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

ANONYMOUS AUTHOR(S)

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 [Karvonen and Morel [n. d.]] 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 [Jung, Krebbers, Jourdan, Bizjak, Birkedal and Dreyer 2018] concurrent separation logic.

1 Introduction

Designing concurrent algorithms, in particular *lock-free* algorithms, is a notoriously difficult task. In this paper, we are concerned with proving the correctness of these algorithms.

Example 1: physical equality. Consider, for example, the OCaml implementation of a concurrent stack [Center and Treiber 1986] in Figure 1. 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 lockfree 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 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.

Example 2: when physical identity matters. Consider another example given in Figure 2: the `Rcfd.close`[1] function from the `Eio` [Madhavapeddy and Leonard [n. d.]] 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`[2]. However, there is a complication.

[1]https://github.com/ocaml-multicore/eio/blob/main/lib_eio/unix/rcfd.ml

[2]Here, we make use of atomic record fields that were recently introduced in OCaml.

```
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
      )
```

Fig. 1. Implementation of a concurrent stack

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 lockfree algorithms; it also occurs in the Kcas [Karvonen [n. d.]] library[3].

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.

A formalized OCaml fragment for the verification of concurrent algorithms. These subtle aspects, illustrated through two realistic examples, justify the need for a faithful formal semantics of a fragment of OCaml tailored for the verification of concurrent algorithms. Ideally, of course, this fragment would include most of the language. However, the direct practical aim of this work—the verification of real-life libraries like Saturn [Karvonen and Morel [n. d.]]—led us to the following

[3]https://github.com/ocaml-multicore/kcas/blob/main/doc/gkmz-with-read-only-cmp-ops.md

```
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
      )
```

Fig. 2. **Rcfd**.close function from Eio [Madhavapeddy and Leonard [n. d.]]

design philosophy: only include what is actually needed to express and reason about concurrent algorithms in a convenient way.

In this paper, we show how we have designed a practical framework, Zoo[4], following this guideline. We review the works related to the verification of OCaml programs in Section 2; we describe our framework in Section 3; we detail the important features, including the treatment of physical equality, in Section 4 before concluding.

2 Related work

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

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

[4]https://github.com/clef-men/zoo

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 [Astrauskas, Bilý, Fiala, Grannan, Matheja, Müller, Poli and Summers 2022; Denis, Jourdan and Marché 2022; Filliâtre and Paskevich 2013; Jacobs, Smans, Philippaerts, Vogels, Penninckx and Piessens 2011; Lattuada, Hance, Cho, Brun, Subasinghe, Zhou, Howell, Parno and Hawblitzel 2023; Müller, Schwerhoff and Summers 2017; Pulte, Makwana, Sewell, Memarian, Sewell and Krishnaswami 2023; Swamy, Chen, Fournet, Strub, Bhargavan and Yang 2013], including to OCaml by Cameleer [Pereira and Ravara 2021], which uses the Gospel specification language [Charguéraud, Filliâtre, Lourenço and Pereira 2019] and Why3 [Filliâtre and Paskevich 2013].

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 [Sammler, Lepigre, Krebbers, Memarian, Dreyer and Garg 2021] and Rust [Gäher, Sammler, Jung, Krebbers and Dreyer 2024].

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` [Spector-Zabusky, Breitner, Rizkallah and Weirich 2018]. For imperative programs, this is more challenging. One solution is to use a monad, *e.g.* in `coq-of-ocaml` [Claret [n. d.]], 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 [Chajed, Tassarotti, Kaashoek and Zeldovich 2019; Charguéraud 2023; Gondelman, Hinrichsen, Pereira, Timany and Birkedal 2023] harnessing the power of separation logic, in particular the Iris [Jung, Krebbers, Jourdan, Bizjak, Birkedal and Dreyer 2018] concurrent separation logic. Iris is a very important work for the verification of concurrent algorithms. It allows for a rich, customizable ghost state that makes it possible to design complex *concurrent protocols*. In our experience, for the lockfree algorithms we considered, there is simply no alternative.

The tool closest to our needs so far is CFML [Charguéraud 2023], which targets OCaml. However, CFML does not support concurrency and is not based on Iris. The Osiris [Daby-Seesaram, Madiot, Pottier, Seassau and Yoon [n. d.]] framework, still under development, also targets OCaml and is based on Iris. However, it does not support concurrency and it is arguably non-trivial to introduce it since the semantics uses interaction trees [Xia, Zakowski, He, Hur, Malecha, Pierce and Zdancewic 2020]—the question of how to handle concurrency in this context is a research subject. Furthermore, Osiris is not usable yet; its ambition to support a large fragment of OCaml makes it a challenge.

3 Zoo in practice

In this section, we give an overview of the framework. We also provide a minimal example[5] demonstrating its use.

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.

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

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

Fig. 3. ZooLang syntax (omitting mutually recursive toplevel functions)

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 [Krebbers, Jourdan, Jung, Tassarotti, Kaiser, Timany, Charguéraud and Dreyer 2018; Krebbers, Timany and Birkedal 2017]. In addition, it supports Diaframe [Mulder, Krebbers and Geuvers 2022], enabling proof automation.

The ZooLang syntax is given in Figure 3[6], 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 $(e_1, \ldots, e_n)$ represents a non-constant constructor

[6]More precisely, it is the syntax of the surface language, including many Rocq notations.

(*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 $\{e_1, \ldots, e_n\}$ or the tagged record syntax ‘$C\ \{e_1, \ldots, e_n\}$. 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 4.5, are used to model atomic references.

The `Proph` and `Resolve` constructs are used to model *prophecy variables* [Jung, Lepigre, Parthasarathy, Rapoport, Timany, Dreyer and Jacobs 2020], as described in Section 4.6.

Finally, `Reveal` is a special source construct that we introduce to handle physical equality. We demystify it in Section 4.4.

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[7] 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.

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"
    ).
```

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)
```

[7] Actually, `ocaml2zoo` processes binary annotation files (`.cmt` files).

```
  | RET (); True
  >>>.
Proof. ... Qed.
```

Here, we use a *logically atomic specification* [da Rocha Pinto, Dinsdale-Young and Gardner 2014], which has been proven [Birkedal, Dinsdale-Young, Guéneau, Jaber, Svendsen and Tzevelekos 2021] to be equivalent to *linearizability* [Herlihy and Wing 1990] 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 [Allain, Clément, Moine and Scherer 2024] and arrays [Allain, Karvonen and Morel 2024] have been successfully used in verification efforts based on Zoo.

4.4 Physical equality

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 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].

4.5 Concurrent primitives

Zoo supports concurrent primitives both on atomic references (from **Atomic**) and atomic record fields (from **Atomic**`.`**Loc**[9]) 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`]`.

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

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

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.

4.6 Prophecy variables

Lockfree algorithms exhibit complex behaviors. To tackle them, Iris provides powerful mechanisms such as *prophecy variables* [Jung, Lepigre, Parthasarathy, Rapoport, Timany, Dreyer and Jacobs 2020]. 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 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

Clément Allain, Basile Clément, Alexandre Moine, and Gabriel Scherer. 2024. Snapshottable Stores. *Proc. ACM Program. Lang.* 8, ICFP (2024), 338–369. https://doi.org/10.1145/3674637

Clément Allain, Vesa Karvonen, and Carine Morel. 2024. Saturn: a library of verified concurrent data structures for OCaml 5. In *OCaml Workshop 2024 - ICFP 2024*. Armaël Guéneau and Sonja Heinze, Milan, Italy. https://inria.hal.science/hal-04681703

Vytautas Astrauskas, Aurel Bilý, Jonás Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. 2022. The Prusti Project: Formal Verification for Rust. In *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13260)*, Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez (Eds.). Springer, 88–108. https://doi.org/10.1007/978-3-031-06773-0_5

Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. 2021. Theorems for free from separation logic specifications. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–29. https://doi.org/10.1145/3473586

Thomas J. Watson IBM Research Center and R.K. Treiber. 1986. *Systems Programming: Coping with Parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center. https://books.google.fr/books?id=YQg3HAAACAAJ

Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 243–258. https://doi.org/10.1145/3341301.3359632

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

Arthur Charguéraud, Jean-Christophe Filliâtre, Cláudio Lourenço, and Mário Pereira. 2019. GOSPEL - Providing OCaml with a Formal Specification Language. In *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11800)*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Springer, 484–501. https://doi.org/10.1007/978-3-030-30942-8_29

Guillaume Claret. [n. d.]. *coq-of-ocaml*. https://github.com/formal-land/coq-of-ocaml

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8586)*, Richard E. Jones (Ed.). Springer, 207–231. https://doi.org/10.1007/978-3-662-44202-9_9

Arnaud Daby-Seesaram, Jean-Marie Madiot, François Pottier, Remy Seassau, and Irene Yoon. [n. d.]. *Osiris*. https://gitlab.inria.fr/fpottier/osiris

Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *Formal Methods and Software Engineering - 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24-27, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13478)*, Adrián Riesco and Min Zhang (Eds.). Springer, 90–105. https://doi.org/10.1007/978-3-031-17244-1_6

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *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 (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2024. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1115–1139. https://doi.org/10.1145/3656422

Léon Gondelman, Jonas Kastberg Hinrichsen, Mário Pereira, Amin Timany, and Lars Birkedal. 2023. Verifying Reliable Network Components in a Distributed Separation Logic with Dependent Separation Protocols. *Proc. ACM Program. Lang.* 7, ICFP (2023), 847–877. https://doi.org/10.1145/3607859

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

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

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

Vesa Karvonen. [n. d.]. *Kcas*. https://github.com/ocaml-multicore/kcas

Vesa Karvonen and Carine Morel. [n. d.]. *Saturn*. https://github.com/ocaml-multicore/saturn

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

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

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 286–315. https://doi.org/10.1145/3586037

Anil Madhavapeddy and Thomas Leonard. [n. d.]. *Eio*. https://github.com/ocaml-multicore/eio

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

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2017. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Dependable Software Systems Engineering*, Alexander Pretschner, Doron Peled, and Thomas Hutzelmann (Eds.). NATO Science for Peace and Security Series - D: Information and Communication Security, Vol. 50. IOS Press, 104–125. https://doi.org/10.3233/978-1-61499-810-5-104

Mário Pereira and António Ravara. 2021. Cameleer: A Deductive Verification Tool for OCaml. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 677–689. https://doi.org/10.1007/978-3-030-81688-9_31

Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. 2023. CN: Verifying Systems C Code with Separation-Logic Refinement Types. *Proc. ACM Program. Lang.* 7, POPL (2023), 1–32. https://doi.org/10.1145/3571194

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 158–174. https://doi.org/10.1145/3453483.3454036

Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is reasonable Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, June Andronick and Amy P. Felty (Eds.). ACM, 14–27. https://doi.org/10.1145/3167092

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2013. Secure distributed programming with value-dependent types. *J. Funct. Program.* 23, 4 (2013), 402–451. https://doi.org/10.1017/S0956796813000142

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119