

Zoo: A framework for the verification of concurrent OCaml 5 programs using separation logic

Anonymous author

Anonymous affiliation

Anonymous author

Anonymous affiliation

Abstract

The release of OCAML 5, which introduced parallelism into the language, drove the need for safe and efficient concurrent data structures. New libraries like SATURN aim at addressing this need. From the perspective of formal verification, this is an opportunity to apply and further state-of-the-art techniques to provide stronger guarantees.

We present a framework for verifying fine-grained concurrent OCAML 5 algorithms. We followed a pragmatic approach, studying OCAML code written by concurrency expert to delimit a limited but sufficient fragment of the language to express these algorithms; the outcome is a dialect of OCAML that we call ZOOLANG. We formalized its semantics carefully via a deep embedding in the ROCQ proof assistant. We provide a tool to translate source OCAML programs into ZOOLANG syntax inside ROCQ, where they can be specified and verified using the IRIS concurrent separation logic.

We verified fine-grained concurrent algorithms, along with subsets of the OCAML standard library necessary to express them: the classic Treiber stack, and a use of reference-counting for file descriptors within the `Eio` library. This formalization work uncovered delicate questions of programming-language semantics, around physical equality for example. In the process, we also extended OCAML to more efficiently express certain concurrent programs.

2012 ACM Subject Classification Software and its engineering → General programming languages; Software and its engineering → Concurrent programming structures; Theory of computation → Program verification; Theory of computation → Separation logic

Keywords and phrases ROCQ, program verification, fine-grained concurrency, separation logic, OCaml

Digital Object Identifier 10.4230/LIPIcs.ITP.2025.23

1 Introduction

Designing concurrent algorithms, in particular fine-grained concurrent algorithms, is a notoriously difficult task. Similarly, the formal verification of such algorithms is also difficult. It typically involves finding and reasoning about non-trivial linearization points [21, 29, 52, 53, 11].

In recent years, concurrent separation logic [5] has enabled significant progress in this area. In particular, the development of IRIS [28], a state-of-the-art mechanized *higher-order* concurrent separation logic with *user-defined ghost state*, has nourished a rich and successful line of works [29, 52, 53, 11, 6, 27, 46, 36, 35, 17, 41, 39, 38], dealing with external [53] and future-dependent [29, 52, 11] linearization points, relaxed memory [36, 35, 17, 41] and automation [39, 38].

Most of these works [29, 52, 53, 6, 27, 46, 39, 38] and many others [19, 43, 51, 33] rely on HEAPLANG [49], the exemplar IRIS language. HEAPLANG is a concurrent, imperative, untyped, call-by-value functional language. To the best of our knowledge, it is currently the closest language to OCAML 5 in the IRIS ecosystem—we review the existing frameworks in Section 2. It has been extended to handle weak memory [36] and algebraic effects [18].



© Anonymous author(s);

licensed under Creative Commons License CC-BY 4.0

16th International Conference on Interactive Theorem Proving (ITP 2025).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:20

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Although HEAPLANG is theoretically expressive enough to represent OCAML programs, our experiments showed that it is fairly impractical when it comes to verifying large OCAML libraries. Indeed, it lacks basic abstractions such as algebraic data types (tuples, mutable and immutable records, variants) and mutually recursive functions. Verifying OCAML programs in HEAPLANG requires difficult translation choices and introduces various encodings, to the point that the relation between the source and verified programs can become difficult to maintain and reason about. It also has very few standard data structures that can be directly reused. This view, we believe, is shared by many people in the IRIS community. Our first motivation in this work is therefore to fill this gap by providing a more practical OCAML-like verification language: ZOOLANG. This language consists in a subset of OCAML 5 extended with atomic record fields and equipped with a formal semantics and a program logic based on IRIS. We were influenced by the PERENNIAL [8, 9, 10, 11] framework, which achieved similar goals for the GO language with a focus on crash-safety. As in PERENNIAL, we also provide a translator from OCAML to ZOOLANG: `ocaml2zoo`. We call the resulting framework ZOO.

Another, maybe less obvious, shortcoming of HEAPLANG is the soundness of its semantics with respect to OCAML, in other words how faithful it is to the original language. One ubiquitous—particularly in lock-free algorithms relying on low-level atomic primitives—and subtle point is *physical equality*. In Section 5, we show that (1) HEAPLANG’s semantics for physical equality is not compatible with OCAML and (2) OCAML’s informal semantics is actually too imprecise to verify basic concurrent algorithms. To remedy this, we propose a new formal semantics for physical equality and structural equality. We hope this work will influence the way these notions are specified in OCAML.

In summary, we claim the following contributions:

1. We present ZOOLANG, a convenient subset of OCAML 5 formalized in ROCQ (Sections 3 and 4). ZOOLANG comes with a program logic based on IRIS and supports proof automation through DIAFRAME [39, 38].
2. We provide a translator from OCAML to ZOOLANG: `ocaml2zoo` (Section 3), built for practical applications – it supports full projects using the `dune` build system.
3. We formalize physical equality (Section 5) and structural equality (Section 7) in a faithful way. The careful analysis of these notions suggests a new OCAML feature: *generative constructors*.
4. We extend OCAML with *atomic record fields* and *atomic arrays* to ease the development of fine-grained concurrent algorithms (Section 8).
5. We verify realistic use cases (Section 5) involving physical equality: (1) Treiber stack [7], (2) a thread-safe wrapper around a file descriptor using reference-counting from the `Eio` [34] library.

2 Related work

The idea of applying formal methods to verify OCAML programs is not new. Generally speaking, there are mainly two ways:

2.1 Non-automated verification

The verified program is translated, manually or in an automated way, into a representation living inside a proof assistant. The user has to write specifications and prove them.

The representation may be primitive, like Gallina for ROCQ. For pure programs, this is rather straightforward, *e.g.* in `hs-to-coq` [47]. For imperative programs, this is more

challenging. One solution is to use a monad, *e.g.* in `coq-of-ocaml` [14], but it does not support concurrency.

The representation may be embedded, meaning the semantics of the language is formalized in the proof assistant. This is the path taken by some recent works [12, 24, 8, 16] harnessing the power of separation logic. In particular, CFML [12] and OSIRIS [16] target OCAML. However, CFML does not support concurrency and is not based on IRIS. OSIRIS, still under development, is based on IRIS but does not support concurrency.

At the time of writing, HEAPLANG is thus the most appropriate tool to verify concurrent OCAML programs. We discussed limitations of HEAPLANG in the introduction, and ZOOLANG is our proposal to improve on this. (Conversely, one notable limitation of ZOOLANG today is its lack of support for OCAML’s relaxed memory model.)

2.2 Semi-automated verification

In semi-automated verification approaches, the verified program is annotated by the user to guide the verification tool: preconditions, postconditions, invariants, *etc.* Given this input, the verification tool generates proof obligations that are mostly automatically discharged. One may further distinguish two types of semi-automated systems: *foundational* and *non-foundational*.

In *non-foundational* automated verification, the tool and the external solvers it may rely on are part of the trusted computing base. It is the most common approach and has been widely applied in the literature [48, 40, 26, 20, 3, 22, 32, 44], including to OCAML by CAMELEER [42], which uses the GOSPEL specification language [13] and WHY3 [22].

In *foundational* automated verification, the proofs are checked by a proof assistant like ROCQ, meaning the automation does not have to be trusted. To our knowledge, it has been applied to C [45] and RUST [23].

ZOO is a non-automated verification framework—except for our use DIAFRAME for local automation of separation logic reasoning. We would be interested in moving towards more automation in the future.

3 Zoo in practice

In this section, we give an overview of our framework. We also provide a minimal example¹ demonstrating its use.

3.1 Language

The core of ZOO is ZOOLANG: a concurrent, imperative, untyped, functional programming language fully formalized in ROCQ. Its semantics has been designed to match OCAML’s.

ZOOLANG comes with a program logic based on IRIS: reasoning rules expressed in separation logic (including rules for the different constructs of the language) along with ROCQ tactics that integrate into the IRIS proof mode [31, 30]. In addition, it supports DIAFRAME [39, 38], enabling proof automation.

The ZOOLANG syntax is given in Figure 1², omitting mutually recursive toplevel functions that are treated specifically. Expressions include standard constructs like booleans, integers, anonymous functions (that may be recursive), applications, `let` bindings, sequence, unary

¹ Non-anonymous link

² More precisely, it is the syntax of the surface language, including ROCQ notations.

RocQ term	t	
constructor	C	
projection	$proj$	
record field	fld	
identifier	s, f	$\in \text{String}$
integer	n	$\in \mathbb{Z}$
boolean	b	$\in \mathbb{B}$
binder	x	$::= \langle \rangle \mid s$
unary operator	\oplus	$::= \sim \mid -$
binary operator	\otimes	$::= + \mid - \mid * \mid \text{'quot'} \mid \text{'rem'} \mid \text{'land'} \mid \text{'lor'} \mid \text{'lsl'} \mid \text{'lsr'}$ $\mid \leq \mid < \mid > \mid = \mid \neq \mid == \mid !=$ $\mid \text{and} \mid \text{or}$
expression	e	$::= t \mid s \mid \#n \mid \#b$ $\mid \text{fun: } x_1 \dots x_n \Rightarrow e \mid \text{rec: } f \ x_1 \dots x_n \Rightarrow e \mid e_1 \ e_2$ $\mid \text{let: } x := e_1 \text{ in } e_2 \mid e_1 \ ; \ ; \ e_2$ $\mid \text{let: } f \ x_1 \dots x_n := e_1 \text{ in } e_2 \mid \text{letrec: } f \ x_1 \dots x_n := e_1 \text{ in } e_2$ $\mid \text{let: 'C } x_1 \dots x_n := e_1 \text{ in } e_2 \mid \text{let: } x_1, \dots, x_n := e_1 \text{ in } e_2$ $\mid \oplus e \mid e_1 \otimes e_2$ $\mid \text{if: } e_0 \text{ then } e_1 \text{ (else } e_2 \text{)}^?$ $\mid \text{for: } x := e_1 \text{ to } e_2 \text{ begin } e_3 \text{ end}$ $\mid \S C \mid \text{'C } (e_1, \dots, e_n) \mid (e_1, \dots, e_n) \mid e.\langle proj \rangle$ $\mid [] \mid e_1 :: e_2$ $\mid \text{'C } \{e_1, \dots, e_n\} \mid \{e_1, \dots, e_n\} \mid e.\{fld\} \mid e_1 \leftarrow \{fld\} e_2$ $\mid \text{ref } e \mid !e \mid e_1 \leftarrow e_2$ $\mid \text{match: } e_0 \text{ with } br_1 \mid \dots \mid br_n \ (l_ \text{ as } s)^? \Rightarrow e \text{)}^? \text{ end}$ $\mid e.[fld] \mid \text{Xchg } e_1 \ e_2 \mid \text{CAS } e_1 \ e_2 \ e_3 \mid \text{FAA } e_1 \ e_2$ $\mid \text{Proph} \mid \text{Resolve } e_0 \ e_1 \ e_2$
branch	br	$::= C \ (x_1 \dots x_n)^? \text{ (as } s \text{)}^? \Rightarrow e$ $\mid [] \text{ (as } s \text{)}^? \Rightarrow e \mid x_1 :: x_2 \text{ (as } s \text{)}^? \Rightarrow e$
toplevel value	v	$::= t \mid \#n \mid \#b$ $\mid \text{fun: } x_1 \dots x_n \Rightarrow e \mid \text{rec: } f \ x_1 \dots x_n \Rightarrow e$ $\mid \S C \mid \text{'C } (v_1, \dots, v_n) \mid (v_1, \dots, v_n)$ $\mid [] \mid v_1 :: v_2$

■ **Figure 1** ZOOLANG syntax (omitting mutually recursive toplevel functions)

and binary operators, conditionals, **for** loops, tuples. In any expression, one can refer to a RocQ term representing a ZOOLANG value (of type **val**) using its RocQ identifier. ZOOLANG is deeply embedded: variables (bound by functions and **let**) are quoted, represented as strings.

Data constructors (immutable memory blocks) are supported through two constructs : $\S C$ represents a constant constructor (e.g. $\S \text{None}$), $\text{'C } (e_1, \dots, e_n)$ represents a non-constant constructor (e.g. $\text{'Some } (e)$). Unlike OCAML, ZOOLANG has projections of the form $e.\langle proj \rangle$ (e.g. $(x, y).\langle 1 \rangle$), that can be used to obtain a specific component of a tuple or data constructor. ZOOLANG supports shallow pattern matching (patterns cannot be nested) on data constructors with an optional fallback case.

Mutable memory blocks are constructed using either the untagged record syntax $\{e_1, \dots, e_n\}$ or the tagged record syntax $\text{'C } \{e_1, \dots, e_n\}$. Reading a record field can be performed using

```

type 'a t =
  'a list Atomic.t

let create () =
  Atomic.make []

let rec push t v =
  let old = Atomic.get t in
  let new_ = v :: old in
  if not @@ Atomic.compare_and_set t old new_ then (
    Domain.cpu_relax () ;
    push t v
  )

let rec pop t =
  match Atomic.get t with
  | [] ->
    None
  | v :: new_ as old ->
    if Atomic.compare_and_set t old new_ then (
      Some v
    ) else (
      Domain.cpu_relax () ;
      pop t
    )

```

■ **Figure 2** Implementation of a concurrent stack

141 $e.\{fld\}$ and writing to a record field using $e_1 \leftarrow \{fld\} e_2$. Pattern matching can also be used
 142 on mutable tagged blocks provided that cases do not bind anything—in other words, only
 143 the tag is examined, no memory access is performed. References are also supported through
 144 the usual constructs : `ref` e creates a reference, `!e` reads a reference and $e_1 \leftarrow e_2$ writes
 145 into a reference. The syntax seemingly does not include constructs for arrays but they are
 146 supported through the `Array` standard module (*e.g.* `array_make`).

147 Note that ZOOLANG follows OCAML in sometimes eschewing orthogonality to provide
 148 more compact memory representations: constructors are n -ary instead of taking a tuple as
 149 parameter, and the tagged record syntax is distinct from a constructor taking a mutable record
 150 as parameter. In each case the simplifying encoding would introduce an extra indirection in
 151 memory, which is absent from the ZOOLANG semantics. Performance-conscious experts care
 152 about these representation choices, and we care about faithfully modeling their programs.

153 Parallelism is mainly supported through the `Domain` standard module (*e.g.* `domain_spawn`).
 154 Special constructs (`Xchg`, `CAS`, `FAA`; see Section 4.4) are used to model atomic references.

155 The `Prop` and `Resolve` constructs model *prophecy variables* [29], see Section 4.5.

156 3.2 Translation from OCaml to ZooLang

157 While ZOOLANG lives in ROCQ, we want to verify OCAML programs. To connect them

we provide the tool `ocaml2zoo` to translate OCAML source files³ into ROCQ files containing ZOOLANG code. This tool can process entire `dune` projects, and support several libraries provided together or as dependencies of the project.

The supported OCAML fragment includes: tuples, variants, records (including inline records), shallow `match`, atomic record fields, unboxed types, toplevel mutually recursive functions.

Consider, for example, the OCAML implementation of a concurrent stack [7] in Figure 2. The `push` function is translated into:

```
Definition stack_push : val :=
  rec: "push" "t" "v" =>
    let: "old" := !"t" in
    let: "new_" := "v" :: "old" in
    if: ~ CAS "t".[contents] "old" "new_" then (
      domain_yield () ;;
      "push" "t" "v"
    ).
```

Gabriel{If we need more space we can move this code to Figure 2, as a second column on the right.}

3.3 Specifications and proofs

Once the translation to ZOOLANG is done, the user can write specifications and prove them in IRIS. For instance, the specification of the `stack_push` function could be:

```
Lemma stack_push_spec t  $\iota$  v :
  <<< stack_inv t  $\iota$ 
  |  $\forall \forall$  vs, stack_model t vs >>>
  stack_push t v @  $\uparrow \iota$ 
  <<< stack_model t (v :: vs)
  | RET (); True >>>.
Proof. ... Qed.
```

Here, we use a *logically atomic specification* [15], which has been proven [4] to be equivalent to *linearizability* [25] in sequentially consistent memory models.

Similarly to Hoare triples, the specification is formed of a precondition and a postcondition, represented in angular brackets. But each is split in two parts, a *public* or *atomic* condition, and a *private* condition. Following standard IRIS notations, the private conditions are on the outside (first line of the precondition, last line of the postcondition) and the atomic conditions are inside.

For this particular operation, the private postcondition is trivial. The private condition `stack_inv t` is the stack invariant. Intuitively, it asserts that t is a valid concurrent stack. More precisely, it enforces a set of logical constraints—a concurrent protocol—that t must respect at all times.

The atomic pre- and post-conditions specify the linearization point of the operation: during the execution of `stack_push`, the abstract state of the stack held by `stack_model` is atomically updated from vs to $v :: vs$; in other words, v is atomically pushed at the top of the stack.

³ Actually, `ocaml2zoo` processes binary annotation files (`.cmt` files).

4 Zoo features

In this section, we review the salient features of ZOO, which we found lacking when we attempted to use HEAPLANG to verify real-world OCAML programs. We start with the most generic ones and then address those related to concurrency.

4.1 Algebraic data types

ZOO is an untyped language but, to write interesting programs, it is convenient to work with abstractions like algebraic data types. To simulate tuples, variants and records, we designed a machinery to define projections, constructors and record fields.

For example, one may define a list-like type with:

```
Notation "'Nil'" := (in_type "t" 0) (in custom zoo_tag).
Notation "'Cons'" := (in_type "t" 1) (in custom zoo_tag).
```

Users do not need to write this incantation directly, as they are generated by `ocaml2zoo` from the OCAML type declarations. Suffice it to say that it introduces the two tags in the `zoo_tag` custom entry, on which the notations for data constructors rely. The `in_type` term is needed to distinguish the tags of distinct data types; crucially, it cannot be simplified away by ROCQ, as this could lead to confusion during the reduction of expressions.

Given this incantation, one may directly use the tags `Nil` and `Cons` in data constructors using the corresponding ZOO LANG constructs:

```
Definition map : val :=
  rec: "map" "fn" "t" =>
    match: "t" with
    | Nil =>
      $Nil
    | Cons "x" "t" =>
      let: "y" := "fn" "x" in
      'Cons( "y", "map" "fn" "t" )
    end.
```

Similarly, one may define a record-like type with two mutable fields `f1` and `f2`:

```
Notation "'f1'" := (in_type "t" 0) (in custom zoo_field).
Notation "'f2'" := (in_type "t" 1) (in custom zoo_field).
```

```
Definition swap : val :=
  fun: "t" =>
    let: "f1" := "t".{f1} in
    "t" <-{f1} "t".{f2} ;;
    "t" <-{f2} "f1".
```

4.2 Mutually recursive functions

ZOO supports non-recursive (`fun: $x_1 \dots x_n \Rightarrow e$`) and recursive (`rec: $f \ x_1 \dots x_n \Rightarrow e$`) functions but only *oplevel* mutually recursive functions. It is non-trivial to properly handle mutual recursion: when applying a mutually recursive function, a naive approach would replace calls to sibling functions by their respective bodies, but this typically makes the

resulting expression unreadable. To prevent it, the mutually recursive functions have to know one another to preserve their names during β -reduction. We simulate this using some boilerplate that can be generated by `ocaml2zoo`. For instance, one may define two mutually recursive functions `f` and `g` as follows:

```

Definition f_g := (
  recs: "f" "x" => "g" "x"
  and:  "g" "x" => "f" "x"
)%zoo_recs.

(* boilerplate *)
Definition f := ValRecs 0 f_g.
Definition g := ValRecs 1 f_g.
Instance : AsValRecs' f 0 f_g [f;g]. Proof. done. Qed.
Instance : AsValRecs' g 1 f_g [f;g]. Proof. done. Qed.

```

4.3 Standard library

To save users from reinventing the wheel, we provide a standard library—more or less a subset of the OCAML standard library. Currently, it mainly includes standard data structures like: array (`Array`), resizable array (`Dynarray`), list (`List`), stack (`Stack`), queue (`Queue`), double-ended queue, mutex (`Mutex`), condition variable (`Condition`).

Each of these standard modules contains ZOO LANG functions and their verified specifications. These specifications are modular: they can be used to verify more complex data structures. As an evidence of this, lists [1] and arrays [2] have been successfully used in verification efforts based on ZOO.

4.4 Concurrent primitives

ZOO supports concurrent primitives both on atomic references (from `Atomic`) and atomic record fields (from `Atomic.Loc`⁴) according to the table below. The OCAML expressions listed in the left-hand column translate into the ZOO expressions in the right-hand column. Notice that an atomic location `[%atomic.loc e.f]` (of type `_ Atomic.Loc.t`) translates directly into `e.[f]`.

OCAML	Zoo
<code>Atomic.get e</code>	<code>!e</code>
<code>Atomic.set e₁ e₂</code>	<code>e₁ <- e₂</code>
<code>Atomic.exchange e₁ e₂</code>	<code>Xchg e₁. [contents] e₂</code>
<code>Atomic.compare_and_set e₁ e₂ e₃</code>	<code>CAS e₁. [contents] e₂ e₃</code>
<code>Atomic.fetch_and_add e₁ e₂</code>	<code>FAA e₁. [contents] e₂</code>
<code>Atomic.Loc.exchange [%atomic.loc e₁.f] e₂</code>	<code>Xchg e₁. [f] e₂</code>
<code>Atomic.Loc.compare_and_set [%atomic.loc e₁.f] e₂ e₃</code>	<code>CAS e₁. [f] e₂ e₃</code>
<code>Atomic.Loc.fetch_and_add [%atomic.loc e₁.f] e₂</code>	<code>FAA e₁. [f] e₂</code>

One important aspect of this translation is that atomic accesses (`Atomic.get` and `Atomic.set`) correspond to plain loads and stores. This is because we are working in a

⁴ The `Atomic.Loc` module is part of the PR that implements atomic record fields.

sequentially consistent memory model: there is no difference between atomic and non-atomic memory locations.

4.5 Prophecy variables

Lock-free algorithms exhibit complex behaviors. To tackle them, IRIS provides powerful mechanisms such as *prophecy variables* [29]. Essentially, prophecy variables can be used to predict the future of the program execution and reason about it. They are key to handle *future-dependent linearization points*: linearization points that may or may not occur at a given location in the code depending on a future observation.

ZOO supports prophecy variables through the `Proph` and `Resolve` expressions—as in HEAPLANG, the canonical IRIS language. In OCAML, these expressions correspond to `Zoo.proph` and `Zoo.resolve`, that are recognized by `ocaml2zoo`.

5 Physical equality

The notion of *physical equality* is ubiquitous in fine-grained concurrent algorithms. It appears not only in the semantics of the `==` operator, but also in the semantics of the `Atomic.compare_and_set` primitive, which atomically sets an atomic reference to a desired value if its current content is physically equal to an expected value. This primitive is commonly used to try committing an atomic operation in a retry loop, as in the `push` and `pop` functions of Figure 2.

In HEAPLANG, this primitive is provided but restricted. Indeed, its semantics is only defined if either the expected or the desired value fits in a single memory word in the HEAPLANG value representation: literals (booleans, integers and pointers⁵) and literal injections⁶; otherwise, the program is stuck. **Gabriel**[In practice, this restriction forces the programmer to introduce an indirection [50, 52] to physically compare complex values, *e.g.* lists.]**{I don't understand this well: in OCaml as well, cons-cells are represented by “an indirection” (a pointer to a block with the head and tail), what is the difference here? I think that you are making a subtle distinction because you think of the “value” of mutable blocks as their pointer (as in OCaml), but the “value” of immutable as something else (whereas OCaml programmers would typically think of the pointer as well). This should be explained, otherwise what you say here is fairly confusing.}** Furthermore, when the semantics is defined, values are compared using their ROCQ representations; physical equality boils down to ROCQ equality.

In OCAML, physical equality is more tricky and often considered dangerous. *Structural equality*, which we describe in Section 7, should be the preferred way of comparing values. However, structural equality is typically much slower than physical equality, as it basically compiles to only one assembly instruction. Also, the `Atomic.compare_and_set` requires the comparison to be atomic, which is the case for physical equality but not structural equality.

In particular, the semantics of physical equality is *non-deterministic*. **Gabriel****{Non-deterministic on high-level values, not on low-level values. Again, omitting this distinction here makes the story hard to understand and potentially confusing.}** To see why, consider the case of *immutable blocks* representing constructors and immutable records (as opposed to *mutable blocks* representing mutable records), *e.g.* `Some 0`. The physical comparison of two

⁵ HEAPLANG allows arbitrary pointer arithmetic and therefore inner pointers. This is forbidden in both OCAML and ZOOLANG, as any reachable value has to be compatible with the garbage collector.

⁶ HEAPLANG has no primitive notion of constructor, only pairs and injections (left and right).

```

type state =
  | Open of Unix.file_descr
  | Closing of (unit -> unit)

type t =
  { mutable ops: int [@atomic];
    mutable state: state [@atomic];
  }

let make fd =
  { ops= 0; state= Open fd }

let closed =
  Closing (fun () -> ())
let close t =
  match t.state with
  | Closing _ ->
    false
  | Open fd as prev ->
    let close () = Unix.close fd in
    let next = Closing close in
    if Atomic.Loc.compare_and_set [%atomic.loc t.state] prev next then (
      if t.ops == 0
      && Atomic.Loc.compare_and_set [%atomic.loc t.state] next closed
      then
        close () ;
      true
    ) else (
      false
    )

```

■ Figure 3 Rcfid.close function from Eio [34]

seemingly identical immutable blocks, according to the ROCQ representation (essentially a tag and a list of fields), may return `false`. Indeed, at runtime, a non-empty immutable block is represented by a pointer to a tagged memory block. In this case, physical equality is just pointer comparison. It is clear that two pointers being distinct does not imply the pointed memory blocks are. In other words, we cannot determine the result of physical comparison just by looking at the abstract values.

The question is then: what guarantees do we get when physical equality returns `true` and when it returns `false`? Unfortunately, the only guarantee that OCAML **Gabriel**[provides]{*documents*} is: if two values are physically equal, they are also structurally equal. This means we don't learn anything when two values are physically distinct. **Gabriel**{We provide more guarantees on mutable values. You are implicitly talking about immutable constructors here.}

In the following, we will explore both cases, looking at the optimizations that the compiler or the runtime system may perform. We will show that the aforementioned guarantee is arguably not sufficient to verify interesting concurrent programs and attempt to establish stronger guarantees.

5.1 When physical equality returns `true`

Let us go back to the concurrent stack of Figure 2 and more specifically the `push` function. To prove the atomic specification given in Section 3, we rely on the fact that, if `Atomic.compare_and_set` returns `true`, we actually observe the same list of values in the sense of ROCQ equality. However, assuming only structural equality as per OCAML’s specification of physical equality, this is not true. To see why, consider, *e.g.*, a stack of references (`'a ref`). *Gabriel*[As structural equality is indeed *structural*, it traverses the references, giving no information whatsoever about their *physical identity*. Hence, we cannot prove the specification.]*{I don’t understand these two sentences.}*

This conclusion might seem surprising and counterintuitive. Indeed, we know that physical equality essentially boils down to a comparison instruction, so we should be able to say more. Departing from OCAML’s imprecise specification, let us attempt to establish stronger guarantees. We assume the following classification of values: booleans, integers, mutable blocks (pointers), immutable blocks, functions.

The easy cases are mutable blocks and functions. Each of these two classes is disjoint from the others. We can reasonably assume that, when physical equality returns `true` and one of the compared values belongs to either of these classes, the two values are actually the same in ROCQ. As far as we are aware, there is no optimization that could break this.

Booleans, integers and empty immutable blocks are represented by immediate integers through an encoding. This encoding induces conflicts: two seemingly distinct values in ROCQ may have the same encoding. For example, the following tests all return `true` (`Obj.repr` is an unsafe primitive revealing the memory representation of a value):

Gabriel{I would rather show the tests with the existential GADT, which do not require the `Obj` module and are more arguably “valid OCaml”.}

```
let test1 = Obj.repr false == Obj.repr 0 (* true *)
let test2 = Obj.repr None == Obj.repr 0 (* true *)
let test3 = Obj.repr [] == Obj.repr 0 (* true *)
```

The semantics of unrestricted physical equality has to reflect these conflicts. In our experience, restricting compared values similarly to typing is quite burdensome; the specification of polymorphic data structures using physical equality has to be systematically restricted. In summary, when physical equality on immediate values returns `true`, it is guaranteed that they have the same encoding.

Finally, let us consider the case of non-empty immutable blocks. At runtime, they are represented by pointers to tagged memory blocks. At first approximation, it is tempting to say that physically equal immutable blocks really are the same in ROCQ. Alas, this is not true. To explain why, we have to recall that the OCAML compiler and the runtime system (*e.g.*, through hash-consing) may perform *sharing*: immutable blocks containing physically equal fields may be shared. For example, the following tests may return `true`:

```
let test1 = Some 0 == Some 0 (* true *)
let test2 = [0;1] == [0;1] (* true *)
```

On its own, sharing is not a problem. However, coupled with representation conflicts, it can be surprising. Indeed, consider the `any` type defined as:

```
type any = Any : 'a -> any
```

The following tests may return `true`:

```

let test1 = Any false == Any 0 (* true *)
let test2 = Any None  == Any 0 (* true *)
let test3 = Any []    == Any 0 (* true *)

```

Now, going back to the `push` function of Figure 2, we have a problem. Given a stack of `any`, it is possible for the `Atomic.compare_and_set` to observe a current list (e.g., `[Any 0]`) physically equal to the expected list (e.g., `[Any false]`) while these are actually distinct in ROCQ. In short, the expected specification of Section 3 is incorrect. To fix it, we would need to reason *modulo physical equality*, which is non-standard and quite burdensome.

We believe this really is a shortcoming, at least from the verification perspective. Therefore, we propose to extend OCAML with *generative immutable blocks*⁷. These generative blocks are just like regular immutable blocks, except they cannot be shared. Hence, if physical equality on two generative blocks returns `true`, these blocks are necessarily equal in ROCQ. At user level, this notion is materialized by *generative constructors*. For instance, to verify the expected `push` specification, we can use a generative version of lists:

```

type 'a list =
  | Nil
  | Cons of 'a * 'a list [@generative]

```

5.2 When physical equality returns `false`

The informal OCAML specification does not give any guarantee when physical equality returns `false`. In most cases, including try loops, this is fine. However, in some specific cases, more information is needed.

Consider the `Rcfd` module from the `Eio` [34] library, an excerpt of which is given in Figure 3. Thomas Leonard, its author, suggested that we verify this real-life example because of its intricate logical state. However, we found out that it is also relevant regarding the semantics of physical equality. Essentially, it consists in wrapping a file descriptor in a thread-safe way using reference-counting.

6 Physical equality – alternative proposal

The notion of *physical equality* is ubiquitous in fine-grained concurrent algorithms. It appears not only in the semantics of the `==` operator, but also in the semantics of the `Atomic.compare_and_set` primitive, which atomically sets an atomic reference to a desired value if its current content is physically equal to an expected value. This primitive is commonly used to try committing an atomic operation in a retry loop, as in the `push` and `pop` functions of Figure 2.

At the same time this notion is difficult to specify correctly, and this can result in dangerous gaps between the programming language used to write code and the semantics used for its verification.

ZOOLANG has a grammar of values, and most operations are specified by defining how they compute with ZOOLANG values. Its definition may look as follows in Rocq (simplified slightly):

⁷ Non-anonymous link

```

Inductive literal :=
| Bool (b : bool)
| Int (n : nat)
| Loc (l : location)
| Proph (pid : prophet_id)
| Poison.

Inductive val :=
| Lit (lit : literal)
| Recs (i : nat) (recs : list (binder * binder * expr))
| Block (tag : nat) (vs : list val).

```

For example, the value `'Cons(42, $Nil)` is represented in Rocq as `Block 1 [Lit (Int 42), Block 0 []]`. Notice that immutable blocks are represented in Rocq using the `Block` constructor directly, and *not* as a location (`Loc`) allocated on the heap. We use locations only for *mutable* records. We would say that our representation of ZOOLANG values is *high-level*, as close to the surface syntax as reasonably possible. This distinction is important to make verification pleasant in practice, by reducing the number of locations and heap indirections that the programmer needs to work with during verification. A ZOOLANG tuple is directly a tuple, etc., and this design decision of using high-level values is important to the verification experience. **Basile**{Not only this, but assuming full ownership of immutable blocks would be incorrect.}

It is tempting to specify, as HEAPLANG does, that physical equality decides equality between high-level values. This specification makes sense for immediate values (integers, booleans), and for mutable records which are compared by location. But it is incorrect on immutable blocks, and HEAPLANG essentially does not specify its behavior on those values. But programmers use physical equality on immutable blocks in practice, as in our example of a Treiber stack of Figure 2.

Defining physical equality as equality of high-level values is problematic in two opposite ways:

1. Some distinct high-level values are physically equal in OCaml, for example `0` and `false`. Their type differ, but it possible to store them in an existential type where they can be compared for physical equality:

```

type any = Any : 'a -> any
let test1 = Any false == Any 0 (* true *)

```

This shows that even on immediate values, specifying physical equality as equality of high-level values is convenient but incorrect in practice.

2. A deeper problem is that some *definitionally equal* high-level values may be physically distinct. For example, if x is defined as the integer 42, then $(x :: []) == (42 :: [])$ may or may not hold, depending on the OCaml implementation being used. But one can prove that both arguments are definitionally equal in Rocq, so physical equality cannot be modeled by a Rocq function of type `val -> val -> bool`.

385 7 Structural equality

386 8 OCaml extensions for fine-grained concurrent programming

387 Over the course of this work, we studied efficient fine-grained concurrent OCAML programs
 388 written by experts. This revealed various limitations of OCAML in these domains, that
 389 those experts would work around using unsafe casts, often at the cost of both readability
 390 and memory-safety; and also some mismatches between their mental model of the semantics
 391 of OCAML and the mental model used by the OCAML compiler authors. We worked on
 392 improving OCAML itself to reduce these work-arounds or semantic mismatches.

393 8.1 Atomic record fields

394 8.1.1 Before

395 OCAML 5 offers a type `'a Atomic.t` of atomic references exposing sequentially-consistent
 396 atomic operations. Data races on non-atomic mutable locations has a much weaker semantics
 397 and is generally considered a programming error. For example, the Michael-Scott concurrent
 398 queue [37] relies on a linked list structure that could be defined as follows:

```
type 'a node =  
  | Nil  
  | Cons of { value : 'a; next : 'a node Atomic.t }
```

399 Performance-minded concurrency experts dislike this representation, because `'a Atomic.t`
 400 introduces an indirection in memory: it is represented as a pointer to a block containing the
 401 value of type `'a`. Instead, they use something like the following:

```
type 'a node =  
  | Nil  
  | Cons of { mutable next: 'a node; value: 'a }  
  
let as_atomic : 'a node -> 'a node Atomic.t option = function  
  | Nil -> None  
  | (Next _) as record -> Some (Obj.magic record : 'a node Atomic.t)
```

402 Notice that the `next` field of the `Cons` constructor has been moved first in the type
 403 declaration. Because the OCAML compiler respects field-declaration order in data layout, a
 404 value `Cons { next; value }` has a similar low-level representation to a reference (atomic
 405 or not) pointing at `next`, with an extra argument. The code uses `Obj.magic` to unsafely cast
 406 this value to an atomic reference, which appears to work as intended.

407 `Obj.magic` is a shunned unsafe cast (the OCAML equivalent of `unsafe` or `unsafePerformIO`).
 408 It is very difficult to be confident about its usage given that it may typically violate
 409 assumptions made by the OCAML compiler and optimizer. In the example above, casting
 410 a two-fields record into a one-argument atomic reference may or may not be sound—but
 411 it gives measurable performance improvements on concurrent queue benchmarks. (TODO:
 412 benchmark to quantify the improvement.)

413 It is possible to statically forbid passing `Nil` to `as_atomic` to avoid error handling,
 414 by turning `'a node` into a GADT indexed over it a type-level representation of its head
 415 constructor. Examples of this pattern can be found in the `Kcas` library by Vesa Karvonen.

416 It is difficult to write correctly and use, in particular as unsafe casts can sometimes hide
 417 type-errors in the intended static discipline.

418 Note that this unsafe approach only works for the first field of a record, so it is not
 419 applicable to records that hold several atomic fields, such as the toplevel record storing
 420 atomic `front` and `back` pointers for the concurrent queue.

421 8.1.2 Atomic fields proposal

422 We proposed a design for atomic record fields as an OCAML language change proposal:
 423 RFC #39⁸. Declaring a record field atomic simply requires an `[@atomic]` attribute—and
 424 could eventually become a proper keyword of the language.

```
(* re-implementation of atomic references *)
type 'a atomic_ref = { mutable contents : 'a [@atomic]; }

(* concurrent linked list *)
type 'a node =
| Nil
| Cons of { value: 'a; mutable next : 'a node [@atomic]; }

(* bounded SPSC circular buffer *)
type 'a bag = {
  data : 'a Atomic.t array;
  mutable front: int [@atomic];
  mutable back: int [@atomic];
}
```

425 The design difficulty is to express atomic operations on atomic record fields. For example,
 426 if `buf` has type `'a bag` above, then one naturally expects the existing notation `buf.front` to
 427 perform an atomic read and `buf.front <- n` to perform an atomic write. But how would
 428 one express exchange, compare-and-set and fetch-and-add? We would like to avoid adding a
 429 new primitive language construct for each atomic operation.

430 Our proposed implementation⁹ introduces a built-in type `'a Atomic.Loc.t` for an atomic
 431 location that holds an element of type `'a`, with a syntax extension `[%atomic.loc <expr>.<field>]`
 432 to construct such locations. Atomic primitives operate on values of type `'a Atomic.Loc.t`,
 433 and they are exposed as functions of the module `Atomic.Loc`.

434 For example, the standard library exposes

```
val Atomic.Loc.fetch_and_add : int Atomic.Loc.t -> int -> int
```

435 and users can write:

```
let preincrement_front (buf : 'a bag) : int =
  Atomic.Loc.fetch_and_add [%atomic.loc buf.front] 1
```

436 where `[%atomic.loc buf.front]` has type `int Atomic.Loc.t`. Internally, a value of type
 437 `'a Atomic.Loc.t` can be represented as a pair of a record and an integer offset for the
 438 desired field, and the `atomic.loc` construction builds this pair in a well-typed manner.

⁸ Non-anonymous link

⁹ Non-anonymous link

When a primitive of the `Atomic.Loc` module is applied to an `atomic.loc` expression, the compiler can optimize away the construction of the pair—but it would happen if there was an abstraction barrier between the construction and its use.

Note: the type `'a Atomic.t` of atomic references exposes a function

```
val Atomic.make_contended : 'a -> 'a Atomic.t
```

that ensures that the returned atomic value is allocated with enough alignment and padding to sit alone on its cache line, to avoid performance issues caused by false sharing. Currently there is no such support for padding of atomic record fields (we are planning to work on this if the support for atomic fields gets merged in standard OCAML), so the less-compact atomic references remain preferable in certain scenarios.

8.2 Atomic arrays

On top of our atomic record fields, we have implemented support for atomic arrays, another facility commonly requested by authors of efficient concurrent programs. Our previous example of a concurrent bag of type `'a bag` used a backing array of type `'a Atomic.t array`, which contains more indirections than may be desirable, as each array element is a pointer to a block containing the value of type `'a`, instead of storing the value of type `'a` directly in the array.

Our implementation of atomic arrays¹⁰ builds on top of the type `'a Atomic.Loc.t` we described in the previous section, and it relies on two new low-level primitives provided by the compiler:

```
val Atomic_array.index : 'a array -> int -> 'a Atomic.Loc.t
val Atomic_array.unsafe_index : 'a array -> int -> 'a Atomic.Loc.t
```

The function `index` takes an array and an integer index within the array, and returns an atomic location into the corresponding element after performing a bound check. `unsafe_index` omits the boundcheck—additional performance at the cost of memory-safety—and allows to express the atomic counterpart of the unsafe operations `Array.unsafe_get` and `Array.unsafe_set`. The atomic primitives of the module `Atomic.Loc` can then be used on these indices; our implementation implements a library module on top of these primitives to provide a higher-level layer to the user, with direct array operations such as:

```
val Atomic_array.exchange : 'a Atomic_array.t -> int -> 'a -> 'a
val Atomic_array.unsafe_exchange : 'a Atomic_array.t -> int -> 'a -> 'a
```

9 Conclusion and future work

The development of ZOO is still ongoing. While it is not yet available on `opam`, it can be installed and used in other ROCQ projects. We provide a minimal example demonstrating its use.

ZOO supports a limited fragment of OCAML that is sufficient for most of our needs. Its main weakness so far is its memory model, which is sequentially consistent as opposed to the relaxed OCAML 5 memory model. It also lacks exceptions and algebraic effects, that we plan to introduce in the future.

¹⁰Non-anonymous link

Another interesting direction would be to combine ZOO with semi-automated techniques. Similarly to WHY3, the simple parts of the verification effort would be done in a semi-automated way, while the most difficult parts would be conducted in ROCQ.

References

- 1 Clément Allain, Basile Clément, Alexandre Moine, and Gabriel Scherer. Snapshottable stores. *Proc. ACM Program. Lang.*, 8(ICFP):338–369, 2024. doi:10.1145/3674637.
- 2 Clément Allain, Vesa Karvonen, and Carine Morel. Saturn: a library of verified concurrent data structures for OCaml 5. In *OCaml Workshop 2024 - ICFP 2024*, Milan, Italy, September 2024. Armaël Guéneau and Sonja Heinze. URL: <https://inria.hal.science/hal-04681703>.
- 3 Vytautas Astrauskas, Aurel Bilý, Jonás Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. The prusti project: Formal verification for rust. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, volume 13260 of *Lecture Notes in Computer Science*, pages 88–108. Springer, 2022. doi:10.1007/978-3-031-06773-0_5.
- 4 Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. Theorems for free from separation logic specifications. *Proc. ACM Program. Lang.*, 5(ICFP):1–29, 2021. doi:10.1145/3473586.
- 5 Stephen Brookes and Peter W. O’Hearn. Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65, 2016. doi:10.1145/2984450.2984457.
- 6 Quentin Carbonneaux, Noam Zilberstein, Christoph Klee, Peter W. O’Hearn, and Francesco Zappa Nardelli. Applying formal verification to microkernel IPC at meta. In Andrei Popescu and Steve Zdancewic, editors, *CPP ’22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, pages 116–129. ACM, 2022. doi:10.1145/3497775.3503681.
- 7 Thomas J. Watson IBM Research Center and R.K. Treiber. *Systems Programming: Coping with Parallelism*. Research Report RJ. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986. URL: <https://books.google.fr/books?id=YQg3HAAACAAJ>.
- 8 Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nikolai Zeldovich. Verifying concurrent, crash-safe systems with perennial. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 243–258. ACM, 2019. doi:10.1145/3341301.3359632.
- 9 Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nikolai Zeldovich. Gojournal: a verified, concurrent, crash-safe journaling system. In Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 423–439. USENIX Association, 2021. URL: <https://www.usenix.org/conference/osdi21/presentation/chajed>.
- 10 Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nikolai Zeldovich. Verifying the daisyngfs concurrent and crash-safe file system with sequential reasoning. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 447–463. USENIX Association, 2022. URL: <https://www.usenix.org/conference/osdi22/presentation/chajed>.
- 11 Yun-Sheng Chang, Ralf Jung, Upamanyu Sharma, Joseph Tassarotti, M. Frans Kaashoek, and Nikolai Zeldovich. Verifying vmvcc, a high-performance transaction library using multi-version concurrency control. In Roxana Geambasu and Ed Nightingale, editors, *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, pages 871–886. USENIX Association, 2023. URL: <https://www.usenix.org/conference/osdi23/presentation/chang>.

- 523 12 Arthur Charguéraud. *A Modern Eye on Separation Logic for Sequential Programs. (Un*
524 *nouveau regard sur la Logique de Séparation pour les programmes séquentiels)*. 2023. URL:
525 <https://tel.archives-ouvertes.fr/tel-04076725>.
- 526 13 Arthur Charguéraud, Jean-Christophe Filliâtre, Cláudio Lourenço, and Mário Pereira.
527 GOSPEL - providing ocaml with a formal specification language. In Maurice H. ter
528 Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods - The Next 30*
529 *Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*,
530 volume 11800 of *Lecture Notes in Computer Science*, pages 484–501. Springer, 2019. doi:
531 10.1007/978-3-030-30942-8_29.
- 532 14 Guillaume Claret. coq-of-ocaml. URL: <https://github.com/formal-land/coq-of-ocaml>.
- 533 15 Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for
534 time and data abstraction. In Richard E. Jones, editor, *ECOOP 2014 - Object-Oriented*
535 *Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014.*
536 *Proceedings*, volume 8586 of *Lecture Notes in Computer Science*, pages 207–231. Springer,
537 2014. doi:10.1007/978-3-662-44202-9_9.
- 538 16 Arnaud Daby-Seesaram, Jean-Marie Madiot, François Pottier, Remy Seassau, and Irene Yoon.
539 Osiris. URL: <https://gitlab.inria.fr/fpottier/osiris>.
- 540 17 Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Thai Nguyen, William Mansky, Jeehoon
541 Kang, and Derek Dreyer. Compass: strong and compositional library specifications in relaxed
542 memory separation logic. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM*
543 *SIGPLAN International Conference on Programming Language Design and Implementation,*
544 *San Diego, CA, USA, June 13 - 17, 2022*, pages 792–808. ACM, 2022. doi:10.1145/3519939.
545 3523451.
- 546 18 Paulo Emílio de Vilhena and François Pottier. A separation logic for effect handlers. *Proc.*
547 *ACM Program. Lang.*, 5(POPL):1–28, 2021. doi:10.1145/3434314.
- 548 19 Paulo Emílio de Vilhena, François Pottier, and Jacques-Henri Jourdan. Spy game: verifying
549 a local generic solver in iris. *Proc. ACM Program. Lang.*, 4(POPL):33:1–33:28, 2020. doi:
550 10.1145/3371101.
- 551 20 Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: A foundry for
552 the deductive verification of rust programs. In Adrián Riesco and Min Zhang, editors,
553 *Formal Methods and Software Engineering - 23rd International Conference on Formal*
554 *Engineering Methods, ICFEM 2022, Madrid, Spain, October 24-27, 2022, Proceedings*,
555 volume 13478 of *Lecture Notes in Computer Science*, pages 90–105. Springer, 2022. doi:
556 10.1007/978-3-031-17244-1_6.
- 557 21 Brijesh Dongol and John Derrick. Verifying linearisability: A comparative survey. *ACM*
558 *Comput. Surv.*, 48(2):19:1–19:43, 2015. doi:10.1145/2796550.
- 559 22 Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In
560 Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems -*
561 *22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint*
562 *Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24,*
563 *2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer,
564 2013. doi:10.1007/978-3-642-37036-6_8.
- 565 23 Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer.
566 Refinedrust: A type system for high-assurance verification of rust programs. *Proc. ACM*
567 *Program. Lang.*, 8(PLDI):1115–1139, 2024. doi:10.1145/3656422.
- 568 24 Léon Gondelman, Jonas Kastberg Hinrichsen, Mário Pereira, Amin Timany, and Lars Birkedal.
569 Verifying reliable network components in a distributed separation logic with dependent
570 separation protocols. *Proc. ACM Program. Lang.*, 7(ICFP):847–877, 2023. doi:10.1145/
571 3607859.
- 572 25 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent
573 objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

- 574 **26** Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and
 575 Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and java. In
 576 Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors,
 577 *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA,*
 578 *April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages
 579 41–55. Springer, 2011. doi:10.1007/978-3-642-20398-5_4.
- 580 **27** Jaehwang Jung, Janggun Lee, Jaemin Choi, Jaewoo Kim, Sunho Park, and Jeehoon Kang.
 581 Modular verification of safe memory reclamation in concurrent separation logic. *Proc. ACM*
 582 *Program. Lang.*, 7(OOPSLA2):828–856, 2023. doi:10.1145/3622827.
- 583 **28** Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek
 584 Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation
 585 logic. *J. Funct. Program.*, 28:e20, 2018. doi:10.1017/S0956796818000151.
- 586 **29** Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany,
 587 Derek Dreyer, and Bart Jacobs. The future is ours: prophecy variables in separation logic.
 588 *Proc. ACM Program. Lang.*, 4(POPL):45:1–45:32, 2020. doi:10.1145/3371113.
- 589 **30** Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser,
 590 Amin Timany, Arthur Charguéraud, and Derek Dreyer. Mosel: a general, extensible modal
 591 framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.*, 2(ICFP):77:1–
 592 77:30, 2018. doi:10.1145/3236772.
- 593 **31** Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order
 594 concurrent separation logic. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings*
 595 *of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL*
 596 *2017, Paris, France, January 18-20, 2017*, pages 205–217. ACM, 2017. doi:10.1145/3009837.
 597 3009855.
- 598 **32** Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou,
 599 Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear
 600 ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1):286–315, 2023. doi:10.1145/3586037.
- 601 **33** Anton Lorenzen, Daan Leijen, Wouter Swierstra, and Sam Lindley. The functional essence
 602 of imperative binary search trees. *Proc. ACM Program. Lang.*, 8(PLDI):518–542, 2024.
 603 doi:10.1145/3656398.
- 604 **34** Anil Madhavapeddy and Thomas Leonard. Eio. URL: [https://github.com/](https://github.com/ocaml-multicore/eio)
 605 [ocaml-multicore/eio](https://github.com/ocaml-multicore/eio).
- 606 **35** Glen Mével and Jacques-Henri Jourdan. Formal verification of a concurrent bounded queue in a
 607 weak memory model. *Proc. ACM Program. Lang.*, 5(ICFP):1–29, 2021. doi:10.1145/3473571.
- 608 **36** Glen Mével, Jacques-Henri Jourdan, and François Pottier. Cosmo: a concurrent separation
 609 logic for multicore ocaml. *Proc. ACM Program. Lang.*, 4(ICFP):96:1–96:29, 2020. doi:
 610 10.1145/3408978.
- 611 **37** Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking
 612 concurrent queue algorithms. In James E. Burns and Yoram Moses, editors, *Proceedings of*
 613 *the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia,*
 614 *Pennsylvania, USA, May 23-26, 1996*, pages 267–275. ACM, 1996. doi:10.1145/248052.
 615 248106.
- 616 **38** Ike Mulder and Robbert Krebbers. Proof automation for linearizability in separation logic.
 617 *Proc. ACM Program. Lang.*, 7(OOPSLA1):462–491, 2023. doi:10.1145/3586043.
- 618 **39** Ike Mulder, Robbert Krebbers, and Herman Geuvers. Diaframe: automated verification
 619 of fine-grained concurrent programs in iris. In Ranjit Jhala and Isil Dillig, editors, *PLDI*
 620 *'22: 43rd ACM SIGPLAN International Conference on Programming Language Design and*
 621 *Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 809–824. ACM, 2022.
 622 doi:10.1145/3519939.3523432.
- 623 **40** Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure
 624 for permission-based reasoning. In Alexander Pretschner, Doron Peled, and Thomas
 625 Hutzelmann, editors, *Dependable Software Systems Engineering*, volume 50 of *NATO Science*

- for *Peace and Security Series - D: Information and Communication Security*, pages 104–125. IOS Press, 2017. doi:10.3233/978-1-61499-810-5-104.
- 41 Sunho Park, Jaewoo Kim, Ike Mulder, Jaehwang Jung, Janggun Lee, Robbert Krebbers, and Jeehoon Kang. A proof recipe for linearizability in relaxed memory separation logic. *Proc. ACM Program. Lang.*, 8(PLDI):175–198, 2024. doi:10.1145/3656384.
- 42 Mário Pereira and António Ravara. Cameleer: A deductive verification tool for ocaml. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 677–689. Springer, 2021. doi:10.1007/978-3-030-81688-9_31.
- 43 François Pottier, Armaël Guéneau, Jacques-Henri Jourdan, and Glen Mével. Thunks and debits in separation logic with time credits. *Proc. ACM Program. Lang.*, 8(POPL):1482–1508, 2024. doi:10.1145/3632892.
- 44 Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. CN: verifying systems C code with separation-logic refinement types. *Proc. ACM Program. Lang.*, 7(POPL):1–32, 2023. doi:10.1145/3571194.
- 45 Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. Refinedc: automating the foundational verification of C code with refined ownership types. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 158–174. ACM, 2021. doi:10.1145/3453483.3454036.
- 46 Thomas Somers and Robbert Krebbers. Verified lock-free session channels with linking. *Proc. ACM Program. Lang.*, 8(OOPSLA2):588–617, 2024. doi:10.1145/3689732.
- 47 Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Total haskell is reasonable coq. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 14–27. ACM, 2018. doi:10.1145/3167092.
- 48 Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4):402–451, 2013. doi:10.1017/S0956796813000142.
- 49 Iris Development Team. The coq mechanization of iris. URL: <https://gitlab.mpi-sws.org/iris/iris/>.
- 50 Iris Development Team. Iris examples. URL: <https://gitlab.mpi-sws.org/iris/examples/>.
- 51 Amin Timany, Armaël Guéneau, and Lars Birkedal. The logical essence of well-bracketed control flow. *Proc. ACM Program. Lang.*, 8(POPL):575–603, 2024. doi:10.1145/3632862.
- 52 Simon Friis Vindum and Lars Birkedal. Contextual refinement of the michael-scott queue (proof pearl). In Catalin Hritcu and Andrei Popescu, editors, *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 76–90. ACM, 2021. doi:10.1145/3437992.3439930.
- 53 Simon Friis Vindum, Dan Frumin, and Lars Birkedal. Mechanized verification of a fine-grained concurrent queue from meta’s folly library. In Andrei Popescu and Steve Zdancewic, editors, *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, pages 100–115. ACM, 2022. doi:10.1145/3497775.3503689.