describe the correctness of a copying collector, which does not use object colors. We believe that our abstract graph-theoretic specification will let us evolve the GC without having to wholesale rewrite the correctness specifications for each revision of the GC.

Gammie et al. [10] verified a concurrent mark-and-sweep collector model in Isabelle/HOL, but over an abstract model rather than the actual code. Our work verifies the GC at both abstract and concrete implementation levels, with the C code extract after the verification integrated with the OCaml run-time. Zakowski et al. [39] used Coq to verify a concurrent mark-and-sweep collector expressed in a compiler intermediate representation, without generating executable code. Xu et al. [37] propose a model checking framework for gaining confidence in collector correctness but do not present a concrete framework. McCreight et al. [24]

provide a framework for verifying GC and mutators, where they have proved the correctness of both mark and sweep and copying collectors written in a RISC-like assembly language. The advantage of our work is that we can extract portable C code from our verified GC, and can support all the platforms that OCaml supports including x86, ARM, Power, RISC-V and IBM s390x.

Another notable prior work is the verification of a generational copying collector for CakeML [6], which employs HOL4. Similar to our work, they also employ a layered approach from abstract algorithmic levels down to assembly closely integrated with CakeML's compiler. Wang et al. [36] develop a mathematical and spatial graph library in Coq, verifying a generational copying collector as part of their framework. They verify the correctness of their copying GC by proving the abstract graph isomorphism established by the copying function. Their basic object representation is similar to ours, where an object consists of a header followed by a variable number of fields. Their 400-line implementation was sufficient to certify a GC for the CertiCoq project. Compared to their object layout support, we need to support additional complexities associated with the OCaml language including no-scan, closure and infix object types.

Lin et al. [20] present the verification of a Yuasa incremental GC in a Hoare-style PCC framework, the Stack-based Certified Assembly Programming (SCAP) system [8] with embedded separation-logic [31] primitives. Their verification in Coq ensures that the collector always preserves the heap objects reachable by the mutator. Some of the specification constructs follow their previous work [21] on verifying a stop-the-world mark-sweep collector. However, their collectors assume that every object has exactly two fields. Our support for different types of OCaml objects with variable-length fields poses additional verification challenge. The extraction to portable C code is a unique feature of our work, not available in any of the prior works.

As part of our development, we have verified the correctness of a DFS algorithm on graphs. Verification of graph algorithms is a well-studied area. Several works also verify complex specifications for graph algorithms. Lammich et al. [19] provide a framework for verifying depth-first search algorithms in Isabelle. Gueneau et al. [14] use a program logic to verify both correctness and complexity of an incremental cycle detection algorithm. Chen et al. [4] verify Tarjan's strongly connected components algorithm using different verification frameworks, encountering challenges with reasoning about reachability over arbitrary-length paths. We believe that the prior work on graph algorithms will pave way for reasoning about the correctness of complex GC algorithms. Our approach of separating out graph-theoretic correctness from the GC implementation will be suitable to integrate such complex graph algorithms into GC verification.

10 Limitations, Conclusion and Future Work

In this work, we have successfully developed a correct-by-construction GC for OCaml in a proof-oriented manner using F*/Low* proof-oriented programming language. We have extracted C code from the Low* program and have integrated the verified GC with the OCaml. The OCaml compiler with the verified GC is able to run standard benchmark programs as well as larger programs from the OCaml ecosystem. The experimental results demonstrate that our verified GC is pragmatic. We believe that our layered verification strategy should enable us to get close to the generational, incremental mark-and-sweep GC used by OCaml.

We have described how our specifications can be extended to cover the correctness of these algorithms.

In our current work, we have the limitation that the size of the mark stack should be equal to the size of the heap (Sect. 6.3). This is necessary to prove the absence of mark stack overflow. In the literature, there are a number of techniques to handle mark stack overflow. For example, one approach on mark stack overflow is to continue marking but not push the objects into the stack. After the mark stack is empty, we linearly scan the heap to identify those objects which are marked but have at least one unmarked child, and mark them. Proving the correctness of this approach is non-trivial, and we would like to explore this approach in the future.

Another limitation of our work is that we do not short-circuit evaluated lazy values. OCaml has support for lazy evaluation through lazy values. A lazy value is represented by an object with the lazy_tag, with one field that holds a reference to the closure that represents the lazy computation. When the lazy computation is forced, the tag of the object is updated to forward_tag, and the result written to the first field. The observation is that the GC can short-circuit the reference to the result, avoiding the intermediate forward_tag object. Short-circuiting lazy values is an optimization and does not affect the correctness of the GC. We would like to explore this optimization in the future.

One of the challenges that we encountered with Low* is the need for explicit anti-aliasing proofs. The proofs are not difficult to write, but they are tedious. F* has support for concurrent separation logic through Steel [9] and its successor Pulse [7], which we believe can not only simplify the proofs, but also allow us to reason about the correctness of concurrent GCs.

References

1. Bhargavan, K., Bond, B., Delignat-Lavaud, A., Fournet, C., Hawblitzel, C., Hritcu, C., Ishtiaq, S., Kohlweiss, M., Leino, R., Lorch, J., et al.: Everest: towards a verified, drop-in replacement of https. In: 2nd Summit on Advances in Programming Languages (SNAPL 2017). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2017)
2. Boehm, H.-J., Weiser, M.: Garbage collection in an uncooperative environment. Softw. Pract. Exp. **18**(9), 807–820 (1988)
3. Burdy, L.: B vs. Coq to prove a garbage collector. In: the 14th International Conference on Theorem Proving in Higher Order Logics: Supplemental Proceedings (2001)
4. Chen, R., Cohen, C., Lévy, J.-J., Merz, S., Théry, L.: Formal proofs of tarjan's strongly connected components algorithm in why3, coq and isabelle. In: ITP 2019—10th International Conference on Interactive Theorem Proving, vol. 141, pp. 13-1. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2019)
5. Coupet-Grimal, S., Nouvet, C.: Formal verification of an incremental garbage collector. J. Logic Comput. **13**(6), 815–833 (2003)
6. Ericsson, A.S., Myreen, M.O., Pohjola, J.Å.: A verified generational garbage collector for CakeML. J. Autom. Reason. **63**(2), 463–488 (2019)
7. F* team: Pulse: Proof-oriented Programming in Concurrent Separation Logic. https://fstar-lang.org/tutorial/book/pulse/pulse.html Accessed 04 Dec 2024

8. Feng, X., Shao, Z., Vaynberg, A., Xiang, S., Ni, Z.: Modular verification of assembly code with stack-based control abstractions. ACM SIGPLAN Not. **41**(6), 401–414 (2006)
9. Fromherz, A., Rastogi, A., Swamy, N., Gibson, S., Martínez, G., Merigoux, D., Ramananandro, T.: Steel: proof-oriented programming in a dependently typed concurrent separation logic. Proc. ACM Program. Lang. **5**(ICFP) (2021). https://doi.org/10.1145/3473590
10. Gammie, P., Hosking, A.L., Engelhardt, K.: Relaxing safely: verified on-the-fly garbage collection for x86 TSO. ACM SIGPLAN Not. **50**(6), 99–109 (2015)
11. Goguen, H., Brooksby, R., Burstall, R.: An abstract formulation of memory management. Technical report, University of Edinburgh, December (1998)
12. Gonthier, G.: Verifying the safety of a practical concurrent garbage collector. In: Computer Aided Verification: 8th International Conference, CAV'96 New Brunswick, NJ, USA, 31 July–3 August 1996 Proceedings 8, pp. 462–465. Springer, Berlin (1996)
13. Gouy, I.: The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/
14. Guéneau, A., Jourdan, J.-H., Charguéraud, A., Pottier, F.: Formal proof and analysis of an incremental cycle detection algorithm. In: Interactive Theorem Proving. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2019)
15. Havelund, K.: Mechanical verification of a garbage collector. In: Parallel and Distributed Processing: 11th IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing San Juan, Puerto Rico, USA, 12–16 April 1999 Proceedings 13, pp. 1258–1283. Springer (1999)
16. Hawblitzel, C., Petrank, E.: Automated verification of practical garbage collectors. ACM SIGPLAN Not. **44**(1), 441–453 (2009)
17. Jackson, P.B.: Verifying a garbage collection algorithm. In: Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs' 98 Canberra, Australia 27 September–1 October 1998, Proceedings 11, pp. 225–244. Springer (1998)
18. Jones, R., Hosking, A., Moss, E.: The Garbage Collection Handbook: The Art of Automatic Memory Management. CRC Press, Boca Raton (2016)
19. Lammich, P., Neumann, R.: A framework for verifying depth-first search algorithms. In: Proceedings of the 2015 Conference on Certified Programs and Proofs, pp. 137–146 (2015)
20. Lin, C., Chen, Y., Hua, B.: Verification of an incremental garbage collector in Hoare-style logic. Int. J. Softw. Informatics **3**(1), 67–88 (2009)
21. Lin, C.-X., Chen, Y.-Y., Li, L., Hua, B.: Garbage collector verification for proof-carrying code. J. Comput. Sci. Technol. **22**(3), 426–437 (2007)
22. Madhavapeddy, A., Minsky, Y.: Real World OCaml: Functional Programming for the Masses. Cambridge University Press, Cambridge (2022)
23. Martínez, G., Ahman, D., Dumitrescu, V., Giannarakis, N., Hawblitzel, C., Hriţcu, C., Narasimhamurthy, M., Paraskevopoulou, Z., Pit-Claudel, C., Protzenko, J., et al.: Meta-f: Proof automation with SMT, tactics, and metaprograms. In: Programming Languages and Systems: 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, 6–11 April 2019, Proceedings, pp. 30–59. Springer, Cham (2019)
24. McCreight, A., Shao, Z., Lin, C., Li, L.: A general framework for certifying garbage collectors and their mutators. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 468–479 (2007)
25. Mccreight, A.E.: The mechanized verification of garbage collector implementations. PhD thesis, Yale University (2008)
26. Mo, M.Y.: Chrome in-the-wild bug analysis: CVE-2021-37975 (2021). https://securitylab.github.com/research/in_the_wild_chrome_cve_2021_37975/
27. Myreen, M.O.: Reusable verification of a copying collector. In: International Conference on Verified Software: Theories, Tools, and Experiments, pp. 142–156. Springer (2010)
28. Protzenko, J., Zinzindohoué, J.-K., Rastogi, A., Ramananandro, T., Wang, P., Zanella-Béguelin, S., Delignat-Lavaud, A., Hritcu, C., Bhargavan, K., Fournet, C., Swamy, N.: Verified Low-Level Programming Embedded in F*. arXiV Preprint (2018). https://doi.org/10.48550/arXiv.1703.00053
29. Ramananandro, T., Delignat-Lavaud, A., Fournet, C., Swamy, N., Chajed, T., Kobeissi, N., Protzenko, J.: {EverParse}: verified secure {Zero-Copy} parsers for authenticated message formats. In: 28th USENIX Security Symposium (USENIX Security 19), pp. 1465–1482 (2019)
30. Reitz, A., Fromherz, A., Protzenko, J.: Starmalloc: verifying a modern, hardened memory allocator. Proc. ACM Program. Lang. (2024). https://doi.org/10.1145/3689773

31. Reynolds, J.C.: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science. IEEE Computer Society (2002)
32. Russinoff, D.M.: A mechanically verified incremental garbage collector. Formal Asp. Comput. **6**(4), 359–390 (1994)
33. Sivaramakrishnan, K., Dolan, S., White, L., Jaffer, S., Kelly, T., Sahoo, A., Parimala, S., Dhiman, A., Madhavapeddy, A.: Retrofitting parallelism onto OCAML. Proc ACM Program. Lang. **4**(ICFP), 1–30 (2020)
34. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.-Y., Kohlweiss, M., et al.: Dependent types and multi-monadic effects in f. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 256–270 (2016)
35. Wan, Z., Lo, D., Xia, X., Cai, L.: Bug characteristics in blockchain systems: a large-scale empirical study. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 413–424. IEEE (2017)
36. Wang, S., Cao, Q., Mohan, A., Hobor, A.: Certifying graph-manipulating C programs via localizations within data structures. Proc. ACM Program. Lang. **3**(OOPSLA) (2019). https://doi.org/10.1145/3360597
37. Xu, B., Moss, E., Blackburn, S.M.: Towards a model checking framework for a new collector framework. In: Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes, pp. 128–139 (2022)
38. Yuasa, T.: Real-time garbage collection on general-purpose machines. J. Syst. Softw. **11**(3), 181–198 (1990)
39. 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, 26–29 September 2017, Proceedings, vol. 8, pp. 496–513. Springer (2017)