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 [31] 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. Following a pragmatic approach, we support a limited but sufficient fragment of the language whose semantics has been carefully formalized to faithfully express such algorithms. Source programs are translated to a deeply-embedded language living inside Rocq where they can be specified and verified using the Iris [28] concurrent separation logic.

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases Rocq, program verification, separation logic

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, 47, 38, 37, 17, 42, 40, 39], dealing with external [53] and future-dependent [29, 52, 11] linearization points, relaxed memory [38, 37, 17, 42] and automation [40, 39].

Most of these works [29, 52, 53, 6, 27, 47, 40, 39] and many others [19, 44, 51, 35] rely on HeapLang [50], the canonical 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 [38] and algebraic effects [18].

Although HeapLang is theoretically expressive enough to represent OCaml programs, our experience 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. 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 make 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 [40, 39].
2. We provide a translator from OCaml to ZooLang: `ocaml2zoo` (Section 3).
3. We formalize physical equality (Section 5) and structural equality (Section 6) 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 7).
5. We analyze realistic use cases (Section 5) involving physical equality: (1) Treiber stack [7], (2) a thread-safe wrapper around a file descriptor based on reference-counting.

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 Semi-automated verification

The verified program is annotated by the user to guide the verification tool: preconditions, postconditions, invariants, *etc.* Given this input, the 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 [49, 41, 26, 20, 3, 22, 34, 45], including to OCaml by Cameleer [43], 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 [46] and Rust [23].

2.2 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` [48]. 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.

3 Zoo in practice

$$
\begin{array}{llll}
\text{identifier} & s, f & \in & \text{String} \\
\text{integer} & n & \in & \mathbb{Z} \\
\text{boolean} & b & \in & \mathbb{B} \\
\text{binder} & x & ::= & \texttt{<>} \mid s \\
\text{unary operator} & \oplus & ::= & \texttt{\textasciitilde} \mid \texttt{-} \\
\text{binary operator} & \otimes & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{`quot`} \mid \texttt{`rem`} \mid \texttt{`land`} \mid \texttt{`lor`} \mid \texttt{`lsl`} \mid \texttt{`lsr`} \\
 & & \mid & \texttt{<=} \mid \texttt{<} \mid \texttt{>=} \mid \texttt{>} \mid \texttt{=} \mid \neq \mid \texttt{==} \mid \texttt{!=} \\
 & & \mid & \texttt{and} \mid \texttt{or} \\
\text{expression} & e & ::= & t \mid s \mid \texttt{\#}n \mid \texttt{\#}b \\
 & & \mid & \texttt{fun:}\ x_1 \ldots x_n\ \texttt{=>}\ e \mid \texttt{rec:}\ f\ x_1 \ldots x_n\ \texttt{=>}\ e \\
 & & \mid & \texttt{let:}\ x\ \texttt{:=}\ e_1\ \texttt{in}\ e_2 \mid e_1\ \texttt{;;}\ e_2 \\
 & & \mid & \texttt{let:}\ f\ x_1 \ldots x_n\ \texttt{:=}\ e_1\ \texttt{in}\ e_2 \mid \texttt{letrec:}\ f\ x_1 \ldots x_n\ \texttt{:=}\ e_1\ \texttt{in}\ e_2 \\
 & & \mid & \texttt{let:}\ \text{‘}C\ x_1 \ldots x_n\ \texttt{:=}\ e_1\ \texttt{in}\ e_2 \mid \texttt{let:}\ x_1\texttt{,}\ \ldots\texttt{,}x_n\ \texttt{:=}\ e_1\ \texttt{in}\ e_2 \\
 & & \mid & \oplus e \mid e_1 \otimes e_2 \\
 & & \mid & \texttt{if:}\ e_0\ \texttt{then}\ e_1\ (\texttt{else}\ e_2)^? \\
 & & \mid & \texttt{for:}\ x\ \texttt{:=}\ e_1\ \texttt{to}\ e_2\ \texttt{begin}\ e_3\ \texttt{end} \\
 & & \mid & \S C \mid \text{‘}C\ \texttt{(}e_1\texttt{,}\ \ldots\texttt{,}e_n\texttt{)} \mid \texttt{(}e_1\texttt{,}\ \ldots\texttt{,}e_n\texttt{)} \mid e\texttt{.<}\mathit{proj}\texttt{>} \\
 & & \mid & \texttt{[]} \mid e_1\ \texttt{::}\ e_2 \\
 & & \mid & \text{‘}C\ \texttt{\{}e_1\texttt{,}\ \ldots\texttt{,}e_n\texttt{\}} \mid \texttt{\{}e_1\texttt{,}\ \ldots\texttt{,}e_n\texttt{\}} \mid e\texttt{.\{}\mathit{fld}\texttt{\}} \mid e_1\ \texttt{<-\{}\mathit{fld}\texttt{\}}\ e_2 \\
 & & \mid & \texttt{ref}\ e \mid \texttt{!}e \mid e_1\ \texttt{<-}\ e_2 \\
 & & \mid & \texttt{match:}\ e_0\ \texttt{with}\ br_1 \texttt{|} \ldots \texttt{|}\ br_n\ (\texttt{|\_}\ (\texttt{as}\ s)^?\ \texttt{=>}\ e)^?\ \texttt{end} \\
 & & \mid & e\texttt{.[}\mathit{fld}\texttt{]} \mid \texttt{Xchg}\ e_1\ e_2 \mid \texttt{CAS}\ e_1\ e_2\ e_3 \mid \texttt{FAA}\ e_1\ e_2 \\
 & & \mid & \texttt{Proph} \mid \texttt{Resolve}\ e_0\ e_1\ e_2 \\
 & & \mid & \texttt{Reveal}\ e \\
\text{branch} & br & ::= & C\ (x_1 \ldots x_n)^?\ (\texttt{as}\ s)^?\ \texttt{=>}\ e \\
 & & \mid & \texttt{[]}\ (\texttt{as}\ s)^?\ \texttt{=>}\ e \mid x_1\ \texttt{::}\ x_2\ (\texttt{as}\ s)^?\ \texttt{=>}\ e \\
\text{toplevel value} & v & ::= & t \mid \texttt{\#}n \mid \texttt{\#}b \\
 & & \mid & \texttt{fun:}\ x_1 \ldots x_n\ \texttt{=>}\ e \mid \texttt{rec:}\ f\ x_1 \ldots x_n\ \texttt{=>}\ e \\
 & & \mid & \S C \mid \text{‘}C\ \texttt{(}v_1\texttt{,}\ \ldots\texttt{,}v_n\texttt{)} \mid \texttt{(}v_1\texttt{,}\ \ldots\texttt{,}v_n\texttt{)} \\
 & & \mid & \texttt{[]} \mid v_1\ \texttt{::}\ v_2
\end{array}
$$

Figure 1 ZooLang syntax (omitting mutually recursive toplevel functions)

In this section, we give an overview of the framework. We also provide a minimal example[1] demonstrating its use.

[1] `https://github.com/clef-men/zoo-demo`

3.0.0.1 Language.

The core of Zoo is ZooLang: an untyped, ML-like, imperative, concurrent programming language that is 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 [33, 32]. In addition, it supports Diaframe [40], enabling proof automation.

The ZooLang syntax is given in Figure 1[2], omitting mutually recursive toplevel functions that are treated specifically. Expressions include standard constructs like booleans, integers, anonymous functions (that may be recursive), `let` bindings, sequence, unary 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 a deeply embedded language: variables (bound by functions and `let`) are quoted, represented as strings.

Data constructors (immutable memory blocks) are supported through two constructs : §C represents a constant constructor (*e.g.* §None), ‘C (e_1, ... ,e_n) represents a non-constant constructor (*e.g.* ‘Some(e)). Unlike OCaml, ZooLang has projections of the form e.<*proj*> (*e.g.* (e_1,e_2).<1>), 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, ... ,e_n} or the tagged record syntax ‘C {e_1, ... ,e_n}. Reading a record field can be performed using e.{*fld*} and writing to a record field using e_1 <-{*fld*} e_2. Pattern matching can also be used on mutable tagged blocks provided that cases do not bind anything—in other words, only the tag is examined, no memory access is performed. References are also supported through the usual constructs : `ref` e creates a reference, !e reads a reference and e_1 <- e_2 writes into a reference. The syntax seemingly does not include constructs for arrays but they are supported through the `Array` standard module (*e.g.* `array_make`).

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

The `Proph` and `Resolve` constructs are used to model *prophecy variables* [29], as described in Section 4.5.

Finally, `Reveal` is a special source construct that we introduce to handle physical equality. We demystify it in Section 5.

3.0.0.2 Translation from OCaml to ZooLang.

While ZooLang lives in Rocq, we want to verify OCaml programs. To connect them, we provide a tool to automatically translate OCaml source files[3] into Rocq files containing ZooLang code: `ocaml2zoo`. This tool can process entire `dune` projects, including many libraries.

The supported OCaml fragment includes: shallow `match`, ADTs, records, inline records, atomic record fields, unboxed types, toplevel mutually recursive functions.

---

[2] More precisely, it is the syntax of the surface language, including many Rocq notations.
[3] Actually, `ocaml2zoo` processes binary annotation files (`.cmt` files).

As an example of what ocaml2zoo can generate, the push function from Section 1 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"
    ).
```

3.0.0.3 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 ι v :
  <<<
    stack_inv t ι
  | ∀∀ vs, stack_model t vs
  >>>
    stack_push t v @ ↑ι
  <<<
    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 two assertions inside curly brackets represent the precondition and postcondition for the caller. For this particular operation, the postcondition is trivial. The stack-inv t precondition 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 other two assertions inside angle brackets represent the *atomic precondition* and *atomic postcondition*. They 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.

4 Zoo features

In this section, we review the main features of Zoo, starting with the most generic ones and then addressing 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).
```

Given this incantation, one may directly use the tags `Nil` and `Cons` in data constructors using the corresponding ZooLang 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.
```

The meaning of this incantation is not really important, as such notations can be generated by `ocaml2zoo`. 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.

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:` $x_1 \dots x_n$ `=>` e) and recursive (`rec:` f $x_1 \dots x_n$ `=>` e) functions but only *toplevel* mutually recursive functions. Indeed, it is non-trivial to properly handle mutual recursion: when applying a mutually recursive function, a naive approach would replace the recursive 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 so as to replace by the names instead of the bodies. 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 ZooLang 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
`Atomic.get` e	`!`e
`Atomic.set` e_1 e_2	e_1 `<-` e_2
`Atomic.exchange` e_1 e_2	`Xchg` e_1`.[contents]` e_2
`Atomic.compare_and_set` e_1 e_2 e_3	`CAS` e_1`.[contents]` e_2 e_3
`Atomic.fetch_and_add` e_1 e_2	`FAA` e_1`.[contents]` e_2
`Atomic.Loc.exchange [%atomic.loc` e_1`.`f`]` e_2	`Xchg` e_1`.[`f`]` e_2
`Atomic.Loc.compare_and_set [%atomic.loc` e_1`.`f`]` e_2 e_3	`CAS` e_1`.[`f`]` e_2 e_3
`Atomic.Loc.fetch_and_add [%atomic.loc` e_1`.`f`]` e_2	`FAA` e_1`.[`f`]` e_2

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

4.5 Prophecy variables

Lockfree 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.

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

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

```
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

5.0.0.1 Example 1: physical equality.

Consider, for example, the OCaml implementation of a concurrent stack [7] in Figure 2. Essentially, it consists of an atomic reference to a list that is updated atomically using the `Atomic.compare_and_set` primitive. While this simple implementation—it is indeed one of the simplest lock-free algorithms—may seem easy to verify, it is actually more subtle than it looks.

Indeed, the semantics of `Atomic.compare_and_set` involves *physical equality*: if the content of the atomic reference is physically equal to the expected value, it is atomically updated to the new value. Comparing physical equality is tricky and can be dangerous—this is why *structural equality* is often preferred—because the programmer has few guarantees about the *physical identity* of a value. In particular, the physical identity of a list, or more generally of an inhabitant of an algebraic data type, is not really specified. The only

```
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 `Rcfd.close` function from `Eio` [36]

guarantee is: if two values are physically equal, they are also structurally equal. Apparently, we don't learn anything interesting when two values are physically distinct. Going back to our example, this is fortunately not an issue, since we always retry the operation when `Atomic.compare_and_set` returns `false`.

Looking at the standard runtime representation of OCaml values, this makes sense. The empty list is represented by a constant while a non-empty list is represented by pointer to a tagged memory block. Physical equality for non-empty lists is just pointer comparison. It is clear that two pointers being distinct does not imply the pointed memory blocks are.

From the viewpoint of formal verification, this means we have to carefully design the semantics of the language to be able to reason about physical equality and other subtleties of concurrent programs. Essentially, the conclusion we can draw is that the semantics of physical equality and therefore `Atomic.compare_and_set` is non-deterministic: we cannot determine the result of physical comparison just by looking at the abstract values.

5.0.0.2 Example 2: when physical identity matters.

Consider another example given in Figure 3: the `Rcfd.close`[5] function from the `Eio` [36] library. Essentially, it consists in protecting a file descriptor using reference counting. Similarly, it relies on atomically updating the `state` field using `Atomic.Loc.compare_and_set`[6]. However, there is a complication. Indeed, we claim that the correctness of `close` derives from the fact that the `Open` state does not change throughout the lifetime of the data structure; it can be replaced by a `Closing` state but never by another `Open`. In other words, we want to say that 1) this `Open` is *physically unique* and 2) `Atomic.Loc.compare_and_set` therefore detects whether the data structure has flipped into the `Closing` state. In fact, this kind of property appears frequently in lock-free algorithms; it also occurs in the `Kcas` [30] library[7].

Once again, this argument requires special care in the semantics of physical equality. In short, we have to reveal something about the physical identity of some abstract values. Yet, we cannot reveal too much—in particular, we cannot simply convert an abstract value to a concrete one (a memory location)—, since the OCaml compiler performs optimizations like sharing of immutable constants, and the semantics should remain compatible with adding other optimizations later on, such as forms of hash-consing.

In Zoo, a value is either a bool, an integer, a memory location, a function or an immutable block. To deal with physical equality in the semantics, we have to specify what guarantees we get when 1) physical comparison returns `true` and 2) when it returns `false`.

We assume that the program is semantically well typed, if not syntactically well typed, in the sense that compared values are loosely compatible: a boolean may be compared with another boolean or a location, an integer may be compared with another integer or a location, an immutable block may be compared with another immutable block or a location. This means we never physically compare, *e.g.*, a boolean and an integer, an integer and an immutable block. If we wanted to allow it, we would have to extend the semantics of physical comparison to account for conflicts in the memory representation of values.

For booleans, integers and memory locations, the semantics of physical equality is plain equality. Let us consider the case of abstract values (functions and immutable blocks).

If physical comparison returns `true`, the semantics of OCaml tells us that these values are structurally equal. This is very weak because structural equality for memory locations is not plain equality. In fact, assuming only that, the stack of Section 1 and many other concurrent algorithms relying on physical equality would be incorrect. Indeed, for *e.g.* a stack of references (`'a ref`), a successful `Atomic.compare_and_set` in `push` or `pop` would not be guaranteed to have seen the exact same list of references; the expected specification of Section 3 would not work. What we want and what we assume in our semantics is plain equality. Hopefully, this should be correct in practice, as we know physical equality is implemented as plain comparison.

If physical comparison returns `false`, the semantics of OCaml tells us essentially nothing: two immutable blocks may have distinct identities but same content. However, given this semantics, we cannot verify the `Rcfd` example of Section 1. To see why, consider the first `Atomic.compare_and_set` in the `close` function. If it fails, we expect to see a `Closing` state because we know there is only one `Open` state ever created, but we cannot prove it. To address it, we take another step back from OCaml's semantics by introducing the `Reveal` construct. When applied to an immutable memory block, `Reveal` yields the same block

[5] https://github.com/ocaml-multicore/eio/blob/main/lib_eio/unix/rcfd.ml
[6] Here, we make use of atomic record fields that were recently introduced in OCaml.
[7] https://github.com/ocaml-multicore/kcas/blob/main/doc/gkmz-with-read-only-cmp-ops.md

annotated with a logical identifier that can be interpreted as its abstract identity. The meaning of this identifier is: if physical comparison of two identified blocks returns `false`, the two identifiers are necessarily distinct. The underling assumption that we make here—which is hopefully also correct in the current implementation of OCaml—is that the compiler may introduce sharing but not unsharing.

The introduction of `Reveal` can be performed automatically by `ocaml2zoo` provided the user annotates the data constructor (*e.g.* `Open`) with the attribute `[@zoo.reveal]`. For `Rcfd.make`, it generates:

```
Definition rcfd_make : val :=
  fun: "fd" =>
    { #0, Reveal ‘Open( "fd" ) }.
```

Given this semantics and having revealed the `Open` block, we can verify the `close` function. Indeed, if the first `Atomic.compare_and_set` fails, we now know that the identifiers of the two blocks, if any, are distinct. As there is only one `Open` block whose identifier does not change, it cannot be the case that the current state is `Open`, hence it is `Closing` and we can conclude.

Structural equality is also supported. Due to space limitations, we do not describe it here but interested readers may refer to the Rocq mechanization[8].

6 Structural equality

7 Improving OCaml for concurrent lock-free programming

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

7.0.0.1 A reminder on OCaml attributes and extension points.

TODO

7.1 Atomic record fields

7.1.1 Before

OCaml 5 offers a type `'a Atomic.t` of atomic references exposing sequentially-consistent atomic operations. Data races on non-atomic mutable locations has a much weaker semantics and is generally considered a programming error. For example, a Michael-Scott concurrent queue uses a linked list structure that can be defined as follows:

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

[8] `https://github.com/clef-men/zoo/blob/main/theories/zoo/program_logic/structeq.v`

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

```
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)
```

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

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

It is possible to statically forbid passing `Nil` to `as_atomic` to avoid error handling, by turning `'a node` into a GADT indexed over it a type-level representation of its head constructor. Examples of this pattern can be found in the `Kcas` library by Vesa Karvonen. It is difficult to write correctly and use, in particular as unsafe casts can sometimes hide type-errors in the intended static discipline.

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

7.1.2 Proposal(s)

We proposed a design for atomic record fields as an OCaml language change proposal: RFC #39[9]. Declaring a record field atomic simply requires an `[@atomic]` attribute – and could eventually become a proper keyword of the language.

Gabriel{Clément proposes to remove the atomic.field part of the description and leave only atomic.loc, to shorten this section.**}**

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

[9] `https://github.com/ocaml/RFCs/pull/39`. Warning: this link is not anonymized.

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

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

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

We implemented two alternative options coming from RFC discussions, available in experimental variants of OCaml and proposed for inclusion in the upstream language and compiler:

1. Our first implementation[10] introduces a built-in type `'a Atomic.Loc.t` for an atomic location that holds an element of type `'a`, with a syntax extension `[%atomic.loc <expr>.<field>]` to construct such locations. Atomic primitives operate on values of type `'a Atomic.Loc.t`, and they are exposed as functions of the module `Atomic.Loc`.
   For example, the standard library exposes

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

   and users can write

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

   where `[%atomic.loc buf.front]` has type `int Atomic.Loc.t`.
   Internally, a value of type `'a Atomic.Loc.t` can be represented as a pair of a record and an integer offset for the desired field, and the `atomic.loc` construction builds this pair in a well-typed manner. 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.
2. Our second implementation[11] introduces a built-in type `('r, 'a) Atomic.Field.t` that denotes a field/index of type `'a` within a record of type `'r`, with a syntax extension `[%atomic.loc <field>]` to construct such field description, and atomic primitives in

[10] `https://github.com/ocaml/ocaml/pull/13404`. Warning: this link is not anonymized.
[11] `https://github.com/ocaml/ocaml/pull/13707`. Warning: this link is not anonymized.

a module `Atomic.Field`, that need both the record value of type `'r` and the field description.

For example, the standard library exposes

```
val Atomic.Field.fetch_and_add : 'a -> ('a, int) Atomic.Field.t -> int -> int
```

and users can write

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

where `[%atomic.field front]` has type `('a bag, int) Atomic.Loc.t`.

Internally, a value of type `('r, 'a) Atomic.Field.t` is just an integer offset locating the field within the record: in exchange for a more complex type, we get a simpler data representation, that does not rely on specific compiler optimizations to generate efficient code, even across abstraction boundaries.

Note that the previous type `'a Atomic.Loc.t` can be reconstructed as a dependent pair of a `'r` and a `('r, 'a) Atomic.Field.t`, which is expressible in OCaml as a GADT:

```
type 'a loc = Loc : 'r * ('r, 'a) Atomic.Field.t -> 'a loc
```

The main downside of this proposal is that it is harder to implement in the type-checker. The extension form `[%atomic.loc buf.front]` has typing rules that are very similar to a field access `buf.front`. On the other hand, `[%atomic.loc front]` interacts in a non-trivial way with the OCaml machinery for type-based disambiguation of record fields – several records with a field named `front` can co-exist in the typing environment. For technical reasons, there is also a non-trivial interaction with the type-checking of inline record types (record types that are not defined by themselves but only as the argument of a sum type constructor), which currently prevents from using this approach with those inline records. We have been working with OCaml maintainers to try to lift this limitation.

At the time of writing, there seems to be a consensus among OCaml maintainers to integrate support for atomic record fields in the language, but there is no final decision on which of the two forms should be preferred. Our work on Zoo relies on our experimental implementation of the first, simpler form for now, and could switch to the second form if it is preferred for merging into the main compiler.

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

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

that ensure 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.

7.2 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 arrays[12] 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
```

7.3 Generative immutable constructors

TODO

8 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.

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 Bílý, 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.

[12] https://github.com/clef-men/ocaml/tree/atomic_array. Warning: this link is not anonymized.

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 Nickolai 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 Nickolai 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 Nickolai Zeldovich. Verifying the daisynfs 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 Nickolai 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.

12 Arthur Charguéraud. *A Modern Eye on Separation Logic for Sequential Programs. (Un nouveau regard sur la Logique de Séparation pour les programmes séquentiels)*. 2023. URL: https://tel.archives-ouvertes.fr/tel-04076725.

13 Arthur Charguéraud, Jean-Christophe Filliâtre, Cláudio Lourenço, and Mário Pereira. GOSPEL - providing ocaml with a formal specification language. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*, volume 11800 of *Lecture Notes in Computer Science*, pages 484–501. Springer, 2019. doi:10.1007/978-3-030-30942-8_29.

14 Guillaume Claret. coq-of-ocaml. URL: https://github.com/formal-land/coq-of-ocaml.

15 Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for time and data abstraction. In Richard E. Jones, editor, *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, volume 8586 of *Lecture Notes in Computer Science*, pages 207–231. Springer, 2014. doi:10.1007/978-3-662-44202-9_9.

16 Arnaud Daby-Seesaram, Jean-Marie Madiot, François Pottier, Remy Seassau, and Irene Yoon. Osiris. URL: https://gitlab.inria.fr/fpottier/osiris.

17 Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. Compass: strong and compositional library specifications in relaxed memory separation logic. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 792–808. ACM, 2022. doi:10.1145/3519939.3523451.

18 Paulo Emílio de Vilhena and François Pottier. A separation logic for effect handlers. *Proc. ACM Program. Lang.*, 5(POPL):1–28, 2021. doi:10.1145/3434314.

19 Paulo Emílio de Vilhena, François Pottier, and Jacques-Henri Jourdan. Spy game: verifying a local generic solver in iris. *Proc. ACM Program. Lang.*, 4(POPL):33:1–33:28, 2020. doi:10.1145/3371101.

20 Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: A foundry for the deductive verification of rust programs. In Adrián Riesco and Min Zhang, editors, *Formal Methods and Software Engineering - 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24-27, 2022, Proceedings*, volume 13478 of *Lecture Notes in Computer Science*, pages 90–105. Springer, 2022. doi:10.1007/978-3-031-17244-1_6.

21 Brijesh Dongol and John Derrick. Verifying linearisability: A comparative survey. *ACM Comput. Surv.*, 48(2):19:1–19:43, 2015. doi:10.1145/2796550.

22 Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013. doi:10.1007/978-3-642-37036-6_8.

23 Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. Refinedrust: A type system for high-assurance verification of rust programs. *Proc. ACM Program. Lang.*, 8(PLDI):1115–1139, 2024. doi:10.1145/3656422.

24 Léon Gondelman, Jonas Kastberg Hinrichsen, Mário Pereira, Amin Timany, and Lars Birkedal. Verifying reliable network components in a distributed separation logic with dependent separation protocols. *Proc. ACM Program. Lang.*, 7(ICFP):847–877, 2023. doi:10.1145/3607859.

25 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

26 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011. doi:10.1007/978-3-642-20398-5_4.

27 Jaehwang Jung, Janggun Lee, Jaemin Choi, Jaewoo Kim, Sunho Park, and Jeehoon Kang. Modular verification of safe memory reclamation in concurrent separation logic. *Proc. ACM Program. Lang.*, 7(OOPSLA2):828–856, 2023. doi:10.1145/3622827.

28 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018. doi:10.1017/S0956796818000151.

29 Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.*, 4(POPL):45:1–45:32, 2020. doi:10.1145/3371113.

30 Vesa Karvonen. Kcas. URL: https://github.com/ocaml-multicore/kcas.

31 Vesa Karvonen and Carine Morel. Saturn. URL: https://github.com/ocaml-multicore/saturn.

32 Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. Mosel: a general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.*, 2(ICFP):77:1–77:30, 2018. doi:10.1145/3236772.

33 Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 205–217. ACM, 2017. doi:10.1145/3009837.3009855.

34 Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1):286–315, 2023. doi:10.1145/3586037.

35 Anton Lorenzen, Daan Leijen, Wouter Swierstra, and Sam Lindley. The functional essence of imperative binary search trees. *Proc. ACM Program. Lang.*, 8(PLDI):518–542, 2024. doi:10.1145/3656398.

36 Anil Madhavapeddy and Thomas Leonard. Eio. URL: https://github.com/ocaml-multicore/eio.

37 Glen Mével and Jacques-Henri Jourdan. Formal verification of a concurrent bounded queue in a weak memory model. *Proc. ACM Program. Lang.*, 5(ICFP):1–29, 2021. doi:10.1145/3473571.

38 Glen Mével, Jacques-Henri Jourdan, and François Pottier. Cosmo: a concurrent separation logic for multicore ocaml. *Proc. ACM Program. Lang.*, 4(ICFP):96:1–96:29, 2020. doi:10.1145/3408978.

39 Ike Mulder and Robbert Krebbers. Proof automation for linearizability in separation logic. *Proc. ACM Program. Lang.*, 7(OOPSLA1):462–491, 2023. doi:10.1145/3586043.

40 Ike Mulder, Robbert Krebbers, and Herman Geuvers. Diaframe: automated verification of fine-grained concurrent programs in iris. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 809–824. ACM, 2022. doi:10.1145/3519939.3523432.

41 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Alexander Pretschner, Doron Peled, and Thomas 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.

42 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.

43 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.

44 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.

45 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.

46 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.

47 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.

48 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.

49 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.

50 Iris Development Team. The coq mechanization of iris. URL: https://gitlab.mpi-sws.org/iris/iris/.

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.