

A Mechanically Verified Garbage Collector for OCaml

Sheera Shamsu¹, Dipesh Kafle², Dhruv Maroo¹, Kartik Nagar¹,
Karthikeyan Bhargavan³, KC Sivaramakrishnan⁴

¹IIT Madras, Chennai, 600036, India.

²NIT Trichy, Trichy, 620015, India.

³Inria, Paris, 75014, France.

⁴Tarides and IIT Madras, Chennai, 600036, India.

Contributing authors: sheera.shms@gmail.com;
dipesh.kaphle111@gmail.com; dhruvsmaroo@gmail.com;
nagark@cse.iitm.ac.in; karthik.bhargavan@gmail.com;
kcsrkc@cse.iitm.ac.in;

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 integrate

001
002
003
004
005
006
007
008
009
010
011
012
013
014
015
016
017
018
019
020
021
022
023
024
025
026
027
028
029
030
031
032
033
034
035
036
037
038
039
040
041
042
043
044
045
046

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

050 **Keywords:** formal verification, mark and sweep garbage collection, F*, Low*,
051 mechanized formal proofs, graph traversal proofs

052
053
054

055 1 Introduction

056
057 Many contemporary programming languages, including OCaml, utilize a garbage col-
058 lector (GC) to manage memory automatically. This reliance on automatic memory
059 management ensures memory safety, effectively preventing the occurrence of many
060 security vulnerabilities [1, 2]. However, it is worth noting that the GC itself is often
061 implemented in a language like C, which lacks inherent memory safety guarantees.
062 Additionally, memory managers for modern languages often feature complex function-
063 alities such as multiple generations, diverse memory layout for supporting different
064 language features, incremental collection, and concurrency. These complexities make it
065 challenging to ascertain the correctness of GC implementations, often resulting in the
066 introduction of memory safety bugs.

067 The GC used in OCaml version 4 is generational and features two heap generations:
068 the minor and major heaps. The minor heap employs copying collection, while the
069 major heap utilizes an incremental mark and sweep GC to automatically reclaim
070 memory. Both the minor and the major GC is implemented in C. Given that the
071 memory safety of OCaml depends on the correctness of the GC, we wondered whether
072 we could formally verify the correctness of the OCaml GC. Some previous works [3, 4]
073 have verified the correctness of abstract GC models, which risk missing out on subtle
074 bugs due to the air gap between the abstract model and the GC implementation. Our
075 goal in this work is to develop a verified GC for OCaml, through a proof-oriented
076 approach, such that executable code compatible with the OCaml compiler can be
077 extracted directly from the verification artifact.

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

085 At its core, garbage collection relies on accurately identifying objects designated
086 as garbage, regardless of the specific GC algorithm employed. In a tracing garbage
087 collector, the allocated objects and their interconnections form a graph, transforming
088 the task of identifying garbage objects into a graph traversal problem. Starting from
089 the root sets of program variables (stack, heap, and globals), solving the graph traversal
090 problem essentially involves identifying all objects transitively reachable from the root
091 set. These reachable objects are termed as *live* objects. In terms of garbage collection,
092

it is imperative that a garbage collector does not free any live objects, a requirement known as the *safety* or *soundness* property of a garbage collector. 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* [5, 6]. F* is a proof-oriented, solver-assisted programming language, along with its low-level subset Low* [7]. 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 [8], 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 [7].

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.

139 Our experimental results demonstrate that the verified GC is competitive with the
 140 existing OCaml GC in terms of performance.

141

142

143

Table 1: Comparison with the related work

144

Author	Mode	Algo	Spec	Code	Heap Layout
Hawblitzel et al. [9]	stw	mark & sweep	algo. specific	assembly	C#
Hawblitzel et al. [9]	stw	copying	algo. specific	assembly	C#
Ericsson et al. [10]	generational	copying	reachability	assembly	CakeML
McCreight [11]	incremental	copying	reachability	assembly	-
Gammie et al. [3]	concurrent	mark & sweep	reachability	model only	-
Zakowski et al. [4]	concurrent	mark & sweep	reachability	model only	-
Our work	stw	mark & sweep	reachability	C	OCaml

150

151

152

153

154

155

156

157

While numerous previous works [12–18] 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.

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

174

175

176

177

178

179

180

181

182

183

184

A notable example of a stop-the-world (STW) mark-and-sweep GC verification is the work of Hawblitzel et al. [9], 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, 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. [10], 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. [11] 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. [3] and Zakowski et al. [4] 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 Section 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. Section 2 offers an overview of OCaml memory management, tricolor mark and sweep garbage collection, and provides an introduction to F* and Low*. Section 3 outlines the abstract GC correctness specifications, and Section 4 describes the path towards a verified OCaml GC. Section 5 is dedicated to OCaml-specific GC correctness specifications. Section 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 Section 7. In Section 8, we discuss how our approach can be extended for copying and incremental collection, thus laying a roadmap towards extensions of our verified GC. Section 9 examines related work, while Section 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 concurrent and a parallel GC [19], the details of which we omit as it is not in the scope of the current work.

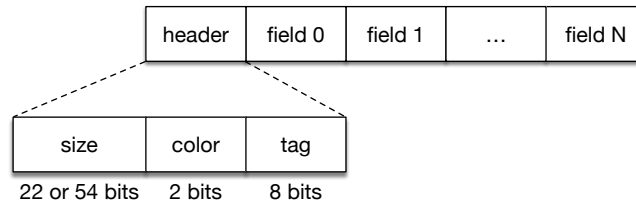
The OCaml uses a uniform memory representation for OCaml values. A value is a single memory word that either represents an immediate integer or a pointer to some other memory. The OCaml runtime, written in C, manages the OCaml heap. The heap is a collection of memory regions obtained from the operating 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 [20]. A typical OCaml object is represented as a *block* in the OCaml heap, which has a header followed by variable number of fields. Figure 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, gray, 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

231
232
233
234
235
236
237



238 **Fig. 1:** Layout of an OCaml object. The header and each field of an OCaml object
239 occupy one word.
240
241

242 least significant bit (LSB) to be 1, with the rest of the bits encoding the value of the
243 data type. Thus, OCaml integers are 31-bits and 63-bits long on 32-bit and 64-bit
244 platforms. Pointers are always guaranteed to be word-aligned and have 0 as their LSB.
245 While the representation is not compact, it simplifies the GC; by examining the LSB,
246 the GC can decide whether the value is a pointer or an immediate.

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

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

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

261
262
263
264
265

```
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
```

266 The values are the *environment* of the closure, which are the values of the free
267 variables. Each entrypoint is either a 2- or 3- word record with the code pointer, closure
268 information and, in the case of a closure with `arity > 1`, another code pointer. The
269 closure information contains the arity of the closure. Importantly, the start of the
270 environment information encodes the offset to the environment from the start of the
271 closure. As an example, a closure with arity 2 and an environment of size 2 would have
272 the following layout shown in Figure 2. The start of environment information says that
273 the environment starts from the field index 3 in the closure object. The GC only needs
274 to scan the environment and uses the start of environment information to locate the
275 environment in the closure.
276

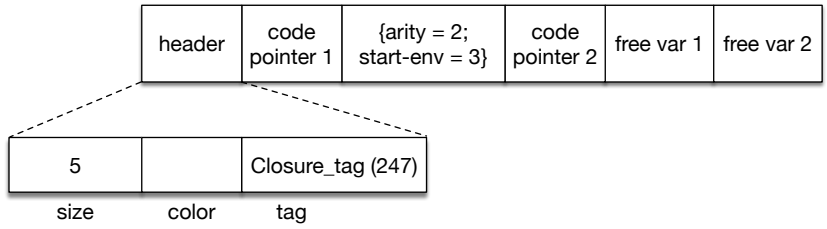


Fig. 2: The layout of a closure object with arity 2 and environment size 2.

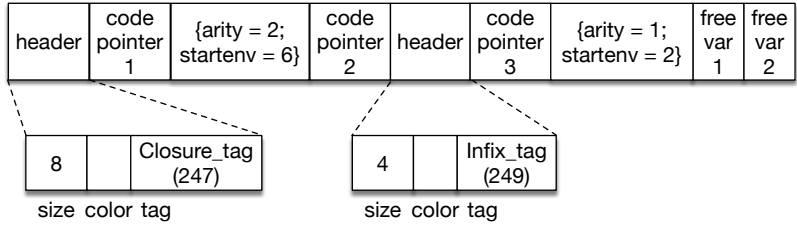
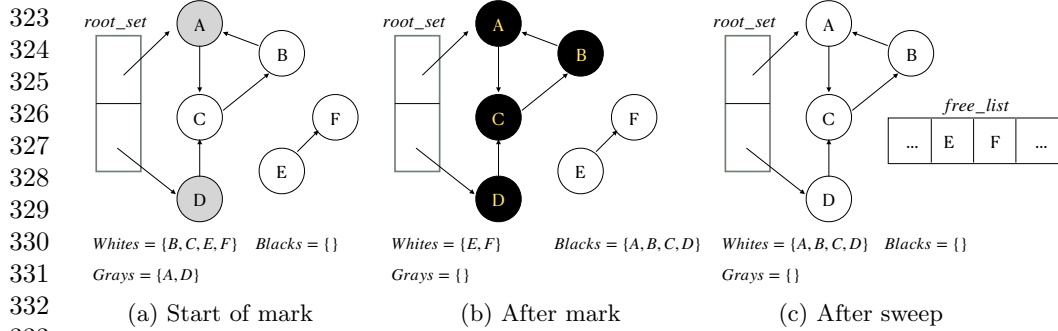


Fig. 3: The layout of a mutually-recursive closure object layout with arities 2 and 1 and environment size 2.

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, Figure 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.

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. Figure 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 gray and are added to the *mark stack* (Figure 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 gray and every live object is reachable from a gray object transitively through a sequence of white objects.



334 **Fig. 4:** Tricolor marking. Initially the objects that are directly pointed by the root
335 pointers are gray in the heap. The stack is populated with such objects first.
336

337
338 The mark phase proceeds by popping a gray object from the mark stack. All of
339 the white successors of the popped object are marked gray and pushed onto the mark
340 stack. Finally, the popped object is marked black. Thus, we also have the invariant
341 that a black object never points to a white object. When the mark stack is empty, the
342 mark phase ends (Figure 4b). We are guaranteed that all the reachable objects are
343 marked black, all unreachable objects remain white and there are no gray objects.
344

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

352 2.3 F* and Low*

353
354 Our correct-by-construction GC is implemented and verified in F* and its low-level
355 subset Low*. F* is a general-purpose proof-oriented programming language, that
356 supports both purely functional and effectful programs. In F*, the expressive power
357 of dependent types is combined with proof automation based on SMT solving and
358 tactic-based interactive theorem proving. After verification, F* programs are usually
359 extracted to OCaml or F#. The keyword `val` is used to define a function signature,
360 whereas functions are defined using the keyword `let` (`let rec` for recursive functions). A
361 variable `x` is declared in the form `x:t`, which means `x` has type `t`. F* provides support
362 for refinement types, which helps to express more properties on the type of the variable.
363 For instance, the type of non-negative integers, `nat`, is defined as `n:int {n ≥ 0}`.

364 **Low*** [7] is a subset of F* with restricted features that allows a programmer to
365 write verified low-level code that can be extracted to C. In Low*, the full expressiveness
366 of F* can be used in proofs and specifications, while also exposing low-level details
367 such as the memory layout which facilitates the development of verified, low-level code.
368


```

module HA = FStar.HyperStack.All 369
module ST = FStar.HyperStack.ST 370
module B = LowStar.Buffer 371

let swap (r0 r1 : B.buffer UInt8.t) 372
: HA.Stack (unit) 373
(* PRE-CONDITIONS *) 375
(* B.live predicate ensures that buffers must be allocated before 376
their use *) 377
(λ m -> B.live m r0 ∧ B.live m r1 ∧ 378
(* Unit length buffers *) 379
B.length r0 == 1 ∧ B.length r1 == 1 ∧ 380
(* Buffer memory locations are not aliased *) 381
B.loc_disjoint (B.loc_buffer r0) (B.loc_buffer r1)) 382
383
(* POST-CONDITIONS *) 384
(λ m0 _ m1 -> B.live m0 r0 ∧ B.live m1 r1 ∧ 385
(* Explicitly specify which memory locations is modified *) 386
(B.modifies (B.loc_union (B.loc_buffer r0) 387
(B.loc_buffer r1)) m0 m1) ∧ 388
(* Encode functional correctness *) 389
Seq.index (B.as_seq m1 r0) 0 == Seq.index (B.as_seq m0 r1) 0 ∧ 390
Seq.index (B.as_seq m1 r1) 0 == Seq.index (B.as_seq m0 r0) 0) = 391
(* Initial memory m0 *) 392
let m0 = ST.get() in 393
(* Initial values in the single length buffers r0 and r1 *) 394
let r0_val = !*r0 in 395
let r1_val = !*r1 in 396
(* Asserts that, if we convert the buffer to its funtional seq 397
data type counter part, the value at index 0 is r0_val and 398
similarly r1_val *) 399
assert (Seq.index (B.as_seq m0 r0) 0 == r0_val); 400
assert (Seq.index (B.as_seq m0 r1) 0 == r1_val); 401
(* Updates r0 at index 0 as r1_val and r1 at index 0 as r0_val *) 402
r0.(0ul) <- r1_val; 403
r1.(0ul) <- r0_val; 404
(* Get the new memory state after the swap *) 405
let m1 = ST.get() in 406
(* Asserts that the values are swapped *) 407
assert (Seq.index (B.as_seq m1 r1) 0 == r0_val); 408
assert (Seq.index (B.as_seq m1 r0) 0 == r1_val); 409
(* Return type is unit, equivalent to C void *) 410
() 411
312
313
314

```

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

```

415 void swap(uint8_t *r0, uint8_t *r1)
416 {
417     uint8_t r0_val = *r0;
418     uint8_t r1_val = *r1;
419     r0[0U] = r1_val;
420     r1[0U] = r0_val;
421 }

```

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

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 Figure 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. `Uln8.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 Figure 6.

3 Abstract GC Correctness

From the discussion in Section 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 $\text{objs}(h)$ 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 $\text{allocs}(h)$ denote the set of allocated (not free) objects represented by their ids in h . Let $\text{ptrs}(x,h)$ be the set of ids of the objects pointed to by x . Let $\text{data}(x,h)$ 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 $\omega(h)$ iff $(\forall x, y. x \in \text{allocs}(h) \wedge y \in \text{ptrs}(x,h) \implies y \in \text{allocs}(h))$

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

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

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. $(\forall x. x \in V \implies \text{data}(x,h_0) = \text{data}(x,h_1))$

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

```

507 void mark_and_sweep_GC (uint8_t *hp, uint64_t *st, uint64_t *tp,
508                        uint64_t *r, uint64_t r_len, uint64_t *sw,
509                        uint64_t *fp) {
510     // GC initialization phase starts with pushing of roots
511     // into the mark stack
512     darken_roots (hp, st, tp, r, r_len);
513     // GC mark phase is dfs that operates on different OCaml objects
514     mark (hp, st, tp);
515     // GC sweep phase frees unreachable objects and updates the free list
516     sweep (hp, sw, fp);
517 }

```

Fig. 7: Extract C code for the top-level stop-the-world mark-and-sweep GC function.

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 [10], which is a verified copying collector for CakeML also captures GC correctness similar to Definition ???. In Section 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 [21].

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 (Section 5).

4 Towards a verified OCaml GC

In this section, we present our approach to verify a practical garbage collector for OCaml. As mentioned in Section 2, OCaml uses a generational and incremental garbage collector, 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 Section 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 [10, 22]. 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 Section 1, the verified GC code written in Low* can be extracted 553
to C. Figure 7 shows the top-level GC function extracted from the verified GC code 554
on a 64-bit platform. No change has been made apart from renaming the functions for 555
readability. Our heap `hp` is a single, contiguous, fixed-size byte buffer of size `heap_size`. 556
The GC takes as inputs an array of roots `r` of length `r_len`, a mark stack array `st` with 557
a stack top pointer `tp` that indexes into the stack, a sweep pointer `sw` and a free list 558
pointer `fp`. Note that `tp`, `sw` and `fp` are singleton buffers containing the stack top pointer, 559
sweep pointer and free list pointer respectively. The free list is a singly linked list of 560
free objects in the heap, which is implicitly stored through the fields of the objects. 561
Recall that free objects are colored blue. Like OCaml, we assume that zero length 562
objects are not on the heap. This implies that each object has at least one field, which 563
we use to store the `next` pointer for the free list. Initially, `st` is empty, `tp` points to the 564
stack base address, `sw` points to the start of the heap (i.e. `hp`), while `fp` points to the 565
first blue object. The GC first calls `darken_roots` to gray all the roots in `r` and pushes 566
them onto the mark stack `st` and suitably updates the stack top pointer `tp`. 567

Next is the `mark` function. We start with a base implementation first (Figure 8), 568
where there is no distinction between different types of objects. The implementation 569
here is to enable us to establish the base invariants necessary to do the verification. 570
Then we extend the base implementation to handle objects with `No_scan_tag` (Figure 9) 571
and finally `closure` and `infix` objects (Figure 10). 572

Let us first discuss the base version of the `mark` function as shown in Figure 8. The 573
`mark` function repeatedly calls `mark_body` until the stack is empty. `mark_body` pops the 574
object `x` from the top of the stack and finds its header address `h_x` using the function 575
`hd_address`. Then the color bits of the value pointed by `h_x` are made `black` through 576
`colorHeader`. `wosize` returns the object size in words stored at `h_x` and after which `darken` 577
iterates through all the fields of `x`, calling `darken_body` on the fields. `darken_body` darkens 578
the white objects (i.e. turning them gray) and pushes the field pointers onto the mark 579
stack as necessary. 580

The code snippets in Figure 8 hints at the verification challenge in front of us. Given 581
that we are in C, we have to ensure the accesses are memory safe, i.e., all memory 582
accesses are to valid memory. Observe that implementation works by coloring the 583
header words with bitwise operations. Hence, we need to reason about the correctness 584
of bitwise arithmetic, also ensuring that the change in color bits does not affect `wosize` 585
and `tag` of the object, which are also stored in the same header word. 586

The version of `mark` function as shown in Figure 9, incorporates the usage of the 587
`tag` bits which are part of the OCaml object layout. Here, `tag` is used to determine 588
whether the newly popped out object from mark stack needs to be scanned by the GC. 589
Recall from Section 2.1 that any object with a tag greater than or equal to `no_scan` 590
(value 251) is not scanned by the GC. In this case, the `mark` skips scanning this object 591
and moves on to next object from the mark stack. 592

The third version of the `mark` function, shown in Figure 10, further extends the 593
marking process for objects with tag less than `no_scan`. In particular, it checks whether 594
the object under consideration is a `closure` object or an `infix` object. `mark_body` calls 595
`darken_wrapper` instead of `darken`, which decides the starting address of fields of the 596
particular object under consideration. In the case of `closure` objects, as explained in 597
598

```

599 void mark(uint8_t *hp, uint64_t *st, uint64_t *tp) {
600     while (*tp > (uint64_t)0U)
601         mark_body(hp, st, tp);
602 }
603
604 void mark_body(uint8_t *hp, uint64_t *st, uint64_t *tp) {
605     tp[0U] = *tp - (uint64_t)1U; // Decrement tp
606     uint64_t x = st[*tp];
607     uint64_t h_x = hd_address (x);
608     colorHeader (hp, h_x, black);
609     uint64_t wz = wosize (h_x, hp);
610     darken (hp, st, tp, h_x, (uint64_t)1U);
611 }
612
613 void darken (uint8_t *hp, uint64_t *st,
614             uint64_t *tp, uint64_t h_addr, uint64_t j) {
615     uint64_t wz = wosize (h_addr, hp);
616     for (uint32_t i = j; i < (wz + (uint64_t)1U)); i++) {
617         darken_body(hp, st, tp, h_addr, i);
618     }
619 }
620
621 void darken_body (uint8_t *hp, uint64_t *st, uint64_t *tp,
622                 uint64_t h_addr, uint64_t i) {
623     uint64_t succ_indx = h_addr + i * mword;
624     uint64_t succ = load64 (hp + succ_indx);
625     uint64_t c = color (hd_address (succ), hp);
626     if (isPointer (succ_indx, hp)) {
627         if (c == white) {
628             push_to_stack(hp, st, tp, succ);
629         }
630     }
631 }
632 }

```

Fig. 8: Base version of mark and darken function

```

635 void mark_body(..omitted..) {
636     // Code omitted, same as before...
637     uint64_t tg = tag (h_x, hp);
638     if (tg < (uint64_t)251U) {
639         darken (hp, st, tp, h_x, (uint64_t)1U);
640     }
641 }
642 }
643
644

```

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

```

void mark_body (...omitted...) {                               645
    // Omitted...                                             646
    if (tg < (uint64_t)251U) {                                  647
        // Wrapper function for darken                         648
        darken_wrapper (hp, st, tp, h_x);                      649
    }                                                         650
}                                                             651
                                                            652
void darken_wrapper (...omitted...) {                          653
    // If the object is closure objs                           654
    if (tag(h_x, hp) == (uint64_t)247U) {                      655
        uint64_t x = f_address(h_x);                           656
        // Start of environment has to be extracted for closure objects 657
        uint64_t start_env = start_env_clos_info (hp, x);      658
        darken (hp, st, tp, h_x, start_env + (uint64_t)1U);   659
    } else {                                                  660
        darken (hp, st, tp, h_x, (uint64_t)1U);               661
    }                                                         662
}                                                             663
                                                            664
// Darken remains the same as that in base version           665
void darken_body(...omitted...) {                              666
    // Omitted...                                             667
    if (isPointer(succ_indx, hp)) {                            668
        uint64_t h_addr_succ = hd_address(succ);              669
        uint64_t tg = tag (h_addr_succ, hp);                  670
        // If the field points to an infix object             671
        if (tg == (uint64_t)249U) {                            672
            // Finds the parent closure                         673
            uint64_t parent_hdr = parent (hp, h_addr, i);     674
            darken_helper (hp, st, tp, parent_hdr);           675
        } else {                                              676
            darken_helper (hp, st, tp, h_addr_succ);          677
        }                                                     678
    }                                                         679
}                                                             680
                                                            681
void darken_helper(...omitted...) {                             682
    if (color(hdr_id, hp) == white) {                          683
        push_to_stack (hp, st, tp, hdr_id);                   684
    }                                                         685
}                                                             686
                                                            687
Fig. 10: Version of mark and darken function that deals with closure and infix 688
objects                                                         689
                                                            690

```



```

691 void sweep (uint8_t *g, uint64_t *sw, uint64_t *fp,
692             uint64_t limit, uint64_t mword) {
693     while (*sw < limit) {
694         uint64_t curr_obj_ptr = *sw;
695         uint64_t curr_header = hd_address(curr_obj_ptr);
696         uint64_t wz = wosize_of_block(curr_header, g);
697         uint64_t next_header = curr_header + (wz + 1ULL) * mword;
698         uint64_t next_obj_ptr = next_header + mword;
699         sweep_body (g, sw, fp);
700         sw[0U] = next_obj_ptr;
701     }
702 }
703
704 void sweep_body (uint8_t *g, uint64_t *sw, uint64_t *fp) {
705     uint64_t curr_obj_ptr = *sw;
706     uint64_t curr_header = hd_address(curr_obj_ptr);
707     uint64_t c = color_of_block(curr_header, g);
708     uint64_t wz = wosize_of_block(curr_header, g);
709
710     if (c == white || c == blue) {
711         colorHeader(g, curr_header, blue);
712         uint64_t fp_val = *fp;
713         uint32_t x1 = fp_val;
714         store64_le(g + x1, curr_obj_ptr);
715         fp[0U] = curr_obj_ptr;
716     } else {
717         colorHeader(g, curr_header, white);
718     }
719 }
720

```

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

723
724
725 Section 2, the offset of the environment need to be extracted first. The details of
726 the extraction is not shown for brevity. Another change is in `darken_body`, where, if a
727 field points to an `infix_object`, then the *parent* closure object is determined. This parent
728 closure is the one that is darkened by the GC. We note that this goes beyond just a
729 simple DFS traversal, and the details of these operations are necessary to reason about
730 the correctness of the GC.

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

736

```

noeq type graph (#a:etype) = {
  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

```

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

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 Figure 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 Section 3 for our GC compatible with OCaml. We express this specification in F* as is done in our artifact.

5.1 Basic Definitions

We first define the object graph in F* in Figure 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 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 Figure 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)

783 (& is product type operator in F^*). The `reach g x y` predicate encodes a proof of
 784 reachability from vertex x to vertex y in graph g . There are two ways to construct this
 785 proof: either through `ReachRefl` which encodes that every vertex is reachable from itself,
 786 or through `ReachTrans`, which requires a proof of reachability from x to z , and an edge
 787 in g from z to y , captured by the type refinement `edge g z y`.

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

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

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

814 We now describe a few definitions, which are not shown in the code. `wosize_t`, `color_t`
 815 and `tag_t` define the type of `wosize`, `color` and `tag` of an object, respectively. The functions
 816 `wosize h x h`, `color h x h` and `tag h x h` returns the `wosize`, `color` and `tag` respectively of
 817 the object x with header address `h_x` in heap h . The value stored at a heap address x in
 818 a heap h is read using a function `r_word h x`. Similarly, `w_word h x` writes to h in location
 819 specified by x .

820 Unlike the OCaml runtime, in the formalisation, for convenience, we address the
 821 fields from the header address of an object. Hence, the first field will have the offset 1.
 822 The function `valid_field_number` (shown in Figure 14), checks whether a field number is
 823 valid. A field number i is valid only if it lies within the range of 1 and the `wosize` of the
 824 object. The function `field` reads the i^{th} field of object x in h , if i is a valid field number
 825 for the object x . We define a boolean predicate `isPointer i h = U64.logand (r_word h i) 1UL`
 826 `= 0`, which holds when the value at address i holds a pointer (the least-significant bit
 827
 828

```

829
830
831 //Machine integers
832 module U64 = FStar.UInt64
833 module U8 = FStar.UInt8
834 let mword = 8UL
835 val heap_size : n:int{n `mod` U64.v mword == 0 ^ n >=16 ^
836   n < 1099511627776}
837 (* heap is a sequence of 8 bit unsigned machine integers *)
838 type heap = h:seq U8.t{length h == heap_size}
839 (* A valid heap address *)
840 type hp_addr = addr:U64.t {U64.v addr < heap_size ^
841   is_multiple_of_mword addr}
842 (* object address *)
843 type obj_addr = x:hp_addr {U64.v x >= mword}
844 (* header address *)
845 type hdr_addr = x:hp_addr {U64.v x + mword < heap_size}
846
847 (* header address from object address *)
848 let hd_address (o:obj_addr) = U64.sub o mword
849
850 (* object address from header address *)
851 let f_address (h:hdr_addr) = U64.add h mword
852
853 let valid_field_number (i:U64.t) (h:heap) (x:obj_addr) =
854   i >= 1 ^ i <= wosize(hd_address x, h)
855
856 let field_addr (x:obj_addr) (h:heap)
857   (i:U64.t{valid_field_number i h x}) =
858   U64.add (hd_address x) (U64.mul i mword)
859
860 (* Field reads of ith field of object x *)
861 let field (x:obj_addr) (h:heap)
862   (i:U64.t{valid_field_number i h x}) =
863   r_word h (field_addr x h i)
864
865 (* allocs(h) is the set of allocated objects in the heap *)
866 let well_formed_heap h =
867   (∀ x. seq.mem x (allocs (h)) ⇒
868     (∀ (i:U64.t{valid_field_number i h x}).
869       isPointer (field_addr x h i) h ⇒
870         (field x h i) ∈ allocs(h)))
871
872
873
874

```

Fig. 14: Basic Definitions in F*

875 is 0). The predicate `well_formed_heap` is the instantiation of the abstract well-formed
876 heap (Definition 1) defined in Section 3.

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

887

888 5.2 Specification for GC functions

889

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

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

910

911 5.3 Object graph construction

912

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

917 OCaml features such as closure and infix objects also affect graph construction,
918 as these objects have a different layouts to regular objects and influence how the GC
919 scans the objects. For simplifying the presentation, the graph construction of Figure 16
920 ignores closure and infix objects and objects which only have opaque bytes. The details

```

921
922
923 (* Product type *)
924 type st_hp = seq obj_addr & heap
925 type hp_fp = heap & obj_addr
926
927 val darken_roots (h:heap) (st:seq obj_addr) (r_list:seq obj_addr)
928   : Pure (st_hp)
929   (requires (* Only core conditions shown *))
930   (*1*) well_formed_heap (h) ∧
931   (*2*) (∀ x. x ∈ h_objs(h) ⇒ (x ∈ whites(h) ∨ (x ∈ blues(h))))
932   (*3*) (∀ x. x ∈ st ⇔ hd_address x ∈ greys(h))
933   (ensures (* Only core conditions shown *))
934   (*1*) (λ res → well_formed_heap (snd res) ∧
935   (*2*) (∀ x. x ∈ r_list ⇒ x ∈ (fst res)) ∧
936   (*3*) (∀ x i. (hd_address x) ∈ h_objs(h) ⇒
937             field x h i = field x (snd res) i)) ∧
938   (*4*) (∀ x. x ∈ (fst res) ⇔ hd_address x ∈ greys(snd res)))
939
940 val mark (h:heap) (st:seq obj_addr)
941   : Pure (heap)
942   (requires (* Only core conditions shown *))
943   (*1*) well_formed_heap (h) ∧
944   (*2*) (∀ x.x ∈ st ⇔ hd_address x ∈ greys(h))
945   (ensures (* Only core conditions shown *))
946   (*1*) (λ h1 → well_formed_heap (h1) ∧
947   (*2*) (∀ x i. (hd_address x) ∈ h_objs(h) ⇒
948             field x h i = field x h1 i) ∧
949   (*3*) (∀ x.x ∈ h_objs(h1) ⇒ (color (hd_address x h)1 ≠ grey)))
950
951 val sweep (h:heap) (curr_ptr:obj_addr) (fp:obj_addr)
952   : Pure (hp_fp)
953   (requires (* Only core conditions shown *))
954   (*1*) well_formed_heap (h) ∧
955   (*2*) (∀ x.x ∈ objs(h) ⇒ (color (hd_address x h) ≠ grey)) ∧
956   (ensures (* Only core conditions shown *))
957   (*1*) (λ h1, fp1 → well_formed_heap (h1) ∧
958   (*2*) (∀ x. x ∈ blacks(h) ⇔ x ∈ whites(h1)) ∧
959   (*3*) (∀ x. x ∈ whites(h) ∨ blues(h) ⇔ x ∈ blues(h1)) ∧
960   (*4*) (∀ x i. (hd_address x) ∈ h_allocs(h) ⇒
961             field x h i = field x h1 i) ∧
962   (*5*) (∀ x. x ∈ h_objs(h1) ⇒ x ∈ whites(h1) ∨ x ∈ blues(h1)))
963
964 Fig. 15: Algebraic properties of the constituent functions of our verified GC
965
966

```

```

967 module G = Spec.Graph
968 val edges_of_graph (s:seq obj_addr) (h:heap{well_formed_heap (h)})
969   : Tot (e:G.edge_set{ $\forall x y. \text{mem } x \text{ s} \implies \text{mem } (x,y) \text{ e} \iff$ 
970           ( $\exists i. \text{valid\_field\_number } i \text{ h } x \wedge$ 
971             isPointer (field_addr x h i) h  $\wedge$ 
972             y = field x h i}))
973 val graph_from_heap (h:heap)
974   : Pure (G.graph)
975   (requires well_formed_heap (h))
976   (ensures  $\lambda g \rightarrow g.\text{vertices} = \text{allocs } h \wedge$ 
977           g.edges = edges_of_graph allocs h)

```

Fig. 16: Constructing graph from the heap

on how to incorporate the additional features is described in Section 6.5. As shown in Figure 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 Figure 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 Figure 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 Figure 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 ??.

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.

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


```

val mark_reachability_lemma (h:heap) (st:seq obj_addr) 1013
    (r_list:seq obj_addr) 1014
: Lemma 1015
  (requires 1016
    (*1*) well_formed_heap (h) 1017
    (*2*) ( $\forall x. x \in st \iff hd\_address\ x \in greys(h)$ )  $\wedge$  1018
    (*3*) well_formed_heap (mark h st)) 1019
  (ensures 1020
    (*1*) (graph_from_heap (mark h st) = graph_from_heap h)  $\wedge$  1021
    (*2*) ( $\forall x\ y. y \in r\_list \wedge$  1022
      reach (graph_from_heap h) y x  $\iff x \in blacks(mark\ h\ st)$ )) 1023
    1024
    1025
    1026
val sweep_subgraph_lemma (h:heap) (r_list:seq obj_addr) 1027
    (curr_ptr:obj_addr) (fp:obj_addr) 1028
: Lemma 1029
  (requires 1030
    (*1*) well_formed_heap (h)  $\wedge$  1031
    (*2*) ( $\forall x. x \in h\_objs(h) \implies (color\ (hd\_address\ x\ h) \neq grey)$ )  $\wedge$  1032
    (*3*) well_formed_heap (sweep h curr_ptr fp)) 1033
  (ensures 1034
    (*1*) ( $\forall x. x \in graph\_from\_heap\ (sweep\ h\ curr\_ptr\ fp).vertices \iff$  1035
      x  $\in graph\_from\_heap\ (h).vertices \wedge$  1036
      ( $\exists y. y \in r\_list \wedge reach\ (graph\_from\_heap\ h) y\ x$ ))  $\wedge$  1037
    (*2*) ( $\forall x\ y. (x,y) \in graph\_from\_heap\ (sweep\ h\ curr\_ptr\ fp).edges \iff$  1038
      x  $\in graph\_from\_heap\ (sweep\ h\ curr\_ptr\ fp).vertices \wedge$  1039
      y  $\in graph\_from\_heap\ (sweep\ h\ curr\_ptr\ fp).vertices \wedge$  1040
      (x,y)  $\in graph\_from\_heap\ (h).edges$ )) 1041
    1042

```

Fig. 17: Mark reachability and reachable subgraph equivalence

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 Figure 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.

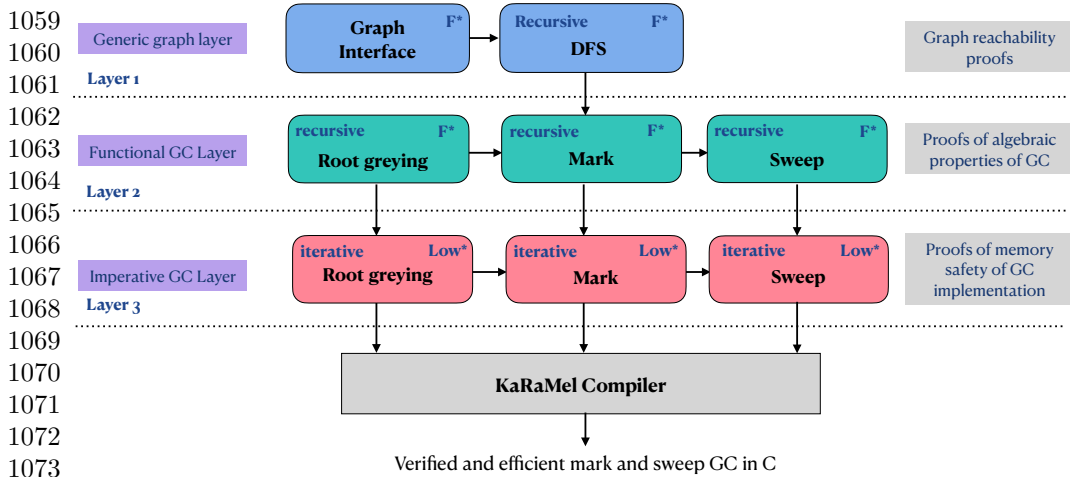


Fig. 18: The layering approach for GC verification

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 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 Figure 19, directly takes as input the object graph (whose type defined in Section 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 Figure 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 Figure 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

```

(* dfs calls dfs_body until stack empty.                               1105
   Inputs are graph, stack and visited set. *)                          1106
let rec dfs (g:graph) (st:seq U64.t) (vl:seq U64.t)                    1107
: Pure (seq U64.t)                                                    1108
  (requires ...)                                                    1109
  (ensures (λ res → ...)                                           1110
  (decreases (length g.vertices - length vl; length st)) =         1111
  if length st = 0 then vl                                          1112
  else                                                                1113
    let st1 ,vl1 = dfs_body g st vl in                             1114
    dfs g st1 vl1                                                1115
                                                                    1116
let dfs_body g st vl                                                1117
: Pure ...                                                            1118
  (requires ...)                                                    1119
  (ensures (λ res → ...) =                                          1120
    let x   = stack_top st in                                       1121
    let xs  = stack_rest st in                                       1122
    let s   = successors g x in                                       1123
    let vl1 = set_insert x vl in                                       1124
    let st1 = push_unvisited s xs vl1 in                                   1125
    (st1, vl1)                                                    1126

```

Fig. 19: Functional dfs

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 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 Figure 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.

```

1151 (* mark calls mark body until stack empty.
1152    Inputs are heap and stack *)
1153 let rec mark (h:heap) (st:seq obj_addr)
1154 : Pure (heap)
1155 (requires ...)
1156 (ensures (λ res → ...))
1157 (decreases (length allocs h - length blacks h;
1158             length st)) =
1159 if length st = 0 then h
1160 else
1161   let st1, h1 = mark_body h st in
1162   mark h1 st1
1163
1164 let mark_body (h:heap) (st:seq obj_addr)
1165 : Pure ...
1166 (requires ...)
1167 (ensures (λ res → ...)) =
1168 let x = stack_top st in
1169 let xs = stack_rest st in
1170 let h1 = colorHeader h x black in
1171 let st1 = darken h1 xs x 1UL in
1172 (st1, h1)

```

Fig. 20: mark implementation

```

1174
1175
1176
1177 (* r_list is the root set, stack is filled with r_list initially *)
1178 val dfs_reachability_lemma (g:graph) (st:seq obj_addr)
1179                             (vl:seq obj_addr) (r_list:seq obj_addr)
1180 : Lemma
1181 (requires
1182   (* Pre-conditions required to prove forward direction *)
1183   (*F1*) mutually_exclusive_sets st vl ∧
1184   (*F2*) (∀ y.y ∈ st ⇒ (∃ x.x ∈ r_list ∧ reach g x y) ∧
1185   (*F3*) (∀ y.y ∈ vl ⇒ (∃ x.x ∈ r_list ∧ reach g x y) ∧
1186
1187   (* Pre-conditions required to show the backward direction *)
1188   (*B1*) (∀ x y.x ∈ r_list ∧ reach g x y ⇒
1189           (∃ z.z ∈ st ∧ reach g z y) ∨ y ∈ vl)
1190   (*B2*) (∀ x y.x ∈ vl ∧ reach g x y ⇒
1191           (∃ z.z ∈ st ∧ reach g z y) ∨ y ∈ vl)
1192
1193   (ensures (∀ y.y ∈ (dfs g st vl) ⇔ (∃ x.x ∈ r_list ∧ reach g x y))

```

Fig. 21: Correctness specification of dfs

1196

```

val makeHeader (wz:wosize_t) (c:color_t) (t:tag_t)      1197
  : Pure (U64.t)                                       1198
  (requires True)                                     1199
  (ensures (λ res →                                     1200
    (U64.shift_right res 10UL = wz) ∧                1201
    (U64.logand (U64.shift_right res 8) 3UL = c) ∧    1202
    (U64.logand res 255UL = t)))                      1203
                                                    1204
val colorHeader (h:heap) (hdr:hdr_addr) (c:color_t)   1205
  : Pure (heap)                                       1206
  (requires (* Only core conditions shown *)          1207
    well_formed_heap (h) ∧                            1208
    valid_hdr hdr h)                                  1209
  (ensures λ res →                                    1210
    well_formed_heap (res) ∧                          1211
    valid_hdr hdr res ∧                                1212
    objs h = objs res ∧                              1213
    heap_same_except hdr h res ∧                     1214
    color hdr res = c ∧                              1215
    wosize hdr res = wosize hdr h ∧                 1216
    tag hdr res = tag hdr h ∧                       1217
    r_word res hdr =                                1218
      makeHeader (wosize hdr h) (c) (tag hdr h))     1219
  Fig. 22: Specifications of colorHeader              1220
                                                    1221

```

6.2.1 Specification of colorHeader

The specifications of `colorHeader` are shown in Figure 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 Figure 15.

6.2.2 Proof outline for the sub-functions of the GC

The specifications are shown in Figure 15. The `darken_roots` function pushes all the root pointers in `h.list` to an empty stack and then colors them as gray. Let `h0` be the initial heap before the GC and let `h1` be the heap resulted after `darken_roots`. Since the GC starts with `well_formed_heap h0` that contains only white and blue objects and

```

1243 val dfs_mark_equivalence_lemma (h:heap) (st:seq obj_addr)
1244                               (vl:seq obj_addr) (h_list:seq obj_addr)
1245   : Lemma
1246     (* Only important properties shown *)
1247     (requires (*1*) mutually_exclusive_sets st vl ∧
1248              (*2*) well_formed_heap(h) ∧
1249              (* stack invariant *)
1250              (*3*) (∀ x.x ∈ st ⇔ (hd_address x) ∈ grays(h))
1251              (* visited-list invariant *)
1252              (*4*) (∀ x.x ∈ vl ⇔ (hd_address x) ∈ blacks(h))
1253
1254     (ensures (∀ x. x ∈ (dfs (graph_from_heap h) st vl) ⇔
1255              (hd_address x) ∈ blacks(mark h st))))

```

Fig. 23: Behavioral equivalence between mark and dfs

1259 the fact that `h_list` contents are members of `allocs(h0)` implies that `well_formed_heap h1` is
1260 preserved as the only change to heap is coloring of `h_list` members from white to gray.
1261 Since both white and gray are considered as allocated objects, changing the color from
1262 white to gray still preserves the membership in allocated set. Therefore, `allocs(h1) =`
1263 `allocs(h0)`. Therefore the vertex set of both graphs constructed from `h0` and `h1` remains
1264 the same. Since the specification of `colorHeader` ensures that except the color bits of the
1265 address `hdr` in the heap, nothing else in the heap changes, the edge set of the graphs
1266 from `h0` and `h1` also remains the same. Therefore, both the graphs are the same. Since
1267 `darken_roots` starts with an empty stack, when the objects in `h_list` are pushed into the
1268 stack and colored them gray in the heap, the only gray objects in the heap are the
1269 ones that are pushed onto the stack.

1270 Let `h2` be the heap formed after `mark`. All the algebraic properties of `mark` as in
1271 Figure 15 can be proved by using the same approach as `darken_roots` and the fact that
1272 `mark` starts with a stack that contains all the gray and only the gray objects of the
1273 heap. When `mark` terminates after the stack becomes empty, there are no gray objects
1274 in the resultant heap after `mark`.

1275 Similarly the algebraic properties of `sweep` can also be directly proven using the
1276 specifications of `colorHeader`. Recall that, `sweep` changes the color of white objects to
1277 blue and black objects to white. As well as the first field of the free list pointer is
1278 changed to point to the newly created blue block during a sweep invocation. Since the
1279 well-formedness property only affects the allocated objects and the fact that free list
1280 pointer points to a blue object, with the help of some extra lemmas related to the `sweep`
1281 properties, F* can prove that the heap resulted after `sweep` remains as well-formed.

1283 6.2.3 Proof outline for mark_reachability_lemma

1284
1285 As the direct proof of reachability of `mark` requires reasoning about bit-wise arith-
1286 metic and graph reachability together, we have avoided that path due to the inherent
1287 complexities to handle such proofs with an SMT solver. Instead, we use the reachability
1288

proofs of `dfs` for `mark` by proving the program equivalence between `dfs` and `mark` using the lemma in Figure 23. 1289

Proof outline for `dfs_mark_equivalence_lemma`: Both `dfs` (in Figure 19) and `mark` (in Figure 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 gray 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`. 1290-1302

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 `h1` and `st1` are the results of one invocation of `mark.body` and `st2` and `vl2` 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: 1303-1310

- `well_formed_heap h1` 1311
- $(\forall x. x \in vl_2 \iff (hd_address\ x) \in blacks(h_1))$ 1312
- $(\forall x. x \in st_1 \iff (hd_address\ x) \in grays(h_1))$ 1313
- `st2 = st1` 1314

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 gray and gray objects become black. That means, the `allocs h` and `allocs h1` remains the same. Combining this property with the fact that the fields remain unchanged due to the coloring operation, the well-formedness of `h1` can be proved. This also ensures that the graphs formed from both `h` and `h1` 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 `vl2` and `h1`. While pushing the field pointers of the top of the stack, `mark.body` colors them gray, which ensures the `stack invariant` of `st1` with respect to `h1`. 1315-1326

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 `st2` and `st1` respectively. A comparison between these functions are shown in Figure 24. 1327-1330

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 1331-1333


```

1335 (* push_unvisited calls push_unvisited_body until successors is empty *)
1336 let push_unvisited_body s st vl j
1337   : Pure ...
1338   (requires j < length s...)
1339   (ensures (λ res → ...) =
1340   if (not (mem (index s j) (set_union st vl))) then
1341     let st2 = push (head s) st in
1342     st'
1343
1344 (* darken calls darken_body until i = wosize + 1 *)
1345 let darken_body h st h_addr i
1346   : Pure ...
1347   (requires ...)
1348   (ensures (λ res → ...) =
1349   let succ = r_word h (h_addr + i * mword) in
1350   if not (isPointer succ) then (st, h)
1351   else
1352     let c = color h (hd_address succ) in
1353     if not (c = white) then (st, h)
1354     else
1355       let h1, st1 = push_to_stack h st succ in
1356       (st1, h1)
1357

```

Fig. 24: A comparison of `push_unvisited_body` and `darken_body` functions

1360 unvisited (white) field pointers of x into the stack. `push_unvisited` and `darken` perform
1361 the bulk of the work by calling `push_unvisited_body` and `darken_body` respectively.

1362 The parameter j in `push_unvisited_body` indicates the index of the successor in s
1363 to be examined. Similarly, the parameter i in `darken_body` indicates the field number
1364 of x to be scanned in h . Recall that st and vl are mutually exclusive. The function
1365 `push_unvisited_body` decides whether an element is unvisited by checking the membership
1366 in st and in vl . Due to the invariants on st and vl , such an object will also be colored
1367 white in the heap. The same action is being performed by the `darken_body` as well. But
1368 the difficulty here is that s is already a filtered list of field pointers (i.e. successors),
1369 while the field scan in fields by `darken` may encounter non-pointer fields as well. Hence,
1370 there may not exist a direct one-to-one correspondence between the invocations of
1371 `push_unvisited_body` and `darken_body`. To get around this issue, we can make use of the
1372 observation that if the `field_slice` which starts at the i^{th} field of x in the heap returns
1373 the same set of field pointers as that of `successor_slice` that starts at j^{th} index of the
1374 successors list in s , then the stacks produced by `darken` that starts at field index i of x
1375 in h and `push_unvisited` that starts at index j of s are the same. Formally in F^* we prove
1376 the below lemma,

1377
1378
1379
1380

```

val darken_push_unvisited_produces_same_stack (h:heap) (st:seq obj_addr) 1381
                                              (vl:seq obj_addr) 1382
                                              (curr:hdr_addr) 1383
                                              (i:U64.t) (j:U64.t) 1384
: Lemma 1385
  (requires mutually_exclusive_sets st vl ∧ 1386
           well_formed_heap (h) ∧ 1387
           successor_slice s j = field_slice h hdr_addr i) 1388
  (ensures darken h st hdr_addr i = push_unvisited s st vl j) 1389

```

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 Figure 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 Figure 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:

$$(\forall x y. y \in \text{h_list} \wedge \text{reach}(\text{graph_from_heap } h) y x \iff (\text{hd_address } x) \in \text{whites}(\text{sweep } h \text{ curr_ptr } fp))$$

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 Def. ?? 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

1427 of challenges to be dealt with in this layer: the heap and stack are now modeled as
1428 fixed-length buffers, thus requiring proofs of absence of buffer overflows, the heap/stack
1429 mutations are now in-place instead of functional, thus requiring anti-aliasing proofs.
1430 This layer uses Low^* , which allows extraction to verified C code. To understand how
1431 Low^* ensures memory safety of the C implementations, let us first see the code listing
1432 of `mark` in Low^* as an example as shown in Figure 25 and its pre- and post-conditions
1433 in Figure 26. To differentiate from a functional implementation, imperative `mark` is
1434 qualified with a suffix `imp`. The function takes as input a buffer `hp` to store the heap,
1435 another buffer `st` to store the stack and `tp` to store the top pointer of `st`.

1436 Each of the Low^* functions takes a pre-condition (`requires` clause) on the initial
1437 memory state m and a post-condition (`ensures` clause) about the initial and the final
1438 memory states m_0 and m_1 respectively. For example, the `mark_imp` requires the pre and
1439 post conditions about the memory state as shown in Figure 26¹.

1440 The term `live m hp` states that `hp` is a live buffer in memory state m , where the
1441 location is specified by `loc_buffer hp`. The condition that the buffers `hp` and `st` should
1442 be disjoint in memory is captured in `disjoint (loc_hp) (loc_st)`. In the post-condition, a
1443 `modifies` clause is used which ensures that the buffers `hp`, `st` and `tp` got modified between
1444 the memory states m_0 and m_1 . The `as_seq` function takes as input a memory state m
1445 and a buffer and creates the functional `seq` equivalent of the buffer in m . We need
1446 to reason about the stack contents upto stack top only. Therefore, we take a slice or
1447 portion of the stack from the start of the stack upto the stack top. This is captured in
1448 `(slice seq_st0 0 (index (seq_tp0) 0))`. The final clause ensures that the functional equivalent
1449 of buffer `hp` in the final memory state is equivalent to running a functional `mark` with
1450 the functional equivalent of `hp` in the initial memory m_0 and the slice of the functional
1451 equivalent of `st` upto `tp` in m_0 . This clause ensures the output equivalence of functional
1452 `mark` and imperative `mark`. This way the Low^* specification ensures the memory safety
1453 of the GC implementation as well as the functional equivalence with the functional
1454 implementation. The `inv` and `body` functions are used to specify the loop invariants and
1455 the body of the loop respectively.

1456 But there is one more hurdle, because of the size limitations of concrete buffers.
1457 The allocated stack has a fixed-size. Therefore, there is a probability that the stack
1458 might overflow during `mark`. Low^* rightfully captures this caveat and fails to typecheck
1459 if no conditions are provided that prevents stack overflow. Hence, to work around this,
1460 we set the stack size equals to the heap size and prove a lemma that states that when
1461 there is a non-gray object in the heap, the stack top is less than the heap size. The
1462 maximum size required to store all the objects in the heap is heap size in the worst case.
1463 Since the stack preserves the `stack invariant`, existence of one non-gray object means the
1464 stack top is less than the heap size, and thus less than the stack size. Thus, there is
1465 room in the stack to store this non-gray object, which will be converted to gray once it
1466 enters the stack.

1467 Now, let us see, how we can establish the functional equivalence between the
1468 functional GC, where all algebraic GC properties and the equivalence with a `dfs` traversal
1469 is proved, and the imperative GC. In Low^* , we need to prove the program equivalence
1470 between the F^* and Low^* GC intrinsically, that is along with the implementation of
1471 _____

1472 ¹Some details have been elided. The complete specification can be found in the supplemental material.

```

let mark_imp hp st tp                                1473
:Stack unit                                         1474
(requires λ m → ...)                               1475
(ensures λ m0 _ m1 → ...) =                    1476
let inv m = ...(*loop invariants*)                 1477
let guard (t: bool) m = inv m ∧                    1478
  (t = true  ⇒ B.get m tp 0) > 0) ∧                1479
  (t = false ⇒ B.get m tp 0) = 0) in               1480
let test ()                                         1481
:Stack bool                                         1482
  (requires λ m → inv m)                            1483
  (ensures λ _ ret m1 → guard ret m1)          1484
  = (!*tp) >^ OUL in                                1485
let body ()                                         1486
: Stack unit                                        1487
  (requires λ m → guard true m)                     1488
  (ensures λ _ _ m1 → inv m1)                 1489
  = mark_heap_body_imp hp st tp in                  1490
C.Loops.while #(inv) #(guard) test body            1491

```

Fig. 25: A Low* implementation of mark

```

requires λ m →                                     1492
let loc_hp = loc_buffer hp in                       1493
let loc_st = loc_buffer st in                       1494
let loc_tp = loc_buffer tp in                       1495
live m hp ∧ live m st ∧ live m tp ∧                1496
disjoint (loc_hp) (loc_st) ∧ disjoint (loc_st) (loc_tp) ∧ 1497
disjoint (loc_hp) (loc_tp)                          1498
ensures λ m0 _ m1 →                               1499
let union = loc_union (loc_buffer hp)              1500
  (loc_union (loc_buffer st) (loc_buffer tp))      1501
in                                                  1502
let seq_st0 = as_seq m0 st in                       1503
let seq_hp0 = as_seq m0 hp in                       1504
let seq_tp0 = as_seq m0 tp in                       1505
let seq_hp1 = as_seq m1 hp in                       1506
let slice_st0 = slice seq_st0 0 (index (seq_tp0) 0) in 1507
live m1 hp ∧ live m1 st ∧ live m1 tp ∧            1508
(* Same disjoint clause as above *) ∧             1509
(modifies union m0 m1) ∧                            1510
seq_hp1 = mark seq_hp0 slice_st0                    1511

```

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

```

1519 val mark_sweep_gc_imp hp st tp rlist rlist_len sw fp
1520 :Stack unit
1521 (* Similar conditions for mark...omitted *)
1522 (requires λ m → ...)
1523 (ensures λ m0 _ m1 →
1524   let seq_st0 = as_seq m0 st in
1525   let seq_hp0 = as_seq m0 hp in
1526   let seq_r_list = as_seq m0 r_list in
1527   let seq_tp0 = as_seq m0 tp in
1528   let seq_rlist_len0 = as_seq m0 rlist_len in
1529   let seq_hp1 = as_seq m1 hp in
1530   let slice_rlist0 = slice seq_r_list0 0 (index (seq_rlist_len0) 0) in
1531   let slice_st0 = slice seq_st0 0 (index (seq_tp0) 0) in
1532     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 function. The specification is shown in Figure 27. As explained earlier, `hp`, `st`, `tp` are buffers representing 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 ??.

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 Figure 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 gray, and a free-list pointer `fp`. We elide additional pre-conditions in the `requires` clause that relate the arguments to the heap.

```

val end_to_end_correctness_theorem 1565
  (* Initial heap *) 1566
  (h_init:heap{well_formed_heap h_init}) 1567
  (* mark stack - contains grey objects *) 1568
  (st: seq Usize.t {pre_conditions_on_stack h_init st }) 1569
  (* root set *) 1570
  (roots : seq Usize.t{pre_conditions_on_root_set h_init roots}) 1571
  (* free list pointer *) 1572
  (fp:hp_addr{pre_conditions_on_free_list h_init fp}) 1573
: Lemma 1574
( requires 1575
  (* Pre-conditions elided for brevity. Important ones are: 1576
    + The mark stack [st] contains all the [roots]. 1577
    + All the grey objects in the heap are in the mark stack [st]. 1578
  *) ) 1579
( ensures 1580
  (* heap after mark *) 1581
  let h_mark = mark h_init st in 1582
  (* heap after sweep *) 1583
  let h_sweep = fst (sweep h_mark mword fp) in 1584
  (* graph at init *) 1585
  let g_init = graph_from_heap h_init in 1586
  (* graph after sweep *) 1587
  let g_sweep = graph_from_heap h_sweep in 1588
  (* GC preserves well-formedness of the heap *) 1589
  (* 1 *) well_formed_heap h_sweep ^ 1590
  1591
  (* GC preserves reachable object set *) 1592
  (* 2 *) (( $\forall x. x \in g\_sweep.vertices \iff$  1593
    ( $\exists o. mem\ o\ roots \wedge reach\ g\_init\ o\ x$ ))) ^ 1594
  1595
  (* GC preserves pointers between objects *) 1596
  (* 3 *) (( $\forall x. mem\ x\ (g\_sweep.vertices) \implies$  1597
    (successors g_init x) == 1598
    (successors g_sweep x))) ^ 1599
  1600
  (* The resultant heap objects are either white or blue only *) 1601
  (* 4 *) ( $\forall x. mem\ x\ (h\_objs\ h\_sweep) \implies$  1602
    color x h_sweep == white ^ 1603
    color x h_sweep == blue) ^ 1604
  1605
  (* No object field (either pointer or immediate) is modified *) 1606
  (* 5 *) field_reads_equal h_init h_sweep ) 1607
  1608
Fig. 28: Overall correctness theorem for the mark and sweep GC 1609
  1610

```

1611 The ensures clause captures the post-conditions after the execution of `mark` and
1612 `sweep`. The ensures clause is a conjunction of five properties. The first property states
1613 that the final heap after sweep `h_sweep` is well-formed. This corresponds to the first
1614 property in abstract GC correctness definition (Definition 5). The second property
1615 states that graph formed out of the final heap `g_sweep` will only have those objects that
1616 are reachable from the roots in the initial heap, and every reachable object in the initial
1617 heap is present in the final graph. The third property states that the edges between
1618 the reachable objects are preserved by the GC. The second and the third properties
1619 together correspond to the second property in Definition 5. The fourth property states
1620 that the resultant heap only has white and blue objects. The abstract GC correctness
1621 does not refer to object colours as the notion of GC colour is an implementation details
1622 of the GC. However, it is an important implementation detail that ensures that the
1623 GC leaves the heap in a consistent state for the next cycle. Finally, the fifth property
1624 states that fields (both immediate and pointer) of the non-blue objects are unmodified
1625 by the GC. This property is stronger than the third property in Definition 5, which
1626 only says that the data fields remain the same. Since the mark and sweep GC does not
1627 move objects, the pointer fields are also preserved.

1628 One might wonder why the end to end correctness theorem is defined in layer 2 (F^*)
1629 and not layer 3 (Low^*). As described in Section 6.3, our approach is to have the proofs
1630 of the GC correctness in Section 2 and then prove the equivalence of the imperative GC
1631 with the functional GC in Section 3 only focussing on the memory safety properties.

1632

1633 6.5 Extending the GC functionality

1634

1635 As we change the GC variant to deal with different types of OCaml objects (i.e., closure
1636 and infix objects), both the object graph construction, and the scanning of fields
1637 performed by `mark` needs to change in sync with each other. The graph construction acts
1638 as the bridge between the abstract graph world and the functional GC world and hence
1639 the graph construction should be carefully done to connect the two worlds together.

1640 In the case of the second version of `mark` (Figure 9), the edge set of an object with
1641 `No.scan_tag` is made empty. For closure objects, the edge set is constructed by scanning
1642 the fields that are stored from the start of the environment (See Section 2.1). During
1643 the edge set construction of an object, if the field points to an infix object, then the
1644 parent closure of that infix object is added as the successor, instead of the infix object.
1645 Also the definitions of well-formedness have to capture the property that the return
1646 sequence of `h.objs` should never have an infix object, as the infix object pointers are
1647 *interior* pointers. There are additional properties related to the closure info field of
1648 closure objects such as the minimum number of fields for a closure object should be at
1649 least 2 (See Section 2), which are also part of the well-formedness property. By adapting
1650 the graph construction, and with the help of some more additional lemmas, we have
1651 verified the third version of the `mark` function as mentioned in Section 4 (Figure 10).

1652

1653

1654

1655

1656

Table 2: Verification effort. Development effort (Dev effort) is in person-months.

Modules	#Lines	#Defns	#Lemmas	Verif time	Dev effort
Graph	4653	72	81	2m3s	3
DFS	657	1	9	2m5s	9
Functional GC	18401	65	218	120m	12
Imperative GC	2734	19	19	27m43s	3

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 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 (Figure 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.

1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702

1703 Note that our trusted code base now includes F*/Low* and the KaRaMel compiler
1704 which translates Low* to C. It has to be noted that these tools also serve as the trusted
1705 code bases for previous projects such as Everest [23] and Everparse [24], which focus
1706 on verifying cryptographic implementations and parsers.

1707 The verification through F*/Low* was challenging due to several reasons. F* uses
1708 Z3 SMT solver. Our verification conditions (VCs) are not restricted to decidable logics.
1709 While Z3 does best-effort reasoning, it may take a long time to prove some VCs or
1710 the proof does not terminate. We had to tune the timeouts for individual lemmas to
1711 get them to discharge. In addition, Z3 is also non-deterministic. The same VC may be
1712 discharged in one run and not in another depending upon the solver state. F* provides
1713 some support for dealing with proof instability, such as running in `quake` mode or using
1714 `proof-recovery` mode to recover from proof failures. However, the fundamental issue of
1715 non-determinism remains.

1716

1717 7.2 Integration with OCaml

1718

1719 Our goal in this work is to develop a practical verified GC for OCaml that can serve
1720 as a replacement for the unverified GC. We have successfully extracted the verified C
1721 code for the GC functionality from Low* using the KaRaMel compiler [7]. We have
1722 integrated the extracted code into the OCaml 4.14.1 runtime system, replacing the
1723 existing GC in the bytecode runtime.

1724 Unlike OCaml 4.14.1 GC, our verified GC is stop-the-world and non-generational.
1725 We use an unverified next-fit allocator written in Rust that allocates objects in the
1726 verified heap. As mentioned before, our heap is a single, contiguous block of memory
1727 (encoded as an F* buffer), into which the objects are allocated. The verification of the
1728 allocator is orthogonal to the focus of the work. There is a recent work on StarMalloc [25]
1729 which provides a verified, hardened memory allocator written in F*/Low*. We plan to
1730 investigate integrating StarMalloc with our verified GC in the future. When the heap
1731 is full, the verified GC is triggered. We use the existing root marking procedure in the
1732 OCaml runtime to darken the roots and push them to the verified mark stack. This is
1733 followed by the call to the verified mark and sweep function.

1734 We made the following small modifications to the extracted code to facilitate
1735 integration with the OCaml runtime. The first modification is in the sweep code, where
1736 we have implemented coalescing of consecutive free blocks. This is done to reduce
1737 fragmentation. The second modification is necessitated by the fact that infix objects
1738 do not appear in the mark stack in the verified GC, whereas they do (during root
1739 marking) in the OCaml runtime. Since the root marking is done by the OCaml runtime,
1740 we have added a wrapper function that inserts the parent closure of an infix object
1741 into the mark stack if an infix object appears as a GC root.

1742

1743 7.3 GC evaluation

1744 We evaluate the performance of the verified GC on a variety of benchmark programs
1745 from the Computer Language Benchmarks Game [26] as well as larger programs – `cpdf`
1746 (an industrial-strength pdf processing tool) and `yjson` (JSON processing library) –
1747 from the OCaml ecosystem. The larger programs have a deep dependency graph of
1748

Table 3: Benchmark characteristics. Alloc, Promote and maxRSS are in MB.

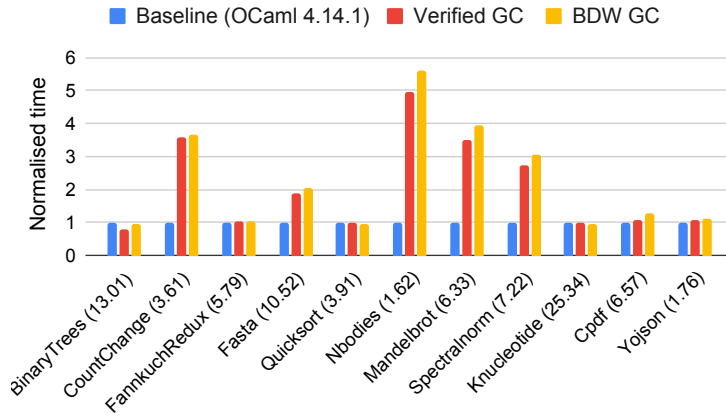
Benchmark	OCaml 4.14.1				Verified GC		
	Alloc	Promote	# Minor	# Major	MaxRSS	GCs	MaxRSS
BinaryTrees	15206	7900	7647	87	516	42	515
CountChange	905	140	458	11	145	5	260
FannkuchRedux	0.03	0	0	0	2.68	0	3.5
Fasta	3171	0.03	1569	4	44	72	67
Quicksort	19	0.02	1	0	22	0	22
Nbodies	808	0.04	405	2	4.66	3819	3.63
Mandelbrot	3009	0.13	1508	9	4.3	34201	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

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.

Figure 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 [27]. 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.

1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807



1808 **Fig. 29:** Normalized running time of different OCaml GCs. The numbers in the
1809 parenthesis next to the benchmark names are the running time in seconds for the
1810 baseline OCaml 4.14.1 GC.
1811

1812

1813 8 Extending the verified GC

1814

1815 In this section, we show how we can extend our verified GC to collectors that are
1816 different from our stop-the-world mark-and-sweep GC. For this exercise, we pick two
1817 collectors that are used in the current OCaml runtime system, namely (1) a copying
1818 collector and (2) an incremental version of the mark-and-sweep collector. Our aim
1819 in this section is not to develop a full-fledged verified versions of these collectors,
1820 but rather show that we can model their correctness specifications by extending the
1821 specifications and the proofs that we have used in the stop-the-world mark-and-sweep
1822 collector. As we will show, we can extend the abstract GC correctness specification
1823 from Section 3 to cover these collectors as well.

1824

1825 8.1 Incremental mark and sweep GC

1826

1827 An incremental mark and sweep GC, as the name suggests performs the GC work
1828 in *slices*. During each slice, the GC performs a part of the mark or sweep work and
1829 these slices are interleaved with the mutator actions, i.e., the execution of the OCaml
1830 program. The main advantage of an incremental GC is that it can reduce the pause
1831 times of the application. In a stop-the-world GC, the GC actions are performed in a
1832 single shot, which means that the application is paused for the entire duration of the
1833 GC. With an incremental GC, the program is paused only for the duration of a slice.

1834 OCaml uses an incremental mark and sweep collector for the major heap. OCaml
1835 uses snapshot-at-the-beginning variant [21] of the incremental GC, which ensures that
1836 the objects that are reachable at the beginning of the cycle are reachable at the end of
1837 the cycle as well. This is achieved by using a deletion write barrier, which, on a field
1838 update, marks the old value at that field. As the result, the old value, if unmarked,
1839 gets marked before it is overwritten by the new value. This ensures that the old value
1840 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 ??) from Section 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 `added_edges` \cap `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 $\omega(h_0)$ holds, and let r be the set of *roots*, which are pointers to objects into h_0 . Let $G(h_0).V$ be the vertex-set and $G(h_0).E$ be the edge-set of $G(h_0)$. Let h_1 be the heap after a full cycle of the incremental mark and sweep GC. Let $G(h_1).V$ be the vertex-set and $G(h_1).E$ be the edge-set of $G(h_1)$. Let $RG(h_0,r)$ be the reachable sub-graph residing in $G(h_0)$.

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

1. $\omega(h_1)$
2. (a) $G(h_1).V = RG(h_0,r).V \cup \text{new_allocs}$
 (b) $G(h_1).E = (RG(h_0,r).E \cup \text{added_edges}) \setminus \text{deleted_edges}$
3. $(\forall x. x \in G(h_1).V) \implies \text{data}(x,h_0) = \text{data}(x,h_1)$

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

1887 In addition to the correctness specification, we also observe that the incremental
1888 mark and sweep GC can use the correctness proofs of the stop-the-world mark and
1889 sweep GC. The intuition is that each slice of the incremental mark and sweep GC
1890 is a sequence of atomic mark and sweep steps. The atomic mark and sweep steps
1891 are exactly the definitions in `mark_body` (in Figure 10) and `sweep_body` (in Figure 11),
1892 respectively. Given that the `new_allocs`, `added_edges` and `deleted_edges` are not modified
1893 during a GC slice, we conjecture that the proofs of `mark_body` and `sweep_body` can be
1894 reused for the incremental mark and sweep GC without any significant change. As a
1895 result, we anticipate that the incremental mark and sweep GC will reuse significant
1896 parts of the proofs of the stop-the-world mark and sweep GC.

1897

1898 8.2 Copying collector

1899

1900 In the context of a copying collector, the heap is divided into two disjoint spaces, namely
1901 `from_space` and `to_space`. The goal of the copying collector is to copy all the reachable
1902 and only the reachable objects from the `from_space` to the `to_space`. Earlier works [22]
1903 have proved the correctness of a standalone copying collector (albeit with a different
1904 object layout than OCaml), and we simply adapt their correctness specifications in
1905 our framework.

1906

1907 Let h_0 be the state of `from_space` of the heap on which the collector operates, such
1908 that $\omega(h_0)$ holds, and let r be the set of *roots*, which are pointers to objects into h_0 .
1909 Let $G(h_0).V$ be the vertex-set and $G(h_0).E$ be the edge-set of $G(h_0)$. Let h_1 be the state
1910 of the `to_space` of the heap after the GC terminates and let $G(h_1).V$ be the vertex-set
1911 and $G(h_1).E$ be the edge-set of $G(h_1)$. Note that in a copying collector, the `to_space` is
1912 empty at the beginning of the GC.

1913 Let $RG(h_0, r)$ be the reachable sub-graph residing in $G(h_0)$. Let f be a one-to-one
1914 mapping function which maps objects in the `from_space` to objects in the `to_space`. For
1915 a set s , we define $f_S(s)$ as the set obtained by applying f to every element in s . For a
1916 graph g , we define $f_G(g)$ as the graph obtained by applying f to every object in vertex
1917 set $g.V$ and to the components of the pair in the edge set $g.E$.

1918

1919 **Definition 7 (Correctness of copying collector).** *The copying collector is said to*
1920 *be correct if the following conditions hold:*

1921

1. $\omega(h_1)$

1922

2. $G(h_1) = f_G(RG(h_0, r))$, where $G(h_1).V = f_S(RG(h_0).V)$ and

1923

$$(\forall x_1, y_1. x_1 \in G(h_1).V \wedge y_1 \in G(h_1).V \wedge (x_1, y_1) \in G(h_1).E) \iff$$

1924

$$(\exists x_0, y_0. x_0 \in G(h_0).V \wedge y_0 \in G(h_0).V \wedge x_1 = f(x_0) \wedge y_1 = f(y_0) \wedge$$

1925

$$(x_0, y_0) \in G(h_0).E)$$

1926

3. $(\forall x_1. x_1 \in G(h_1).V \iff$

1927

$$(\exists x_0. x_0 \in G(h_0).V \wedge x_1 = f(x_0) \wedge \mathit{data}(x_0, h_0) = \mathit{data}(x_1, h_1)))$$

1928

1929 Notice that this definition is almost identical to the correctness specification of the
1930 mark-and-sweep GC defined in Section 3. In particular, it uses the object reachability
1931 predicate to define the reachable subgraph RG in the `from_space` heap, which needs to
1932

be preserved by the heap in the `to_space`, along with well-formedness of the `to_space` heap and preserving the data values. 1933
1934

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. 1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946

9 Related Work 1947 1948 1949

Previous works on verifying garbage collectors (GC) 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. [9] 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 Section 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. 1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964

Gammie et al.[3] 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.[4] used Coq to verify a concurrent mark-and-sweep collector expressed in a compiler intermediate representation, without generating executable code. Xu et al. [28] propose a model checking framework for gaining confidence in collector correctness but do not present a concrete framework. McCreight et al.[29] 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. 1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978

1979 Another notable prior work is the verification of a generational copying collector for
1980 CakeML [10], which employs HOL4. Similar to our work, they also employ a layered
1981 approach from abstract algorithmic levels down to assembly closely integrated with
1982 CakeML’s compiler. Wang et al. [30] develop a mathematical and spatial graph library
1983 in Coq, verifying a generational copying collector as part of their framework. They
1984 verify the correctness of their copying garbage collector by proving the abstract graph
1985 isomorphism established by the copying function. Their basic object representation is
1986 similar to ours, where an object consists of a header followed by a variable number of
1987 fields. Their 400-line implementation was sufficient to certify a garbage collector for
1988 the CertiCoq project. Compared to their object layout support, we need to support
1989 additional complexities associated with the OCaml language including no-scan, closure
1990 and infix object types.

1991 Lin et al. [31] present the verification of a Yuasa incremental garbage collector in a
1992 Hoare-style PCC framework, the Stack-based Certified Assembly Programming (SCAP)
1993 system [32] with embedded separation-logic [33] primitives. Their verification in Coq
1994 ensures that the collector always preserves the heap objects reachable by the mutator.
1995 Some of the specification constructs follow their previous work [34] on verifying a stop-
1996 the-world mark-sweep collector. However, their collectors assume that every object
1997 has exactly two fields. Our support for different types of OCaml objects with variable-
1998 length fields poses additional verification challenge. The extraction to portable C code
1999 is a unique feature of our work, not available in any of the prior works.

2000 As part of our development, we have verified the correctness of a DFS algorithm on
2001 graphs. Verification of graph algorithms is a well-studied area [12–18]. Several works
2002 also verify complex specifications for graph algorithms. Lammich et al.[35] provide a
2003 framework for verifying depth-first search algorithms in Isabelle. Gueneau et al.[36]
2004 use a program logic to verify both correctness and complexity of an incremental cycle
2005 detection algorithm. Chen et al.[37] verify Tarjan’s strongly connected components
2006 algorithm using different verification frameworks, encountering challenges with reason-
2007 ing about reachability over arbitrary-length paths. We believe that the prior work on
2008 graph algorithms will pave way for reasoning about the correctness of complex GC
2009 algorithms. Our approach of separating out graph-theoretic correctness from the GC
2010 implementation will be suitable to integrate such complex graph algorithms into GC
2011 verification.

2012

2013 10 Limitations, Conclusion and Future Work

2014

2015 In this work, we have successfully developed a correct-by-construction GC for OCaml in
2016 a proof-oriented manner using F*/Low* proof-oriented programming language. We have
2017 extracted C code from the Low* program and have integrated the verified GC with the
2018 OCaml. The OCaml compiler with the verified GC is able to run standard benchmark
2019 programs as well as larger programs from the OCaml ecosystem. The experimental
2020 results demonstrate that our verified GC is pragmatic. We believe that our layered
2021 verification strategy should enable us to get close to the generational, incremental
2022 mark-and-sweep GC used by OCaml. We have described how our specifications can be
2023 extended to cover the correctness of these algorithms.

2024

In our current work, we have the limitation that the size of the mark stack should be equal to the size of the heap (Section 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 [38] and its successor Pulse [39], which we believe can not only simplify the proofs, but also allow us to reason about the correctness of concurrent GCs.

References

- [1] Mo, M.Y.: Chrome in-the-wild bug analysis: CVE-2021-37975. https://securitylab.github.com/research/in_the_wild_chrome_cve.2021.37975/ (2021)
- [2] 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 (2017). IEEE
- [3] Gammie, P., Hosking, A.L., Engelhardt, K.: Relaxing safely: verified on-the-fly garbage collection for x86-tso. *ACM SIGPLAN Notices* **50**(6), 99–109 (2015)
- [4] Zakowski, Y., Cachera, D., Demange, D., Petri, G., Pichardie, D., Jagannathan, S., Vitek, J.: Verifying a concurrent garbage collector using a rely-guarantee methodology. In: Interactive Theorem Proving: 8th International Conference, ITP 2017, Brasília, Brazil, September 26–29, 2017, Proceedings 8, pp. 496–513 (2017). Springer
- [5] Swamy, N., Hrițcu, 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)

- 2071 [6] Martínez, G., Ahman, D., Dumitrescu, V., Giannarakis, N., Hawblitzel, C., Hritcu,
2072 C., Narasimhamurthy, M., Paraskevopoulou, Z., Pit-Claudel, C., Protzenko, J., *et*
2073 *al.*: Meta-f: Proof automation with smt, tactics, and metaprograms. In: Program-
2074 ming Languages and Systems: 28th European Symposium on Programming, ESOP
2075 2019, Held as Part of the European Joint Conferences on Theory and Practice of
2076 Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings,
2077 pp. 30–59 (2019). Springer International Publishing Cham
- 2078 [7] Protzenko, J., Zinzindohoué, J.-K., Rastogi, A., Ramananandro, T., Wang, P.,
2079 Zanella-Béguelin, S., Delignat-Lavaud, A., Hritcu, C., Bhargavan, K., Fournet, C.,
2080 Swamy, N.: Verified Low-Level Programming Embedded in F* (2018)
- 2082 [8] Jones, R., Hosking, A., Moss, E.: The Garbage Collection Handbook: the Art of
2083 Automatic Memory Management. CRC Press, Boca Raton, FL (2016)
- 2084 [9] Hawblitzel, C., Petrank, E.: Automated verification of practical garbage collectors.
2085 ACM SIGPLAN Notices **44**(1), 441–453 (2009)
- 2086 [10] Ericsson, A.S., Myreen, M.O., Pohjola, J.Å.: A Verified Generational Garbage
2087 Collector for CakeML. Journal of Automated Reasoning **63**(2), 463–488 (2019)
- 2088 [11] McCreight, A.E.: The mechanized verification of garbage collector implementations.
2089 Yale University (2008)
- 2090 [12] Russinoff, D.M.: A mechanically verified incremental garbage collector. Formal
2091 Aspects of Computing **6**(4), 359–390 (1994)
- 2092 [13] Gonthier, G.: Verifying the safety of a practical concurrent garbage collector.
2093 In: Computer Aided Verification: 8th International Conference, CAV’96 New
2094 Brunswick, NJ, USA, July 31–August 3, 1996 Proceedings 8, pp. 462–465 (1996).
2095 Springer Berlin Heidelberg
- 2096 [14] Havelund, K.: Mechanical verification of a garbage collector. In: Parallel and
2097 Distributed Processing: 11th IPPS/SPDP’99 Workshops Held in Conjunction with
2098 the 13th International Parallel Processing Symposium and 10th Symposium on
2099 Parallel and Distributed Processing San Juan, Puerto Rico, USA, April 12–16,
2100 1999 Proceedings 13, pp. 1258–1283 (1999). Springer
- 2101 [15] Jackson, P.B.: Verifying a garbage collection algorithm. In: Theorem Proving
2102 in Higher Order Logics: 11th International Conference, TPHOLS’ 98 Canberra,
2103 Australia September 27–October 1, 1998 Proceedings 11, pp. 225–244 (1998).
2104 Springer
- 2105 [16] Goguen, H., Brooksby, R., Burstall, R.: An abstract formulation of memory
2106 management. December (1998)
- 2107 [17] Burdy, L.: B vs. coq to prove a garbage collector. In: the 14th International
2108
2109
2110
2111
2112
2113
2114
2115
2116

Conference on Theorem Proving in Higher Order Logics: Supplemental Proceedings (2001)	2117 2118 2119
[18] Coupet-Grimal, S., Nouvet, C.: Formal verification of an incremental garbage collector. <i>Journal of Logic and Computation</i> 13 (6), 815–833 (2003)	2120 2121 2122
[19] Sivaramakrishnan, K., Dolan, S., White, L., Jaffer, S., Kelly, T., Sahoo, A., Parimala, S., Dhiman, A., Madhavapeddy, A.: Retrofitting parallelism onto ocaml. <i>Proceedings of the ACM on Programming Languages</i> 4 (ICFP), 1–30 (2020)	2123 2124 2125 2126
[20] Madhavapeddy, A., Minsky, Y.: <i>Real World OCaml: Functional Programming for the Masses</i> . Cambridge University Press, Cambridge (2022)	2127 2128 2129
[21] Yuasa, T.: Real-time garbage collection on general-purpose machines. <i>Journal of Systems and Software</i> 11 (3), 181–198 (1990)	2130 2131
[22] Myreen, M.O.: Reusable verification of a copying collector. In: <i>International Conference on Verified Software: Theories, Tools, and Experiments</i> , pp. 142–156 (2010). Springer	2132 2133 2134 2135
[23] Bhargavan, K., Bond, B., Delignat-Lavaud, A., Fournet, C., Hawblitzel, C., Hritcu, C., Ishtiaq, S., Kohlweiss, M., Leino, R., Lorch, J., <i>et al.</i> : Everest: Towards a verified, drop-in replacement of https. In: <i>2nd Summit on Advances in Programming Languages (SNAPL 2017)</i> (2017). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik	2136 2137 2138 2139 2140 2141
[24] 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: <i>28th USENIX Security Symposium (USENIX Security 19)</i> , pp. 1465–1482 (2019)	2142 2143 2144 2145 2146
[25] Reitz, A., Fromherz, A., Protzenko, J.: Starmalloc: Verifying a modern, hardened memory allocator. <i>Proc. ACM Program. Lang.</i> 8 (OOPSLA2) (2024) https://doi.org/10.1145/3689773	2147 2148 2149 2150
[26] Gouy, I.: The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/	2151 2152
[27] Boehm, H.-J., Weiser, M.: Garbage collection in an uncooperative environment. <i>Software: Practice and Experience</i> 18 (9), 807–820 (1988)	2153 2154 2155
[28] Xu, B., Moss, E., Blackburn, S.M.: Towards a model checking framework for a new collector framework. In: <i>Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes</i> , pp. 128–139 (2022)	2156 2157 2158 2159
[29] McCreight, A., Shao, Z., Lin, C., Li, L.: A general framework for certifying garbage collectors and their mutators. In: <i>Proceedings of the 28th ACM SIGPLAN</i>	2160 2161 2162

2163 Conference on Programming Language Design and Implementation, pp. 468–479
2164 (2007)
2165
2166 [30] Wang, S., Cao, Q., Mohan, A., Hobor, A.: Certifying Graph-Manipulating C
2167 Programs via Localizations within Data Structures. *Proc. ACM Program. Lang.*
2168 **3**(OOPSLA) (2019) <https://doi.org/10.1145/3360597>
2169
2170 [31] Lin, C., Chen, Y., Hua, B.: Verification of an incremental garbage collector in
2171 hoare-style logic. *Int. J. Softw. Informatics* **3**(1), 67–88 (2009)
2172
2173 [32] Feng, X., Shao, Z., Vaynberg, A., Xiang, S., Ni, Z.: Modular verification of
2174 assembly code with stack-based control abstractions. *ACM SIGPLAN Notices*
2175 **41**(6), 401–414 (2006)
2176
2177 [33] Logic, S.: A logic for shared mutable data structures. John C. Reynolds. *LICS*
2178 (2002)
2179
2180 [34] Lin, C.-X., Chen, Y.-Y., Li, L., Hua, B.: Garbage collector verification for proof-
2181 carrying code. *Journal of Computer Science and Technology* **22**(3), 426–437
2182 (2007)
2183
2184 [35] Lammich, P., Neumann, R.: A framework for verifying depth-first search algorithms.
2185 In: *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pp.
2186 137–146 (2015)
2187
2188 [36] Guéneau, A., Jourdan, J.-H., Charguéraud, A., Pottier, F.: Formal proof and
2189 analysis of an incremental cycle detection algorithm. In: *Interactive Theorem*
2190 *Proving* (2019). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik
2191
2192 [37] Chen, R., Cohen, C., Lévy, J.-J., Merz, S., Théry, L.: Formal proofs of tarjan’s
2193 strongly connected components algorithm in why3, coq and isabelle. In: *ITP 2019-*
2194 *10th International Conference on Interactive Theorem Proving*, vol. 141, pp. 13–1
2195 (2019). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik
2196
2197 [38] Fromherz, A., Rastogi, A., Swamy, N., Gibson, S., Martínez, G., Merigoux, D.,
2198 Ramananandro, T.: Steel: proof-oriented programming in a dependently typed
2199 concurrent separation logic. *Proc. ACM Program. Lang.* **5**(ICFP) (2021) <https://doi.org/10.1145/3473590>
2200
2201 [39] F* team: Pulse: Proof-oriented Programming in Concurrent Separation Logic.
2202 <https://fstar-lang.org/tutorial/book/pulse/pulse.html> Accessed 2024-12-04
2203
2204
2205
2206
2207
2208