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

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Abstract

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

We present Zoo, a framework for verifying fine-grained concurrent OCaml 5 algorithms. We followed a pragmatic approach, studying OCaml code written by concurrency experts to delimit a limited but sufficient fragment of the language to express these algorithms: ZooLang. We formalized its semantics carefully via a deep embedding in the Rocq proof assistant. We provide a tool to translate source OCaml programs into ZooLang syntax inside Rocq, where they can be specified and verified using the Iris concurrent separation logic.

We verified a subset of the standard library along with fine-grained concurrent algorithms, including Treiber stack and a use of reference-counting for file descriptors from the `Eio` library. This formalization work uncovered delicate questions of programming language semantics, especially around physical equality. In the process, we also extended OCaml to more efficiently express certain concurrent programs.

2012 ACM Subject Classification Theory of computation → Program verification

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

Digital Object Identifier 10.4230/LIPIcs.ITP.2025.23

1 Introduction

Designing concurrent algorithms, in particular fine-grained concurrent algorithms, is a notoriously difficult task. Similarly, the formal verification of such algorithms is also difficult. It typically involves finding and reasoning about non-trivial linearization points [21, 30, 55, 56, 11].

In recent years, concurrent separation logic [5] has enabled significant progress in this area. In particular, the development of Iris [29], a state-of-the-art mechanized *higher-order* concurrent separation logic with *user-defined ghost state*, has nourished a rich and successful line of works [30, 55, 56, 11, 6, 28, 49, 38, 37, 17, 43, 41, 40], dealing with external [56] and future-dependent [30, 55, 11] linearization points, relaxed memory [38, 37, 17, 43] and automation [41, 40].

Most of these works [30, 55, 56, 6, 28, 49, 41, 40] and many others [19, 45, 54, 35] rely on HeapLang [52], the exemplar Iris language. HeapLang is a concurrent, imperative, untyped, call-by-value functional language. To the best of our knowledge, it is currently the closest language to OCaml 5 in the Iris ecosystem—we review the existing frameworks in Section 2. It has been extended to handle weak memory [38] and algebraic effects [18].

Although HeapLang is theoretically expressive enough to represent OCaml programs, our experiments showed that it is fairly impractical when it comes to verifying large OCaml libraries. Indeed, it lacks basic abstractions such as algebraic data types (tuples, mutable and

16th International Conference on Interactive Theorem Proving (ITP 2025).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:20
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

immutable records, variants) and mutually recursive functions. Verifying OCaml programs in HeapLang requires difficult translation choices and introduces various encodings, to the point that the relation between the source and verified programs can become difficult to maintain and reason about. It also has very few standard data structures that can be directly reused. This view, we believe, is shared by many people in the Iris community. Our first motivation in this work is therefore to fill this gap by providing a more practical OCaml-like verification language: ZooLang. This language consists of a subset of OCaml 5 extended with atomic record fields and equipped with a formal semantics and a program logic based on Iris. We were influenced by the Perennial [8, 9, 10, 11] framework, which achieved similar goals for the Go language with a focus on crash-safety. As in Perennial, we also provide a translator from OCaml to ZooLang: `ocaml2zoo`. We call the resulting framework Zoo.

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

In summary, we claim the following contributions:

1. We present ZooLang, a convenient subset of OCaml 5 formalized in Rocq (Sections 3 and 4). ZooLang comes with a program logic based on Iris and supports proof automation through Diaframe [41, 40].
2. We provide a translator from OCaml to ZooLang: `ocaml2zoo` (Section 3), built for practical applications—it supports full projects using the `dune` build system.
3. We formalize physical equality (Section 5) and structural equality (Section 6) in a faithful way. To our knowledge this is the first detailed specification of physical equality for a practical fragment of OCaml. The careful analysis of these notions suggests a new OCaml feature: *generative constructors*.
4. We extend OCaml with *atomic record fields* and *atomic arrays* to ease the development of fine-grained concurrent algorithms (Section 7).
5. We verify realistic use cases (Section 5) involving physical equality: (1) Treiber stack [7], (2) a thread-safe wrapper around a file descriptor using reference-counting from the `Eio` [36] library.

2 Related work

In general there are two approaches to practical program verification:

2.1 Non-automated verification

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

The representation may be primitive, like Gallina for Rocq. For pure programs, this is rather straightforward, *e.g.* in `hs-to-coq` [50]. For imperative programs, this is more challenging. One solution is to use a monad, *e.g.* in `coq-of-ocaml` [14], but it does not support concurrency.

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

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

2.2 Semi-automated verification

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

In *non-foundational* automated verification, the tool and the external solvers it may rely on are part of the trusted computing base. It is the most common approach and has been widely applied in the literature [51, 42, 26, 20, 1, 22, 34, 46], including to OCaml by Cameleer [44], which uses the Gospel specification language [13] and Why3 [22].

In *foundational* automated verification, the proofs are checked by a proof assistant like Rocq, meaning the automation does not have to be trusted. To our knowledge, it has been applied to C [47] and Rust [23].

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

2.3 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 [27] (chapter 9), also presented more succintly in [4]. This work introduces the Jourdan condition, that physical equality implies equality of values. [3] 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 [48].

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 5) can be reproduced with Σ-types.

$$
\begin{array}{llcl}
\textsc{Rocq}\ \text{term} & t & & \\
\text{constructor} & C & & \\
\text{projection} & \mathit{proj} & & \\
\text{record field} & \mathit{fld} & & \\
\text{identifier} & s, f & \in & \text{String} \\
\text{integer} & n & \in & \mathbb{Z} \\
\text{boolean} & b & \in & \mathbb{B} \\
\text{binder} & x & ::= & \texttt{<>} \mid s \\
\text{unary operator} & \oplus & ::= & \texttt{\textasciitilde} \mid \texttt{-} \\
\text{binary operator} & \otimes & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{`quot`} \mid \texttt{`rem`} \mid \texttt{`land`} \mid \texttt{`lor`} \mid \texttt{`lsl`} \mid \texttt{`lsr`} \\
 & & \mid & \texttt{<=} \mid \texttt{<} \mid \texttt{>=} \mid \texttt{>} \mid \texttt{=} \mid \neq \mid \texttt{==} \mid \texttt{!=} \\
 & & \mid & \texttt{and} \mid \texttt{or} \\
\text{expression} & e & ::= & t \mid s \mid \texttt{\#}n \mid \texttt{\#}b \\
 & & \mid & \texttt{fun:}\ x_1 \dots x_n\ \texttt{=>}\ e \mid \texttt{rec:}\ f\ x_1 \dots x_n\ \texttt{=>}\ e \mid e_1\ e_2 \\
 & & \mid & \texttt{let:}\ x\ \texttt{:=}\ e_1\ \texttt{in}\ e_2 \mid e_1\ \texttt{;;}\ e_2 \\
 & & \mid & \texttt{let:}\ f\ x_1 \dots x_n\ \texttt{:=}\ e_1\ \texttt{in}\ e_2 \mid \texttt{letrec:}\ f\ x_1 \dots x_n\ \texttt{:=}\ e_1\ \texttt{in}\ e_2 \\
 & & \mid & \texttt{let:}\ \texttt{`}C\ x_1 \dots x_n\ \texttt{:=}\ e_1\ \texttt{in}\ e_2 \mid \texttt{let:}\ x_1\texttt{,}\dots\texttt{,}x_n\ \texttt{:=}\ e_1\ \texttt{in}\ e_2 \\
 & & \mid & \oplus e \mid e_1 \otimes e_2 \\
 & & \mid & \texttt{if:}\ e_0\ \texttt{then}\ e_1\ (\texttt{else}\ e_2)^? \\
 & & \mid & \texttt{for:}\ x\ \texttt{:=}\ e_1\ \texttt{to}\ e_2\ \texttt{begin}\ e_3\ \texttt{end} \\
 & & \mid & \S C \mid \texttt{`}C\ \texttt{(}e_1\texttt{,}\dots\texttt{,}e_n\texttt{)} \mid \texttt{(}e_1\texttt{,}\dots\texttt{,}e_n\texttt{)} \mid e\texttt{.<}\mathit{proj}\texttt{>} \\
 & & \mid & \texttt{[]} \mid e_1\ \texttt{::}\ e_2 \\
 & & \mid & \texttt{`}C\ \texttt{\{}e_1\texttt{,}\dots\texttt{,}e_n\texttt{\}} \mid \texttt{\{}e_1\texttt{,}\dots\texttt{,}e_n\texttt{\}} \mid e\texttt{.\{}\mathit{fld}\texttt{\}} \mid e_1\ \texttt{<-\{}\mathit{fld}\texttt{\}}\ e_2 \\
 & & \mid & \texttt{ref}\ e \mid \texttt{!}e \mid e_1\ \texttt{<-}\ e_2 \\
 & & \mid & \texttt{match:}\ e_0\ \texttt{with}\ \mathit{br}_1\ \texttt{|}\ \dots\ \texttt{|}\ \mathit{br}_n\ (\texttt{|\_}\ (\texttt{as}\ s)^?\ \texttt{=>}\ e)^?\ \texttt{end} \\
 & & \mid & e\texttt{.[}\mathit{fld}\texttt{]} \mid \texttt{Xchg}\ e_1\ e_2 \mid \texttt{CAS}\ e_1\ e_2\ e_3 \mid \texttt{FAA}\ e_1\ e_2 \\
 & & \mid & \texttt{Proph} \mid \texttt{Resolve}\ e_0\ e_1\ e_2 \\
\text{branch} & \mathit{br} & ::= & C\ (x_1 \dots x_n)^?\ (\texttt{as}\ s)^?\ \texttt{=>}\ e \\
 & & \mid & \texttt{[]}\ (\texttt{as}\ s)^?\ \texttt{=>}\ e \mid x_1\ \texttt{::}\ x_2\ (\texttt{as}\ s)^?\ \texttt{=>}\ e \\
\text{toplevel value} & v & ::= & t \mid \texttt{\#}n \mid \texttt{\#}b \\
 & & \mid & \texttt{fun:}\ x_1 \dots x_n\ \texttt{=>}\ e \mid \texttt{rec:}\ f\ x_1 \dots x_n\ \texttt{=>}\ e \\
 & & \mid & \S C \mid \texttt{`}C\ \texttt{(}v_1\texttt{,}\dots\texttt{,}v_n\texttt{)} \mid \texttt{(}v_1\texttt{,}\dots\texttt{,}v_n\texttt{)} \\
 & & \mid & \texttt{[]} \mid v_1\ \texttt{::}\ v_2
\end{array}
$$

Figure 1 ZooLang syntax (omitting mutually recursive toplevel functions)

3 Zoo in practice

3.1 Language

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

ZooLang comes with a program logic based on Iris: reasoning rules expressed in separation logic (including rules for the different constructs of the language) along with Rocq tactics that integrate into the Iris proof mode [33, 32]. In addition, it supports Diaframe [41, 40], enabling proof automation.

The ZooLang syntax is given in Figure 1[1], omitting mutually recursive toplevel functions

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

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

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 4.4) are used to model atomic references.

The `Proph` and `Resolve` constructs model *prophecy variables* [30], see Section 4.5.

3.2 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.

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

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

[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
      )
```

Figure 2 Implementation of a concurrent stack

3.3 Specifications and proofs

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

```
Lemma stack_push_spec t ι v :
  <<< stack_inv t ι
    | ∀∀ vs, stack_model t vs >>>
    stack_push t v @ ↑ι
  <<< stack_model t (v :: vs)
    | RET (); True >>>.
Proof. ... Qed.
```

Here, we use a *logically atomic specification* [15], which has been proven [2] to be equivalent to *linearizability* [25] in sequentially consistent memory models.

Similarly to Hoare triples, the specification is formed of a precondition and a postcondition, represented in 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.

4 Zoo features

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

4.1 Algebraic data types

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

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

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

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

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

4.2 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.
```

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

4.4 Concurrent primitives

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

One important aspect of this translation is that atomic accesses (`Atomic.get` and `Atomic.set`) correspond to plain loads and stores. This is because we are working in a sequentially consistent memory model: there is no difference between atomic and non-atomic memory locations.

4.5 Prophecy variables

Lock-free algorithms exhibit complex behaviors. To tackle them, Iris provides powerful mechanisms such as *prophecy variables* [30]. Essentially, prophecy variables can be used to predict the future of the program execution and reason about it. They are key to handle

[3] The `Atomic.Loc` module is part of the PR that implements atomic record fields (see Section 7).

future-dependent linearization points: linearization points that may or may not occur at a given location in the code depending on a future observation.

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

5 Physical equality

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

5.1 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 [53, 30, 55] 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.

5.2 Physical equality in OCaml

In OCaml, physical equality is more tricky and often considered dangerous. *Structural equality*, which we describe in Section 6, 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.

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 }}}
```

[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).

```
    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.

5.3 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 3, we rely on the fact that, if `Atomic.compare_and_set` returns `true`, we actually observe the same list of values in the sense of Rocq equality. However, assuming only structural equality as per OCaml's specification of physical equality, this 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 3 is incorrect. To fix it, we would need to reason *modulo physical equality*, which is non-standard and quite burdensome.

We believe this really is a shortcoming, at least from the verification perspective. Therefore, we propose to extend OCaml with *generative immutable blocks*[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]
```

5.4 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` [36] 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.

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`.

[6] Non-anonymous link: https://<clickable>
[7] We make use of *atomic record fields* as introduced in Section 7.1.

```
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
```

Figure 3 `Rcfd` module from `Eio` [36] (excerpt)

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`:

```
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.

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

[8] This has been confirmed by OCaml experts developing the Flambda backend.
[9] Actually, for practical reasons, we distinguish identified and unidentified generative blocks.

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)
```

5.5 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.

6 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 5). 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 →
  val_abstract v2 →
  {{{ True }}}
    v1 = v2
```

[10] We could also have implemented it in ZooLang, but that would require more low-level primitives.

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

7 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.

7.1 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 [39] 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` [31] library by Vesa Karvonen. It is difficult to write correctly and use, in particular as unsafe casts can sometimes hide type-errors in the intended static discipline.

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

7.1.1 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`.

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.

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.

[11] De-anonymizing link: https://github.com/ocaml/RFCs/pull/39
[12] De-anonymizing link: https://github.com/ocaml/ocaml/pull/13404

7.2 Atomic arrays

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

Our implementation of atomic arrays[13] builds on top of the type `'a Atomic.Loc.t` we described in the previous section, and it relies on two new low-level primitives provided by the compiler:

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

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

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

8 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[14] 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

1 Vytautas Astrauskas, Aurel Bilý, Jonás Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. The prusti project: Formal verification for rust. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, volume 13260 of *Lecture Notes in Computer Science*, pages 88–108. Springer, 2022. doi:10.1007/978-3-031-06773-0_5.

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

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

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

4 Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux. Implementing and reasoning about hash-consed data structures in coq. *J. Autom. Reason.*, 53(3):271–304, 2014. URL: https://doi.org/10.1007/s10817-014-9306-0, doi:10.1007/S10817-014-9306-0.

5 Stephen Brookes and Peter W. O'Hearn. Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65, 2016. doi:10.1145/2984450.2984457.

6 Quentin Carbonneaux, Noam Zilberstein, Christoph Klee, Peter W. O'Hearn, and Francesco Zappa Nardelli. Applying formal verification to microkernel IPC at meta. In Andrei Popescu and Steve Zdancewic, editors, *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, pages 116–129. ACM, 2022. doi:10.1145/3497775.3503681.

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

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

9 Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. Gojournal: a verified, concurrent, crash-safe journaling system. In Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 423–439. USENIX Association, 2021. URL: https://www.usenix.org/conference/osdi21/presentation/chajed.

10 Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying the daisynfs concurrent and crash-safe file system with sequential reasoning. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 447–463. USENIX Association, 2022. URL: https://www.usenix.org/conference/osdi22/presentation/chajed.

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

12 Arthur Charguéraud. *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, 2023. URL: https://tel.archives-ouvertes.fr/tel-04076725.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

27 Jacques-Henri Jourdan. *Verasco: a Formally Verified C Static Analyzer. (Verasco: un analyseur statique pour C formellement vérifié)*. PhD thesis, Paris Diderot University, France, 2016. URL: https://tel.archives-ouvertes.fr/tel-01327023.

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

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

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

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

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

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

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

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

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

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

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

39 Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In James E. Burns and Yoram Moses, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, pages 267–275. ACM, 1996. doi:10.1145/248052.248106.

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

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

42 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Alexander Pretschner, Doron Peled, and Thomas Hutzelmann, editors, *Dependable Software Systems Engineering*, volume 50 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 104–125. IOS Press, 2017. doi:10.3233/978-1-61499-810-5-104.

43 Sunho Park, Jaewoo Kim, Ike Mulder, Jaehwang Jung, Janggun Lee, Robbert Krebbers, and Jeehoon Kang. A proof recipe for linearizability in relaxed memory separation logic. *Proc. ACM Program. Lang.*, 8(PLDI):175–198, 2024. doi:10.1145/3656384.

44 Mário Pereira and António Ravara. Cameleer: A deductive verification tool for ocaml. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 677–689. Springer, 2021. doi:10.1007/978-3-030-81688-9_31.

45 François Pottier, Armaël Guéneau, Jacques-Henri Jourdan, and Glen Mével. Thunks and debits in separation logic with time credits. *Proc. ACM Program. Lang.*, 8(POPL):1482–1508, 2024. doi:10.1145/3632892.

46 Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. CN: verifying systems C code with separation-logic refinement types. *Proc. ACM Program. Lang.*, 7(POPL):1–32, 2023. doi:10.1145/3571194.

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

48 Daniel Selsam, Simon Hudon, and Leonardo de Moura. Sealing pointer-based optimizations behind pure functions. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020. doi:10.1145/3408997.

49 Thomas Somers and Robbert Krebbers. Verified lock-free session channels with linking. *Proc. ACM Program. Lang.*, 8(OOPSLA2):588–617, 2024. doi:10.1145/3689732.

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

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

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

53 Iris Development Team. Iris examples. URL: https://gitlab.mpi-sws.org/iris/examples/.

54 Amin Timany, Armaël Guéneau, and Lars Birkedal. The logical essence of well-bracketed control flow. *Proc. ACM Program. Lang.*, 8(POPL):575–603, 2024. doi:10.1145/3632862.

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

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