

A Mechanically Verified Garbage Collector for OCaml

Sheera Shamsu
IIT Madras
Chennai, India
sheera.shms@gmail.com

Dipesh Kafle
NIT Trichy
Thiruchirappally, India
dipesh.kaphle111@gmail.com

Dhruv Maroo
IIT Madras
Chennai, India
dhruvsmaroo@gmail.com

Kartik Nagar
IIT Madras
Chennai, India
nagark@cse.iitm.ac.in

Karthikeyan Bhargavan
Cryspen
France
karthik.bhargavan@gmail.com

KC Sivaramakrishnan
IIT Madras and Tarides
Chennai, India
kcsrk@cse.iitm.ac.in

Abstract

The GC is a critical component of the OCaml runtime system, and its correctness is essential for the safety of OCaml programs. Therefore, 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 mechanized in F^* and its low-level subset Low^* . We use the KaRaMel compiler to compile Low^* to C, and the verified GC is integrated with the OCaml runtime. Our GC is evaluated against off-the-shelf OCaml GC and Boehm-Demers-Weiser conservative GC, and the experimental results show that verified GC is pragmatic.

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

1 Introduction

The GC used in OCaml version 4 is generational and features two heap generations: the minor and major heaps. The minor heap employs copying collection, while the major heap utilizes an incremental mark and sweep GC to automatically reclaim memory. Both the minor and the major GC is implemented in C. Given that the memory safety of OCaml depends on the correctness of the GC, we wondered whether we could formally verify the correctness of the OCaml GC. Some previous works [3, 10] have verified the correctness of abstract GC models, which risk missing out on subtle bugs due to the air gap between the abstract model and the GC implementation. The practical garbage collectors, such as [2, 5], utilize a different memory model. Their approach produces a verified garbage collector in assembly code, while we require a verified C code to integrate with the OCaml runtime. Our goal in this work is to develop a verified GC for OCaml, through a proof-oriented approach, such that

executable code compatible with the OCaml compiler can be extracted directly from the verification artifact.

Rather than undertake the challenging task of verifying the full functional correctness of the existing OCaml GC in C, we have chosen to develop the verified GC from scratch in a proof-oriented language. We start from a feature complete GC that can run OCaml programs, but one which lacks the optimizations and features of the existing OCaml GC, and aim to incrementally enhance this GC with more features. To support this evolution, we have structured our verification approach such that the core correctness conditions for the GC need minimal changes throughout the enhancements. 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 [6], 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.
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 [8].

2 Background

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.

Every object in OCaml has a word-sized header in which meta-data about the object is stored [7]. 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, grey, and black), with the rest of the most significant bits representing the object size in words. Every field of the object is also word-sized, which ensures that the pointers to objects are always word-aligned. Immediate values such as integers and booleans are also word-sized. Immediate values are encoded with their least significant bit (LSB) to be 1, with the rest of the bits encoding the value of the data type.

The tag bits in the header of an OCaml object is used to determine whether the fields of the objects may contain pointers. In particular, for objects with tag greater or equal to `No_scan_tag` (251), the fields are all opaque bytes, and are not scanned by the GC. If an object's tag is less than `No_scan_tag` (251), then the fields of the objects may be pointers. Among these, apart from `Closure_tag` (247) and `Infix_tag` (249) objects, the GC scans each field of the object to determine if it is a pointer or an immediate value and takes appropriate action.

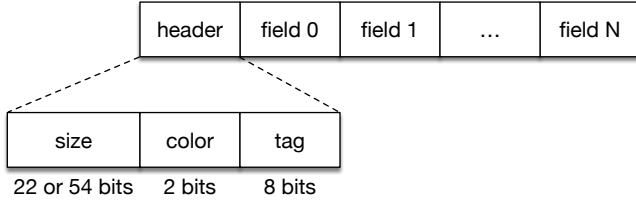


Figure 1. Layout of an OCaml object. The header and each field of an OCaml object occupy one word.

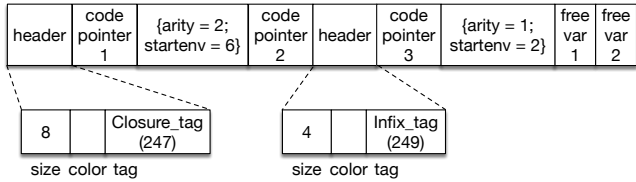


Figure 2. The layout of a mutually-recursive closure object layout with arities 2 and 1 and environment size 2.

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 2 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.

3 Abstract GC Correctness

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 $E = \{(x, y) \mid x \in V \wedge y \in V \wedge y \in \text{ptrs}(x, h)\}$. 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,

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

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

```

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

```

Figure 3. Extract C code for the top-level stop-the-world mark-and-sweep GC function.

h_0 . Let h_1 be the heap after the GC terminates and let V be the vertex set of $G(h_1)$. Then, the GC is said to be correct if:

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

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

4 Towards a verified OCaml GC

As mentioned in Section 1, the verified garbage collection (GC) code written in Low* can be extracted to C. Figure 3 shows the top-level GC function for a 64-bit platform, with functions renamed for clarity.

The heap, hp , is a contiguous byte buffer of fixed size $heap_size$. The GC takes inputs: an array of roots r (length r_len), a mark stack array st with a stack top pointer tp , a sweep pointer sw , and a free list pointer fp . The pointers tp , sw , and fp are singleton buffers, while the free list is a linked list of blue-colored free objects.

We assume zero-length objects are not stored in the heap, meaning each object has at least one field to store the next pointer for the free list. Initially, st is empty, tp points to the stack base, sw points to the start of the heap (hp), and fp points to the first blue object.

The GC begins by calling `darken_roots` to mark all roots in r as grey, pushing them onto the mark stack st and updating the stack top pointer tp .

To keep things concise, we have excluded the full implementation code and instead focus on the essential elements of the mark and sweep functions that highlight the complexities of the garbage collection (GC) process. In Figure 4, the `mark_heap_body` function is repeatedly called by `mark_heap` until the mark stack is empty. It pops the top object from the stack and colors it black, then performs a field scan using `darken_wrapper` if the tag is less than `no_scan_tag`. `darken_wrapper` checks the object type: for infix objects, it locates the parent closure for scanning. For closure objects, it finds the field start address using closure information. For other object types, the field address follows the object's header. This information is passed to `darken`, which inspects each field pointer. If a field pointer is infix, the parent closure is found and added to the stack if white; otherwise, non-infix white objects are directly added. After the mark phase, the sweep phase frees white objects encountered by changing its header color to blue (Figure 5). If a new blue object is adjacent to the free list pointer, they are coalesced; otherwise, the free list pointer is updated to point to the new blue object.

5 Verification Framework and Correctness Proofs

The verification approach identifies garbage collection (GC) operations that manipulate the heap and proves that these manipulations do not affect the heap's well-formedness and the structure of the reachable object sub-graph. Once we have a verified depth-first search (DFS), we can show that our mark function correctly colors all and only the reachable objects black by establishing the equivalence between the outputs of the mark function and the DFS. Our method separates the operations of DFS on graphs and the mark function on the heap into two distinct layers, as shown in Figure 6.

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 and the heap manipulations 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.

5.1 End-to-end GC correctness

The end-to-end GC correctness is shown in Figure 7. The theorem is formulated using Layer 2 primitives, with a specification that accepts a well-formed heap h_init , a root set $roots$, a mark stack st containing grey roots, and a free-list pointer fp . Pre-conditions relating the arguments to the heap are omitted.

The ensures clause outlines the post-conditions after executing mark and sweep, consisting of five properties. First, the final heap h_sweep must be well-formed, aligning with

```

void mark_heap_body (uint8_t *hp, uint64_t *st,
                    uint64_t *tp) {
    tp[0U] = *tp - 1ULL;
    uint64_t x = st[(*tp)];
    uint64_t h_x = hd_address(x);
    colorHeader(hp, h_x, black);

    uint64_t wz = wsize_of_block(h_x, hp);
    uint64_t tg = tag_of_block(h_x, hp);
    if (tg < 251ULL)
        darken_wrapper (hp, st, tp, h_x, wz);
}

void darken_wrapper (uint8_t *hp, uint64_t *st,
                    uint64_t *tp,
                    uint64_t h_x, uint64_t wz) {

    if (tag_of_block(h_x, g) == 247ULL) {
        uint64_t x = f_address(h_x);
        uint64_t st_env = start_env_clos_info(hp, x);
        uint64_t st_env_one = st_env + 1ULL;
        darken(hp, st, tp, h_x, wz, st_env);

    } else if (tag_of_block(h_x, hp) ==
                (uint64_t)249U) {
        wz = *(uint64_t *)h_x >> 10;
        h_x = h_x - wz * 8;
        wz = *(uint64_t *)h_x >> 10;
        darken_wrapper(hp, st, tp, h_x, wz);
    } else
        darken(hp, st, tp, h_x, wz, 1ULL);
}

void darken_body (uint8_t *hp, uint64_t *st,
                  uint64_t *tp, uint64_t h_x,
                  uint64_t i) {
    uint64_t succ_index = h_x + i * 8ULL;
    uint64_t succ = load64_le(hp + succ_index);
    if (isPointer(succ_index, hp)) {
        uint64_t h_addr_succ = hd_address(succ);
        if (tag_of_block(h_addr_succ, hp) == 249ULL) {
            uint64_t parent_hdr =
                parent_closure_of_infix_object(hp, h_x, i);
            darken_helper(hp, st, st_len, parent_hdr);
        } else
            darken_helper(hp, st, st_len, h_addr_succ);
    }
}

```

Figure 4. Extracted C code for mark body function.

the first property in the abstract garbage collection (GC) correctness definition (Definition 5). Second, the graph formed from `h_sweep` should only include objects reachable from the initial heap’s roots, ensuring all reachable objects are retained. Third, the GC must preserve the edges between these reachable objects, corresponding to the second property in Definition 5.

```

void sweep_body(uint8_t *hp, uint64_t *sw,
                uint64_t *fp) {
    uint64_t sw_val = *sw;
    uint64_t hdr_sw_val = hd_address(sw_val);
    uint64_t c = color_of_block(hdr_sw_val, hp);
    uint64_t wz = wsize_of_block(hdr_sw_val, hp);
    if (wz == 0) {
        return;
    }
    if (c == white || c == blue) {
        colorHeader(hp, hdr_sw_val, blue);
        f_list_ptr_val = *fp;
        uint64_t cur_wsize = wsize_of_block(
            hd_address(fp), hp);

        uint64_t next =
            fp + cur_wsize * 8 + 8;
        if (next == sw_val) {
            uint64_t next_wz =
                wsize_of_block(hdr_address(sw_val), hp);
            ((uint64_t *)fp)[-1] =
                makeHeader(cur_wsize + next_wz + 1,
                    blue, 0);
        } else {
            store64_le(hp + fp, sw_val);
            fp[0U] = sw_val;
        }
    } else
        colorHeader(g, hdr_sw_val, white);
}

```

Figure 5. Extracted C code for sweep body function.

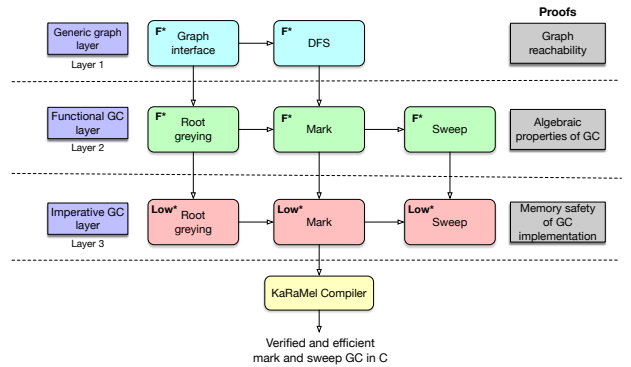


Figure 6. The layering approach for GC verification

The fourth property states that the resultant heap contains only white and blue objects, vital for maintaining consistency for future cycles. Lastly, the fifth property ensures that the fields of non-blue objects remain unchanged, a stronger assertion than the corresponding third property in Definition 5.

6 Evaluation

In this section, we evaluate the performance of the verified GC on a variety of benchmarks.

```

val end_to_end_correctness_theorem
  (* Initial heap *)
  (h_init:heap{well_formed_heap h_init})
  (* mark stack - contains grey objects *)
  (st : seq Usize.t
    {pre_conditions_on_stack h_init st})
  (* root set *)
  (roots : seq Usize.t
    {pre_conditions_on_root h_init roots})
  (* free list pointer *)
  (fp : hp_addr
    {pre_conditions_on_free_list h_init fp})

: Lemma
( requires
  (* Pre-conditions elided for brevity.
    + The mark stack [st] contains all the [roots].
    + All the grey objects in the heap are in the
      mark stack [st].
  *) )
( ensures
  (* heap after mark *)
  let h_mark = mark h_init st in
  (* heap after sweep *)
  let h_sweep = fst (sweep h_mark mword fp) in
  (* graph at init *)
  let g_init = graph_from_heap h_init in
  (* graph after sweep *)
  let g_sweep = graph_from_heap h_sweep in
  (* GC preserves well-formedness of the heap *)
  (* 1 *) well_formed_heap h_sweep ∧

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

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

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

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

```

Figure 7. Overall correctness theorem for the mark and sweep GC

6.1 Integration with OCaml

Our goal in this work is to develop a practical verified GC for OCaml that can serve as a replacement for the unverified GC. We have successfully extracted the verified C code for the GC functionality from Low* using the KaRaMel compiler [?].

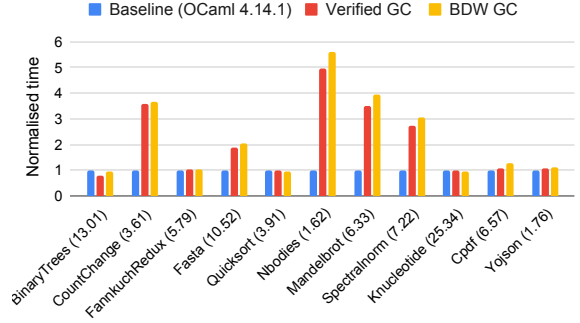


Figure 8. Normalized running time of different OCaml GCs. The numbers in the parenthesis next to the benchmark names are the running time in seconds for the baseline OCaml 4.14.1 GC.

We have integrated the extracted code into the OCaml 4.14.1 runtime system, replacing the existing GC in the bytecode runtime.

6.2 GC evaluation

We evaluate the performance of the verified GC on a variety of benchmark programs from the Computer Language Benchmarks Game [4] as well as larger programs – cpdf (an industrial-strength pdf processing tool). 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.

Figure 8 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 [1]. BDW GC is widely-known, pragmatic GC for uncooperative environments. BDW GC 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× slower than the baseline. The reason is, 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.

7 Future Plans

An extended version of our work has been published in the Journal of Automated Reasoning [9] and our artifacts are available in the repository at https://github.com/prismlab/verified_ocaml_gc/tree/main. We plan to develop a verified incremental mark-and-sweep GC that will build upon the components of our existing stop-the-world mark-and-sweep GC as future enhancements. Additionally, we aim to address the issue of mark stack overflow more effectively by incorporating more practical solutions.

References

- [1] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage collection in an uncooperative environment. *Software: Practice and Experience* 18, 9 (1988), 807–820.
- [2] Adam Sandberg Ericsson, Magnus O Myreen, and Johannes Åman Pohjola. 2019. A Verified Generational Garbage Collector for CakeML. *Journal of Automated Reasoning* 63, 2 (2019), 463–488.
- [3] Peter Gammie, Antony L Hosking, and Kai Engelhardt. 2015. Relaxing safely: verified on-the-fly garbage collection for x86-TSO. *ACM SIGPLAN Notices* 50, 6 (2015), 99–109.
- [4] Isaac Gouy. [n. d.]. The Computer Language Benchmarks Game. <https://benchmarksgame-team.pages.debian.net/benchmarksgame/>.
- [5] Chris Hawblitzel and Erez Petrank. 2009. Automated verification of practical garbage collectors. *ACM SIGPLAN Notices* 44, 1 (2009), 441–453.
- [6] Richard Jones, Antony Hosking, and Eliot Moss. 2016. *The garbage collection handbook: the art of automatic memory management*. CRC Press.
- [7] Anil Madhavapeddy and Yaron Minsky. 2022. *Real World OCaml: Functional Programming for the Masses*. Cambridge University Press, Cambridge.
- [8] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hrițcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified low-level programming embedded in F*. *Proc. ACM Program. Lang.* 1, ICFP, Article 17 (Aug. 2017), 29 pages. <https://doi.org/10.1145/3110261>
- [9] Sheera Shamsu, Dipesh Kafle, Dhruv Maroo, Kartik Nagar, Karthikeyan Bhargavan, and KC Sivaramakrishnan. 2025. A Mechanically Verified Garbage Collector for OCaml. *Journal of Automated Reasoning* 69, 2 (2025), 1–43.
- [10] Yannick Zakowski, David Cachera, Delphine Demange, Gustavo Petri, David Pichardie, Suresh Jagannathan, and Jan Vitek. 2017. 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. Springer, 496–513.