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` 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 Zoo, a framework for verifying fine-grained concurrent OCaml 5 algorithms. Following a pragmatic approach, we defined a limited but sufficient fragment of the language to faithfully express these algorithms: ZooLang. We formalized its semantics carefully via a deep embedding in the Rocq proof assistant, uncovering subtle aspects of physical equality. We provide a tool to translate source OCaml programs into ZooLang syntax inside Rocq, where they can be specified and verified using the Iris concurrent separation logic. To illustrate the applicability of Zoo, we verified a subset of the standard library and a collection of fined-grained concurrent data structures from the `Saturn` and `Eio` libraries.

In the process, we also extended OCaml to more efficiently express certain concurrent programs.

1 Introduction

OCaml 5.0 was released on December 15th 2022, the first version of the OCaml programming language to support parallel execution of OCaml threads by merging the Multicore OCaml runtime [Sivaramakrishnan, Dolan, White, Jaffer, Kelly, Sahoo, Parimala, Dhiman and Madhavapeddy 2020]. It provided basic support in the language runtime to start and stop coarse-grained threads ("domains" in OCaml parlance) and support for strongly sequential atomic references in the standard library. The third-party library `domainslib` offered a simple scheduler for a pool of tasks, used to benchmark the parallel runtime. A world of parallel and concurrent software was waiting to be invented.

Shared-memory concurrency is a difficult programming domain, and existing ecosystems (C++, Java, Haskell, Rust, Go...) took decades to evolve comprehensive libraries of concurrent abstractions and data structures. In the last couple years, a handful of contributors to the OCaml system have been implementing basic libraries for concurrent and parallel programs in OCaml, in particular `Saturn` [Karvonen and Morel 2025], a library of lock-free thread-safe data structures (stacks, queues, a work-stealing dequeue, a skip list, a bag and a hash table), `Eio` [Madhavapeddy and Leonard 2025], a library of asynchronous IO and structured concurrency, and `Kcas` [Karvonen 2025a], a library offering a software-transactional-memory abstraction for users to build safe yet efficient thread-safe data structures.

Concurrent algorithms and data structures are extremely difficult to reason about. Their implementations tend to be fairly short, a few dozens of lines. There is only a handful of experts able to write such code, and many potential users. They are difficult to test comprehensively. These characteristics make them ideally suited for mechanized program verification.

We embarked on a mission to mechanize correctness proofs of OCaml concurrent algorithms and data structures as they are being written, in contact with their authors, rather than years later. In the process, we not only gained confidence in these complex new building blocks, but we also improved the OCaml language and its verification ecosystem.

OCaml language features. When studying the new codebases of concurrent and parallel data structures, we found a variety of unsafe idioms, working around expressivity or performance limitations with the OCaml language support for lock-free concurrent data structures. In particular, the support for *atomic references* in the OCaml library proved inadequate, as idiomatic concurrent data-structures need the more expressive feature of *atomic record fields*. We designed an extension of OCaml with atomic record fields, implemented it as a an experimental compiler variant, and succeeded in getting it integrated in the upstream OCaml compiler: it should be available as part of OCaml 5.4, which is not yet released at the time of writing.

Verification tools for concurrent programs. The state-of-the-art approach for mechanized verification of fine-grained concurrent algorithms is to use Iris [Jung, Krebbers, Jourdan, Bizjak, Birkedal and Dreyer 2018], a state-of-the-art mechanized *higher-order* concurrent separation logic with *user-defined ghost state*. Its expressivity allows to precisely capture delicate invariants, and to reason about the linearization points of fine-grained concurrent algorithms (including external [Vindum, Frumin and Birkedal 2022] and future-dependent [Chang, Jung, Sharma, Tassarotti, Kaashoek and Zeldovich 2023; Jung, Lepigre, Parthasarathy, Rapoport, Timany, Dreyer and Jacobs 2020; Vindum and Birkedal 2021] linearization). Iris provides a generic mechanism to define programming languages and program logics for them. Much of the existing Iris concurrent verification work has been performed in HeapLang, the exemplar Iris language; 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 11. We started our verification effort in HeapLang, but it eventually proved impractical to verify realistic OCaml libraries. Indeed, it lacks basic abstractions such as algebraic data types (tuples, mutable and immutable records, variants) and mutually recursive functions. Verifying OCaml programs in HeapLang requires difficult translation choices and introduces various encodings, to the point that the relation between the source and verified programs can become difficult to maintain and reason about. It also has very few standard data structures that can be directly reused. This view, we believe, is shared by many people in the Iris community.

We created a new Iris language, ZooLang, that can better express concurrent OCaml programs. Its feature set grew over time as we applied it to more verification scenarios, and we now believe that it allows practical verification of fine-grained concurrent OCaml 5 programs — including the use of our atomic record fields which were co-designed with ZooLang. We were influenced by the Perennial framework [Chajed, Tassarotti, Kaashoek and Zeldovich 2019], which achieved similar goals for the Go language with a focus on crash-safety. As in Perennial, we also provide a translator from (a subset of) OCaml to ZooLang: `ocaml2zoo`. We start from OCaml code and call our translator to obtain a deep ZooLang embedding inside Rocq; we can use lightweight annotations to guide the translation. Inside Rocq we define specifications using Iris, and prove them correct with respect to the ZooLang version, which is syntactically very close to the original OCaml source. We call the resulting framework Zoo.

One notable current limitation of ZooLang is that it assumes a sequentially-consistent memory model, whereas OCaml offers a weaker memory model [Dolan, Sivaramakrishnan and Madhavapeddy 2018]. We made the choice to ensure that we supported practical verification in a sequentially-consistent setting first; in the future we plan to equip ZooLang with the OCaml memory model as formalized in Cosmo [Mével, Jourdan and Pottier 2020]. We discuss the impact of this difference in **??**.

Specified OCaml semantics. Our Iris mechanization of ZooLang defines an operational semantics and a corresponding program logic. Our users on the other hand run their program through the standard OCaml implementation, which is not verified and does not have a precise formal specification. To bridge this formal-informal gap as well as reasonably possible, we carefully audit our ZooLang semantics to ensure that they coincide with OCaml's.

In doing so we discovered a hole in state-of-the-art language semantics for program verification (not just for OCaml), which is the treatment of *physical equality* (pointer quality). Physical equality is typically exposed to language users as an efficient but under-specified equality check, as the physical identity of objects may or may not be preserved by various compiler transformations. It is an essential aspect of concurrent programs, as it underlies the semantics of important atomic instructions such as `compare_and_set`. We found that the current informal semantics in OCaml is incomplete, it does not allow to reason on programs that use structured data which mix mutable and immutable constructors. Existing formalizations of physical equality in verification frameworks typically restrict it to primitive datatypes, but idiomatic concurrent programs do not fit within this restriction. We propose a precise specification of physical equality in Zoo that scales to the verification of all the concurrent programs we encountered.

Worse, our discussions with the maintainers of the OCaml implementation showed that implementors guarantee weaker properties of physical equalities than users assume, in particular they may allow *unsharing*, which makes some existing concurrent programs incorrect. We propose a small new language feature for OCaml, per-constructor unsharing control, which we also integrate in our ZooLang translation, to fix affected programs and verify them. Finally, we discussed these subtleties with authors who axiomatize physical equality within Rocq for the purpose of efficient extraction, and we found out that some subtleties we discovered could translate into incorrectness in their axiomatization, requiring careful restrictions.

Verification results. We verified a small library for ZooLang, typically a subset of the OCaml standard library. It can serve as building blocks to define our concurrent data structures. (The lack of such a reusable standard library is a current limitation of HeapLang.) We verified a specific component of the `Eio` library, whose author Thomas Leonard had pointed to us as being delicate to reason about and worth mechanizing. Finally, we verified a large subset of the `Saturn` library: stacks, queues (list-based and stack-based), and finally the Chase-Lev work-stealing queue [Chase and Lev 2005]. The `Saturn` implementation of these lock-free data structures are used by the concurrent schedulers proposed in the OCaml 5 library ecosystem, notably `domainslib` and `picos`. (The main `Saturn` concurrent structures missing from our verification are a skip-list and a hashtable.) Several of these data structures contained verification challenges, which we will describe in the relevant section.

Contributions. In summary, we claim the following contributions:

(1) Zoo, a program verification framework aimed at practical verification of concurrent OCaml programs, mechanized in Rocq. The language ZooLang comes with a program logic expressed in the Iris concurrent separation logic. A translator `ocaml2zoo` generates the Rocq embedding from source OCaml programs, and works well with OCaml tooling (`dune` support).
(2) The verification (in a sequentially-consistent model) of important structures coming from `Saturn`, the OCaml 5 library of lock-free data structures. In particular we present a precise concurrent invariant for the Chase-Lev work-stealing queue, which gives stronger specifications than its previous formalizations.
Gabriel{It would be nice to emphasize here a "proof technique" for concurrent verification that is novel in Clément's work. (Some novel usage setup for prophecy variables?)**}**
(3) The extension of OCaml with atomic record fields, which after significant design, implementation and discussion work have now been integrated into upstream OCaml.

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

Fig. 1. ZooLang syntax (omitting mutually recursive toplevel functions)

(4) The identification of blind spots in existing specifications of *physical equality* and a new specification that is precise enough to reason about `compare-and-set` in the various programs we considered.
In the process we identified a potential bug in existing OCaml programs related to *unsharing*, and we propose a small language extension to let users selectively disable unsharing.

2 Zoo in practice

2.1 Language

The core of Zoo is ZooLang: a concurrent, imperative, untyped, functional programming language fully formalized in Rocq. Its semantics has been designed to match OCaml's.

ZooLang comes with a program logic based on Iris: reasoning rules expressed in separation logic (including rules for the different constructs of the language) along with Rocq tactics that integrate into the Iris proof mode [Krebbers, Jourdan, Jung, Tassarotti, Kaiser, Timany, Charguéraud and Dreyer 2018; Krebbers, Timany and Birkedal 2017]. In addition, it supports Diaframe [Mulder and Krebbers 2023; Mulder, Krebbers and Geuvers 2022], enabling proof automation.

The ZooLang syntax is given in Figure 1[1], omitting mutually recursive toplevel functions that are treated specially. Expressions include standard constructs like booleans, integers, anonymous functions (that may be recursive), applications, **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 deeply embedded: variables (bound by functions and **let**) are quoted 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 (*e.g.* ‘`Some(` e `)`). Unlike OCaml, ZooLang has projections of the form e`.`<*proj*> (*e.g.* (x, y)`.<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`).

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

Parallelism is mainly supported through the **`Domain`** standard module (*e.g.* `domain_spawn`), including domain-local storage. Special constructs (`Xchg`, `CAS`, `FAA`; see Section 3.4) are used to model atomic references.

The `Proph` and `Resolve` constructs model *prophecy variables* [Jung, Lepigre, Parthasarathy, Rapoport, Timany, Dreyer and Jacobs 2020], see Section 3.5.

2.2 Translation from OCaml to ZooLang

While ZooLang lives in Rocq, we want to verify OCaml programs. To connect them we provide the tool `ocaml2zoo` to translate OCaml source files[2] into Rocq files containing ZooLang code. This tool can process entire `dune` projects, and support several libraries provided together or as dependencies of the project.

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

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

```
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. 2. Implementation of a concurrent stack

Consider, for example, the OCaml implementation of a concurrent stack [Treiber 1986] in Figure 2. The push function is translated into:

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

2.3 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* [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 specification is formed of a precondition and a postcondition, represented in angle brackets. But each is split in two parts, a *public* or *atomic* condition, and a *private* condition. Following standard Iris notations, the private conditions are on the outside (first line of the precondition, last line of the postcondition) and the atomic conditions are inside.

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

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

3 Zoo features

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

3.1 Algebraic data types

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

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

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

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

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

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".
```

3.2 Mutually recursive functions

Zoo supports non-recursive (`fun:` $x_1 \ldots x_n$ `=>` e) and recursive (`rec:` $f\ x_1 \ldots x_n$ `=>` e) functions but only *toplevel* mutually recursive functions. It is non-trivial to properly handle mutual recursion: when applying a mutually recursive function, a naive approach would replace calls to sibling functions by their respective bodies, but this typically makes the resulting expression unreadable. To prevent it, the mutually recursive functions have to know one another to preserve their names during β-reduction. We simulate this using some boilerplate that can be generated by `ocaml2zoo`. For instance, one may define two mutually recursive functions `f` and `g` as follows:

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

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

3.3 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 [anonymous] and arrays [anonymous] have been successfully used in verification efforts based on Zoo.

3.4 Concurrent primitives

Zoo supports concurrent primitives both on atomic references (from **`Atomic`**) and atomic record fields (from **`Atomic.Loc`**[3]) 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

[3]The **`Atomic.Loc`** module is part of the PR that implements atomic record fields (see Section 6).

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.

3.5 Prophecy variables

Lock-free 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`.

4 Physical equality

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

4.1 Physical equality in HeapLang

In HeapLang, this primitive is provided but restricted. Indeed, its semantics is only defined if either the expected or the desired value fits in a single memory word in the HeapLang value representation: literals (booleans, integers and pointers[4]) and literal injections[5]; otherwise, the program is stuck. In practice, this restriction forces the programmer to introduce an indirection [Iris development team 2025; Jung, Lepigre, Parthasarathy, Rapoport, Timany, Dreyer and Jacobs 2020; Vindum and Birkedal 2021] to physically compare complex values, *e.g.* lists. Furthermore, when the semantics is defined, values are compared using their Rocq representations; physical equality boils down to Rocq equality.

4.2 Physical equality in OCaml

In OCaml, physical equality is more tricky and often considered dangerous. *Structural equality*, which we describe in Section 5, should be the preferred way of comparing values. However, physical equality is typically much faster than structural equality, as it basically compiles to only one assembly instruction. Also, the `Atomic.compare_and_set` requires the comparison to be atomic, which is the case for physical equality but not structural equality.

In particular, the semantics of physical equality is *non-deterministic*. To see why, consider the case of *immutable blocks* representing constructors and immutable records (as opposed to *mutable blocks* representing mutable records), *e.g.* `Some 0`. The physical comparison of two seemingly identical immutable blocks, according to the Rocq representation (essentially a tag and a list of fields), may return `false`. Indeed, at runtime, a non-empty immutable block is represented by a pointer to a tagged memory block. In this case, physical equality is just pointer comparison. It is clear that two pointers being distinct does not imply the pointed memory blocks are. In other words, we cannot determine the result of physical comparison just by looking at the abstract values.

[4] HeapLang allows arbitrary pointer arithmetic and therefore inner pointers. This is forbidden in both OCaml and ZooLang, as any reachable value has to be compatible with the garbage collector.

[5] HeapLang has no primitive notion of constructor, only pairs and injections (left and right).

The question is then: what guarantees do we get when physical equality returns true and when it returns false? Given such guarantees, denoted by val_physeq and val_physneq, the non-deterministic semantics is reflected in the logic through the following specification:

```
Lemma physeq_spec v1 v2 :
  {{{ True }}}
    v1 == v2
  {{{ b, RET #b; ⌜(if b then val_physeq else val_physneq) v1 v2⌝ }}}
Proof. ... Qed.
```

The OCaml manual documents a partial specification for physical equality, which is precise for basic types such as references, but does not clearly extend to structured values containing a mix of immutable and mutable constructors. The only guarantee that it provides for all values is: if two values are physically equal, they are also structurally equal. This means we don't learn anything when two values are physically distinct.

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

4.3 When physical equality returns true

Let us go back to the concurrent stack of Figure 2 and more specifically the push function. To prove the atomic specification given in Section 2, we rely on the fact that, if **Atomic**.compare_and_set returns true, we actually observe the same list of values in the sense of Rocq equality. However, assuming only structural equality as per OCaml's specification of physical equality, this cannot be proven. To see why, consider, *e.g.*, a stack of references ('a ref). As structural equality is indeed *structural*, it traverses the references without comparing their *physical identities*. In other words, we cannot conclude the references are *exactly* the same. Hence, we cannot prove the specification.

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

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

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

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

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

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

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

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

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

The following tests may return `true`:

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

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

We believe this really is a shortcoming, at least from the verification perspective. Therefore, we propose to extend OCaml with *generative immutable blocks*[6]. These generative blocks are just like regular immutable blocks, except they cannot be shared. Hence, if physical equality on two generative blocks returns `true`, these blocks are definitionally equal in Rocq. At user level, this notion is materialized by *generative constructors*. For instance, to verify the expected `push` specification, we can use a generative version of lists:

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

4.4 When physical equality returns `false`

Most formalizations of physical equality in the literature do not give any guarantee when physical equality returns `false`. Many use-cases of physical equality, in particular retry loops, can be verified with only sufficient conditions on `true`. However, in some specific cases, more information is needed.

Consider the `Rcfd` module from the `Eio` [Madhavapeddy and Leonard 2025] library, an excerpt of which is given in Figure 3[7]. Thomas Leonard, its author, suggested that we verify this real-life example because of its intricate logical state. However, we found out that it is also relevant regarding the semantics of physical equality. Essentially, it consists in wrapping a file descriptor in a thread-safe way using reference-counting. At creation in the `make` function, the wrapper starts in the `Open` state. At some point, it can switch to the `Closing` state in the `close` function and can never go back to the `Open` state. Crucially, the `Open` state does not change throughout the lifetime of the data structure.

[6] Non-anonymous link: https://<clickable>

[7] We make use of *atomic record fields* as introduced in Section 6.1.

```ocaml
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 next = Closing (fun () -> Unix.close fd) 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. 3. **Rcfd** module from Eio [Madhavapeddy and Leonard 2025] (excerpt)

The interest of **Rcfd** lies in the `close` function. First, the function reads the state. If this state is **Closing**, it returns `false`; the wrapper has been closed. If this state is **Open**, it tries to switch to the **Closing** state using **Atomic.Loc**`.compare_and_set`; if this attempt fails, it also returns `false`. In this particular case, we would like to prove that the wrapper has been closed, or equivalently that **Atomic.Loc**`.compare_and_set` cannot have observed **Open**. Intuitively, this is true because there is only one **Open**.

Obviously, we need some kind of guarantee related to the *physical identity* of **Open** when **Atomic.Loc**`.compare_and_set` returns `false`. If **Open** were a mutable block, we could argue that this block cannot be physically distinct from itself; no optimization we know of would allow that. Unfortunately, it is an immutable block, and immutable blocks are subject to more optimizations. In fact, something surprising but allowed[8] by OCaml can happen: *unsharing*, the dual of sharing. Indeed, any immutable block can be unshared, that is reallocated. For example, the following test may theoretically return `false`:

```ocaml
let x = Some 0
let test = x == x (* false *)
```

Going back to **Rcfd**, we have a problem: in the second branch, the **Open** block corresponding to `prev` could be unshared, which would make **Atomic.Loc**`.compare_and_set` fail. Hence, we cannot prove the expected specification; in fact, the program as it is written has a bug.

To remedy this unfortunate situation, we propose to reuse the notions of generative immutable blocks, that we introduced to prevent sharing, to also forbid unsharing by the OCaml compiler – we implemented this in an experiment branch of OCaml.

[8]This has been confirmed by OCaml experts developing the Flambda backend.

In our semantics, each generative block is annotated with a *logical identifier*[9] representing its physical identity, much like a pointer for a mutable block. If physical equality on two generative blocks returns `false`, the two identifiers are necessarily distinct. Given this semantics, we can verify the `close` function. Indeed, if `Atomic.Loc.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`. We can verify this function after adding the following annotation:

```
type state =
  | Open of Unix.file_descr [@generative]
  | Closing of (unit -> unit)
```

4.5 Summary

In summary, we give the following specification to physical equality in ZooLang, which also serves as a precise specification of physical equality of a practical fragment of OCaml:

- On values whose low-level representation is an immediate integer, physical equality is immediate equality.
- On values whose low-level representation are mutable blocks at some location, or generative immutable blocks with some identity, physical equality is equality of locations or identities.
- On values whose low-level representation are immutable blocks, physical-equality is under-specified, but it implies that the blocks have the same tags and their arguments are in turn physically equal.
- Two values that fall into different categories above are never physically equal.

5 Structural equality

Structural equality is also supported. More precisely, it is not part of the semantics of the language but axiomatized on top of it[10]. The reason is that it is in fact difficult to specify for arbitrary values. In general, we have to compare graphs—which implies structural comparison may diverge.

Accordingly, the specification of $v_1 = v_2$ requires the (partial) ownership of a *memory footprint* corresponding to the union of the two compared graphs, giving the permission to traverse them safely. If it terminates, the comparison decides whether the two graphs are isomorphic (modulo representation conflicts, as described in Section 4). In Iris, this gives:

```
Axiom structeq_spec : ∀ v1 v2 footprint,
  val_traversable footprint v1 →
  val_traversable footprint v2 →
  {{{ structeq_footprint footprint }}}
    v1 = v2
  {{{ b, RET #b;
      structeq_footprint footprint ∗
      ⌜(if b then val_structeq else val_structneq) footprint v1 v2⌝ }}}.
```

Obviously, this general specification is not very convenient to work with. Fortunately, for abstract values (without any mutable part), we can prove a much simpler variant saying that structural equality boils down to physical equality:

```
Lemma structeq_spec_abstract v1 v2 :
  val_abstract v1 →
```

[9] Actually, for practical reasons, we distinguish identified and unidentified generative blocks.
[10] We could also have implemented it in ZooLang, but that would require more low-level primitives.

```
  val_abstract v2 →
  {{{ True }}}
    v1 = v2
  {{{ b, RET #b; ⌜(if b then val_physeq else val_physneq) v1 v2⌝ }}}
Proof. ... Qed.
```

6 OCaml extensions for fine-grained concurrent programming

Over the course of this work, we studied efficient fine-grained 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 between 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.

6.1 Atomic record fields

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, the Michael-Scott concurrent queue [Michael and Scott 1996] relies on a linked list structure that could be defined as follows:

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

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`. Instead, they use something like the following:

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

It is possible to statically forbid passing `Nil` to `as_atomic` to avoid error handling, by turning `'a node` into a GADT indexed over a type-level representation of its head constructor. Examples of this pattern can be found in the `Kcas` [Karvonen 2025a] 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.

6.1.1 Our atomic fields proposal. We proposed a design for atomic record fields as an OCaml language change proposal: RFC #39[11]. Declaring a record field atomic simply requires an `[@atomic]` attribute—and could eventually become a proper keyword of the language.

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

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

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

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.

Our proposed implementation[12] 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`.

(1) Our first implementation[13] 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[14] 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 a module `Atomic.Field`, that need both the record value of type `'r` and the field description. For example, the standard library exposes:

[11] De-anonymizing link: https://github.com/ocaml/RFCs/pull/39
[12] De-anonymizing link: https://github.com/ocaml/ocaml/pull/13404
[13] Non-anonymous link: https://<clickable>
[14] Non-anonymous link: https://<clickable>

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

6.2 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[15] 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
```

[15] Non-anonymous link: https://<clickable>

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 Standard data structures

To save users from reinventing the wheel, we provide a library of verified standard data structures — more or less a subset of the OCaml standard library. Most of these data structures[16] are completely reimplemented in Zoo and axiom-free, including the `Array`[17] module.

Sequential data structures. We provide verified implementations of various sequential data structures: array, dynamic array (vector), list, stack, queue (bounded and unbounded), double-ended queue. We claim that the proven specifications are modular and practical. In fact, most of these data structures have already been used to verify more complex ones — we present some in Section 8 and Section 10. Especially, we developed an extensive collection of flexible specifications for the iterators of the `Array` and `List` modules. Remarkably, our formalization of `Array` features different (fractional) predicates to express the ownership of either an entire array, a slice or even a circular slice — we use it to verify algorithms involving circular arrays, *e.g.* Chase-Lev working-stealing queue [Chase and Lev 2005] as presented in Section 10.4.

Concurrent data structures. We provide verified implementations of various concurrent data structures: domain[18] (including domain-local storage), mutex, semaphore, condition variable, write-once variable (also known as *ivar*), atomic array. Note that there is currently no `Atomic_array` module in the OCaml standard library, but we are planning to propose it.

8 Persistent data structures

To further demonstrate the practicality of Zoo, we verified a collection of persistent data structures. This includes purely functional data structures such as persistent stack and queue, but also efficient imperative implementations of persistent array [Conchon and Filliâtre 2007], store [Allain, Clément, Moine and Scherer 2024] and union-find [Allain, Clément, Moine and Scherer 2024].

Currently, verification of purely functional programs relies on the regular ZooLang translation, *i.e.* on a deeply embedded representation. However, this approach is quite cumbersome. In the future, it would be desirable to be able to verify them directly in Rocq, through a translation to Gallina. Similarly to Hacspec [Haselwarter, Hvass, Hansen, Winterhalter, Hritcu and Spitters 2024], this new translation would come with a generated proof of equivalence with the ZooLang representation.

$$\begin{array}{c} \Psi\ 1 \\ \hline \texttt{make}\ fd \\ \hline t.\ \ \mathsf{inv}\ t\ fd\ \Psi \end{array} \qquad \begin{array}{c} \mathsf{inv}\ t\ fd\ \Psi \\ \mathsf{wp}\ closed\ ()\ \{\ \mathrm{X}\ \texttt{false}\ \} \\ \forall q.\ \Psi\ q \twoheadrightarrow \mathsf{wp}\ open\ fd\ \{\ res.\ \Psi\ q * \mathrm{X}\ \texttt{true}\ res\ \} \\ \hline \texttt{use}\ t\ closed\ open \\ \hline res.\ \ \exists b.\ \mathrm{X}\ b\ res \end{array}$$

$$\begin{array}{c} \mathsf{inv}\ t\ fd\ \Psi \\ \Psi\ 1 \twoheadrightarrow \exists chars.\ \mathsf{unix\text{-}fd}\ fd\ 1\ chars \\ \hline \texttt{close}\ t \\ \hline ().\ \ \mathsf{closing}\ t \end{array} \qquad \begin{array}{c} \mathsf{inv}\ t\ fd\ \Psi \\ \hline \texttt{remove}\ t \\ \hline o.\ \ \begin{array}{c} \mathsf{closing}\ t \\ o = \mathrm{None} \lor o = \mathrm{Some}\ fd * \Psi\ 1 \end{array} \end{array}$$

Fig. 4. Rcfd specification (excerpt)

9 **Rcfd**: Parallelism-safe file descriptor

As mentioned in Section 4, the **Rcfd** module from the Eio library is particularly interesting in several respects. Not only does it justify the introduction of generative constructors in OCaml, but it also demonstrates the use of Iris for expressing realistic concurrent protocols.

Specification. The **Rcfd** module provides a parallelism-safe file descriptor (FD) relying internally on reference-counting. Interestingly, it is used in Eio in two different ways, more precisely two different ownership regimes: 1) any thread can try to access or close the FD; 2) any thread can try to access the FD but only the owner thread can close it — and is responsible for closing it. To verify all uses, the specification of **Rcfd** has to support both ownership regimes. However, due to space constraints, we consider a simplified specification given in Figure 4. The full verified specification can be found in the mechanization.

The specification features four operations[19].

`make` creates a new object t of type **Rcfd**`.t` wrapping a given FD fd, yielding the (persistent) invariant $\mathsf{inv}\ t\ fd\ \Psi$, where Ψ is an arbitrary fractional predicate. Crucially, the user must provide the full predicate $\Psi\ 1$, which is stored in the invariant. Once it is created, a wrapped FD can be accessed through the `use` operation and closed through the `close` operation.

`use` requires the invariant along with the weakest preconditions of the *closed* function, that is called if the FD has been flagged as closed, and *open* function, that is called if the FD is still open. To control the postconditions of the specification and the weakest preconditions, the user can choose an arbitrary predicate X parameterized by a boolean indicating whether the *closed* (`false`) or the *open* (`true`) was called. Furthermore, the *open* function is given a fraction of Ψ during its execution, thereby accessing the FD.

`close` requires the invariant and proving that the full ownership of Ψ entails the full ownership of the FD fd, which is necessary to call **Unix**`.close`. It yields $\mathsf{closing}\ t$, a persistent resource witnessing that t has been flagged as closed. Actually, the wrapped FD is not closed immediately. It will be closed only once it is possible, meaning all ongoing calls to `use` owning a fraction of the FD end.

[16]For practical reasons, to make them completely opaque, we chose to axiomatize a few functions from the **Domain** and **Random** modules. They could trivially be realized in Zoo.

[17]Our implementation of the **Array** module is compatible with the standard one. In particular, it uses the same low-level value representation.

[18]Domains are the units of parallelism in OCaml 5.

[19]We omitted two non-essential operations: `is_open` and `peek`.

Alternatively, instead of closing the FD, `remove` tries to retrieve the full ownership of Ψ. To achieve it, it exploits the same mechanism as `close` — flagging t as closed as witnessed by closing t — but also waits until all `use` calls are done.

Logical state. Thomas Leonard, the author of **`Rcfd`**, suggested verifying it to make sure the informal concurrent protocol he described in the OCaml interface was correct. This protocol introduces a notion of monotonic logical state — modeled in Iris using a specific resource algebra [Timany and Birkedal 2021] — to describe the evolution of a FD. Originally, there were four logical states but we found that only three are necessary for the verification: **open**, **closing/users** and **closing/no-users**.

In the **open** state, the FD is available for use, meaning any thread can access it through `use`. Physically, this corresponds to the **`Open`** constructor.

When some thread flags the FD as closed through `close` or `remove`, the state transitions from **open** to **closing/users**. Crucially, there can only be one such thread. In this state, the FD is not really closed yet because of ongoing `use` operations. Physically, this logical transition corresponds to switching from the **`Open`** to the **`Closing`** constructor using **`Atomic.Loc`**`.compare_and_set`.

Once all `use` operations have finished, when the reference-count reaches zero, it is time to actually 'close' the FD by calling the function carried by the **`Closing`** constructor. However, this has to be done only once. The 'closing' thread is the one that succeeds in updating the **`Closing`** constructor (to a new one carrying a no-op function) using again **`Atomic.Loc`**`.compare_and_set`. At this point, the state transitions from **closing/users** to **closing/no-users** and the wrapper no longer owns the FD.

Generative contructors. As explained in Section 4, the **`Open`** constructor has to be generative to prevent *unsharing*. In fact, the **`Closing`** constructor also has to be generative to prevent *sharing*: when the 'closing' thread wins the **`Atomic.Loc`**`.compare_and_set`, it must be true that the 'closing' function has not changed.

10 `Saturn`: A library of standard lock-free data structures

We have verified a collection of standard lock-free data structures from the `Saturn` [Karvonen and Morel 2025], `Eio` [Madhavapeddy and Leonard 2025] and `Picos` [Karvonen 2025b] libraries. It includes stacks, queues (list-based, array-based and stack-based), bags, and work-stealing queues. These data structures are meant to be used as is or adapted to fit specific needs. To cover a wide range of use cases, we often provide specialized variants: bounded or unbounded, single-producer (SP) or multi-producer (MP), single-consumer (SC) or multi-consumer (MC).

Due to space constraints, we focus on the most important algorithms and refrain from showing the corresponding (non-trivial) Iris invariants, which have all been mechanized in Rocq.

10.1 Stacks

We have verified three variants of Treiber stack [Treiber 1986]: 1) unbounded MPMC (the standard one), 2) bounded MPMC, 3) closable unbounded MPMC. This last variant features a closing mechanism: at some point, some thread can decide to close the stack, retrieving the current content and preventing others from operating on it. For example, we have used it to represent a set of vertex successors in the context of a concurrent graph implementation (not presented in this paper).

As explained in Section 4, the three verified stacks use generative constructors to prevent sharing. One may ask whether it would be easier to use a mutable version of lists instead. From the programmer's perspective, this is unsatisfactory because 1) the compiler will typically emit warnings complaining that the mutability is not exploited and 2) it does not really reflect the intent, *i.e.* we want precise guarantees for physical equality, not modify the list. From the verification perspective, this is also unsatisfactory because the mutable representation is more complex to write and reason about: pointers and points-to assertions versus pure Rocq list.

Although verified stacks may seem like a not-so-new contribution, we argue that it is the first verification of realistic OCaml implementations. For comparison, the exemplary concurrent stacks verified in Iris [Iris development team 2025] all suffer from the same flaw: they need to introduce indirections (pointers) to be able to use the compare-and-set primitive.

10.2 List-based queues

10.3 Stack-based queues

10.4 Work-stealing queues

11 Related work

In general there are two approaches to practical program verification:

11.1 Non-automated verification

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

The representation may be primitive, like Gallina for Rocq. For pure programs, **Gabriel[**this is rather straightforward**]{**I disagree, I believe that hs-to-coq is scientifically problematic as the translation is unsound for higher-order functions and infinite data.**}**, *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 2025], 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; Daby-Seesaram, Madiot, Pottier, Seassau and Yoon 2024; Gondelman, Hinrichsen, Pereira, Timany and Birkedal 2023] harnessing the power of separation logic. In particular, CFML [Charguéraud 2023] and Osiris [Daby-Seesaram, Madiot, Pottier, Seassau and Yoon 2024] target OCaml. However, CFML does not support concurrency and is not based on Iris. Osiris, still under development, is based on Iris but does not support concurrency.

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

11.2 Semi-automated verification

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

In *non-foundational* automated verification, the tool and the external solvers it may rely on are part of the trusted computing base. It is the most common approach and has been widely applied in the literature [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].

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

11.3 Physical equality

There is some literature in proof-assistant research on reflecting physical equality from the implementation language into the proof assistant, for optimization purposes: for example, exposing OCaml's physical equality as a predicate in Rocq lets us implement some memoization and sharing techniques in Rocq libraries. However, axiomatizing physical equality in the proof assistant is difficult, and can result in inconsistencies.

The earlier discussions of this question that we know come from Jourdan's thesis [Jourdan 2016] (chapter 9), also presented more succintly in [Braibant, Jourdan and Monniaux 2014]. This work introduces the Jourdan condition, that physical equality implies equality of values. [Boulmé 2021] extends the treatment of physical equality in Rocq, integrating it in an "extraction monad" to control it more safely. There is also a discussion of similar optimizations in Lean in [Selsam, Hudon and de Moura 2020].

The correctness of the axiomatization of physical equality depends on the type of the values being compared: axiomatizations are typically polymorphic on any type `A`, but their correctness depends on the specific `A` being considered. For example, it is easy to correctly characterize physical on natural numbers, and other non-dependent types arising in Rocq verification projects. One difficulty in HeapLang and ZooLang is that they are untyped languages, their representation of `0` and `false` has the same type. But our remark that structural equality (in OCaml) does not necessarily coincide with definitional equality (in Rocq) also applies to other Rocq types: our examples with an existential **`Any`** constructor (see Section 4) can be reproduced with Σ-types.

12 Conclusion and future work

We presented Zoo, a framework for the verification of concurrent OCaml 5 programs. While it is not yet available on `opam`, it can be installed and used in other Rocq projects. We provide a minimal example[20] demonstrating its use.

Zoo has already been used to verify sequential imperative algorithms [anonymous] and is currently being used to verify a library of lock-free data structures. 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. doi:10.1145/3674637

[20]Non-anonymous link: https://<clickable>

Vytautas Astrauskas, Aurel Bílý, 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. doi: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. doi:10.1145/3473586

Sylvain Boulmé. 2021. *Formally Verified Defensive Programming (efficient Coq-verified computations from untrusted ML oracles)*. Accreditation to supervise research. Université Grenoble-Alpes. https://hal.science/tel-03356701 See also http://www-verimag.imag.fr/ boulme/hdr.html.

Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux. 2014. Implementing and Reasoning About Hash-consed Data Structures in Coq. *J. Autom. Reason.* 53, 3 (2014), 271–304. doi:10.1007/S10817-014-9306-0

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. doi:10.1145/3341301.3359632

Yun-Sheng Chang, Ralf Jung, Upamanyu Sharma, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2023. Verifying vMVCC, a high-performance transaction library using multi-version concurrency control. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, Roxana Geambasu and Ed Nightingale (Eds.). USENIX Association, 871–886. https://www.usenix.org/conference/osdi23/presentation/chang

Arthur Charguéraud. 2023. *Habilitation thesis: A Modern Eye on Separation Logic for Sequential Programs. (Un nouveau regard sur la Logique de Séparation pour les programmes séquentiels)*. Université de Strasbourg. 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. doi:10.1007/978-3-030-30942-8_29

David Chase and Yossi Lev. 2005. Dynamic circular work-stealing deque. In *SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA*, Phillip B. Gibbons and Paul G. Spirakis (Eds.). ACM, 21–28. doi:10.1145/1073970.1073974

Guillaume Claret. 2025. coq-of-ocaml. https://github.com/formal-land/coq-of-ocaml

Sylvain Conchon and Jean-Christophe Filliâtre. 2007. A persistent union-find data structure. In *Proceedings of the ACM Workshop on ML, 2007, Freiburg, Germany, October 5, 2007*, Claudio V. Russo and Derek Dreyer (Eds.). ACM, 37–46. doi:10.1145/1292535.1292541

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. doi:10.1007/978-3-662-44202-9_9

Arnaud Daby-Seesaram, Jean-Marie Madiot, François Pottier, Remy Seassau, and Irene Yoon. 2024. 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. doi:10.1007/978-3-031-17244-1_6

Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding data races in space and time. *SIGPLAN Not.* 53, 4 (June 2018), 242–255. doi:10.1145/3296979.3192421

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. doi: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. doi: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. doi:10.1145/3607859

Philipp G. Haselwarter, Benjamin Salling Hvass, Lasse Letager Hansen, Théo Winterhalter, Catalin Hritcu, and Bas Spitters. 2024. The Last Yard: Foundational End-to-End Verification of High-Speed Cryptography. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2024, London, UK, January 15-16, 2024*, Amin

Timany, Dmitriy Traytel, Brigitte Pientka, and Sandrine Blazy (Eds.). ACM, 30–44. doi:10.1145/3636501.3636961
Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. doi:10.1145/78969.78972
Iris development team. 2025. Iris examples. https://gitlab.mpi-sws.org/iris/examples/
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. doi:10.1007/978-3-642-20398-5_4
Jacques-Henri Jourdan. 2016. *Verasco: a Formally Verified C Static Analyzer. (Verasco: un analyseur statique pour C formellement vérifié).* Ph. D. Dissertation. Paris Diderot University, France. https://tel.archives-ouvertes.fr/tel-01327023
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. doi: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. doi:10.1145/3371113
Vesa Karvonen. 2025a. Kcas. https://github.com/ocaml-multicore/kcas
Vesa Karvonen. 2025b. Picos. https://github.com/ocaml-multicore/picos
Vesa Karvonen and Carine Morel. 2025. 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. doi: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. doi: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. doi:10.1145/3586037
Anil Madhavapeddy and Thomas Leonard. 2025. Eio. https://github.com/ocaml-multicore/eio
Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2020. Cosmo: a concurrent separation logic for multicore OCaml. *Proc. ACM Program. Lang.* 4, ICFP (2020), 96:1–96:29. doi:10.1145/3408978
Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, James E. Burns and Yoram Moses (Eds.). ACM, 267–275. doi:10.1145/248052.248106
Ike Mulder and Robbert Krebbers. 2023. Proof Automation for Linearizability in Separation Logic. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 462–491. doi:10.1145/3586043
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. doi: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. doi: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. doi: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. doi: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. doi:10.1145/3453483.3454036
Daniel Selsam, Simon Hudon, and Leonardo de Moura. 2020. Sealing pointer-based optimizations behind pure functions. *Proc. ACM Program. Lang.* 4, ICFP, Article 115 (Aug. 2020), 20 pages. doi:10.1145/3408997

KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting parallelism onto OCaml. *Proc. ACM Program. Lang.* 4, ICFP, Article 113 (Aug. 2020), 30 pages. doi:10.1145/3408995

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. doi: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. doi:10.1017/S0956796813000142

Amin Timany and Lars Birkedal. 2021. Reasoning about monotonicity in separation logic. In *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, Catalin Hritcu and Andrei Popescu (Eds.). ACM, 91–104. doi:10.1145/3437992.3439931

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

Simon Friis Vindum and Lars Birkedal. 2021. Contextual refinement of the Michael-Scott queue (proof pearl). In *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, Catalin Hritcu and Andrei Popescu (Eds.). ACM, 76–90. doi:10.1145/3437992.3439930

Simon Friis Vindum, Dan Frumin, and Lars Birkedal. 2022. Mechanized verification of a fine-grained concurrent queue from meta's folly library. In *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, Andrei Popescu and Steve Zdancewic (Eds.). ACM, 100–115. doi:10.1145/3497775.3503689