

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 [21] aim at addressing this need. From the perspective of formal verification, this is an opportunity to apply and further state-of-the-art techniques to provide stronger guarantees.

We present a framework for verifying fine-grained concurrent OCAML 5 algorithms. Following a pragmatic approach, we support a limited but sufficient fragment of the language whose semantics has been carefully formalized to faithfully express such algorithms. Source programs are translated to a deeply-embedded language living inside ROCQ where they can be specified and verified using the IRIS [18] concurrent separation logic.

2012 ACM Subject Classification Replace `ccsdsc` macro with valid one

Keywords and phrases ROCQ, program verification, separation logic

Digital Object Identifier 10.4230/LIPIcs.ITP.2025.23

1 Introduction

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

1.0.0.1 Example 1: physical equality.

Consider, for example, the OCAML implementation of a concurrent stack [5] in Figure 1. Essentially, it consists of an atomic reference to a list that is updated atomically using the `Atomic.compare_and_set` primitive. While this simple implementation—it is indeed one of the simplest lockfree algorithms—may seem easy to verify, it is actually more subtle than it looks.

Indeed, the semantics of `Atomic.compare_and_set` involves *physical equality*: if the content of the atomic reference is physically equal to the expected value, it is atomically updated to the new value. Comparing physical equality is tricky and can be dangerous—this is why *structural equality* is often preferred—because the programmer has few guarantees about the *physical identity* of a value. In particular, the physical identity of a list, or more generally of an inhabitant of an algebraic data type, is not really specified. The only guarantee is: if two values are physically equal, they are also structurally equal. Apparently, we don’t learn anything interesting when two values are physically distinct. Going back to our example, this is fortunately not an issue, since we always retry the operation when `Atomic.compare_and_set` returns `false`.

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



© Anonymous author(s);

licensed under Creative Commons License CC-BY 4.0

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

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:17

Leibniz International Proceedings in Informatics



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

```

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 1** Implementation of a concurrent stack

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

48 1.0.0.2 Example 2: when physical identity matters.

49 Consider another example given in Figure 2: the `Rcfd.close`¹ function from the `Eio` [25]
 50 library. Essentially, it consists in protecting a file descriptor using reference counting.
 51 Similarly, it relies on atomically updating the `state` field using `Atomic.Loc.compare_and_set`².
 52 However, there is a complication. Indeed, we claim that the correctness of `close` derives from
 53 the fact that the `Open` state does not change throughout the lifetime of the data structure; it
 54 can be replaced by a `Closing` state but never by another `Open`. In other words, we want to
 55 say that 1) this `Open` is *physically unique* and 2) `Atomic.Loc.compare_and_set` therefore
 56 detects whether the data structure has flipped into the `Closing` state. In fact, this kind of
 57 property appears frequently in lockfree algorithms; it also occurs in the `Kcas` [20] library³.

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

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

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

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

1.0.0.3 A formalized OCaml fragment for the verification of concurrent algorithms.

These subtle aspects, illustrated through two realistic examples, justify the need for a faithful formal semantics of a fragment of OCAML tailored for the verification of concurrent algorithms. Ideally, of course, this fragment would include most of the language. However, the direct practical aim of this work—the verification of real-life libraries like SATURN [21]—led us to the following design philosophy: only include what is actually needed to express and reason about concurrent algorithms in a convenient way.

In this paper, we show how we have designed a practical framework, ZOO⁴, following this guideline. We review the works related to the verification of OCAML programs in Section 2. We describe our framework in Section 3. We detail the important features, including the treatment of physical equality, in Section 4. Finally we mention some side-contributions of this work, which are improvements to the OCAML language and implementation to better support lock-free concurrent programs.

2 Related work

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

2.0.0.1 Semi-automated verification.

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

In *non-foundational* automated verification, the tool and the external solvers it may rely on are part of the trusted computing base. It is the most common approach and has been widely applied in the literature [32, 27, 17, 12, 3, 13, 24, 29], including to OCAML by CAMELEER [28], which uses the GOSPEL specification language [8] and WHY3 [13].

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 [30] and RUST [14].

2.0.0.2 Non-automated verification.

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

The representation may be primitive, like Gallina for ROCQ. For pure programs, this is rather straightforward, *e.g.* in *hs-to-coq* [31]. For imperative programs, this is more

⁴ <https://github.com/clef-men/zoo>

```

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

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

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

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

```

■ Figure 2 `Rcfd.close` function from Eio [25]

97 challenging. One solution is to use a monad, *e.g.* in `coq-of-ocaml` [9], but it does not
 98 support concurrency.

99 The representation may be embedded, meaning the semantics of the language is formalized
 100 in the proof assistant. This is the path taken by some recent works [7, 15, 6] harnessing
 101 the power of separation logic, in particular the IRIS [18] concurrent separation logic. IRIS
 102 is a very important work for the verification of concurrent algorithms. It allows for a rich,
 103 customizable ghost state that makes it possible to design complex *concurrent protocols*. In
 104 our experience, for the lockfree algorithms we considered, there is simply no alternative.

105 The tool closest to our needs so far is CFML [7], which targets OCAML. However, CFML
 106 does not support concurrency and is not based on IRIS. The OSIRIS [11] framework, still
 107 under development, also targets OCAML and is based on IRIS. However, it does not support
 108 concurrency and it is arguably non-trivial to introduce it since the semantics uses interaction
 109 trees [33]—the question of how to handle concurrency in this context is a research subject.
 110 Furthermore, OSIRIS is not usable yet; its ambition to support a large fragment of OCAML
 111 makes it a challenge.

112

3 Zoo in practice

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

■ **Figure 3** ZOOLANG syntax (omitting mutually recursive toplevel functions)

113 In this section, we give an overview of the framework. We also provide a minimal example⁵
 114 demonstrating its use.

115 3.0.0.1 Language.

116 The core of ZOO is ZOOLANG: an untyped, ML-like, imperative, concurrent programming
 117 language that is fully formalized in ROCQ. Its semantics has been designed to match
 118 OCAML's.

119 ZOOLANG comes with a program logic based on IRIS: reasoning rules expressed in
 120 separation logic (including rules for the different constructs of the language) along with

⁵ <https://github.com/clef-men/zoo-demo>

ROCQ tactics that integrate into the IRIS proof mode [23, 22]. In addition, it supports DIAFRAME [26], enabling proof automation.

The ZOOLANG syntax is given in Figure 3⁶, omitting mutually recursive toplevel functions that are treated specifically. Expressions include standard constructs like booleans, integers, anonymous functions (that may be recursive), **let** bindings, sequence, unary and binary operators, conditionals, **for** loops, tuples. In any expression, one can refer to a ROCQ term representing a ZOOLANG value (of type **val**) using its ROCQ identifier. ZOOLANG is a deeply embedded language: variables (bound by functions and **let**) are quoted, represented as strings.

Data constructors (immutable memory blocks) are supported through two constructs: **\$C** represents a constant constructor (e.g. **\$None**), **'C** (e_1, \dots, e_n) represents a non-constant constructor (e.g. **'Some** (e)). Unlike OCAML, ZOOLANG has projections of the form $e.<proj>$ (e.g. $(e_1, e_2).<1>$), that can be used to obtain a specific component of a tuple or data constructor. ZOOLANG supports shallow pattern matching (patterns cannot be nested) on data constructors with an optional fallback case.

Mutable memory blocks are constructed using either the untagged record syntax $\{e_1, \dots, e_n\}$ or the tagged record syntax **'C** $\{e_1, \dots, e_n\}$. Reading a record field can be performed using $e.\{fld\}$ and writing to a record field using $e_1 \leftarrow \{fld\} e_2$. 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 $e_1 \leftarrow e_2$ writes into a reference. The syntax seemingly does not include constructs for arrays but they are supported through the **Array** standard module (e.g. **array_make**).

Parallelism is mainly supported through the **Domain** standard module (e.g. **domain_spawn**). Special constructs (**Xchg**, **CAS**, **FAA**), described in Section 4.5, are used to model atomic references.

The **Proph** and **Resolve** constructs are used to model *prophecy variables* [19], as described in Section 4.6.

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

3.0.0.2 Translation from OCaml to ZooLang.

While ZOOLANG lives in ROCQ, we want to verify OCAML programs. To connect them, we provide a tool to automatically translate OCAML source files⁷ into ROCQ files containing ZOOLANG code: **ocaml2zoo**. This tool can process entire **dune** projects, including many libraries.

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

As an example of what **ocaml2zoo** can generate, the **push** function from Section 1 is translated into:

```
Definition stack_push : val :=
  rec: "push" "t" "v" =>
    let: "old" := !"t" in
    let: "new_" := "v" :: "old" in
```

⁶ More precisely, it is the syntax of the surface language, including many ROCQ notations.

⁷ Actually, **ocaml2zoo** processes binary annotation files (**.cmt** files).

```

    if: ~ CAS "t".[contents] "old" "new_" then (
      domain_yield () ;;
      "push" "t" "v"
    ).

```

160 3.0.0.3 Specifications and proofs.

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

```

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

```

163 Here, we use a *logically atomic specification* [10], which has been proven [4] to be equivalent
 164 to *linearizability* [16] in sequentially consistent memory models.

165 Similarly to Hoare triples, the two assertions inside curly brackets represent the precondition
 166 and postcondition for the caller. For this particular operation, the postcondition is trivial.
 167 The `stack_inv t` precondition is the stack invariant. Intuitively, it asserts that t is a valid
 168 concurrent stack. More precisely, it enforces a set of logical constraints—a concurrent
 169 protocol—that t must respect at all times.

170 The other two assertions inside angle brackets represent the *atomic precondition* and
 171 *atomic postcondition*. They specify the linearization point of the operation: during the
 172 execution of `stack_push`, the abstract state of the stack held by `stack_model` is atomically
 173 updated from vs to $v :: vs$; in other words, v is atomically pushed at the top of the stack.

174 4 Zoo features

175 In this section, we review the main features of ZOO, starting with the most generic ones and
 176 then addressing those related to concurrency.

177 4.1 Algebraic data types

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

181 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).

```

182 Given this incantation, one may directly use the tags `Nil` and `Cons` in data constructors
 183 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.

```

184 The meaning of this incantation is not really important, as such notations can be generated
 185 by `ocaml2zoo`. Suffice it to say that it introduces the two tags in the `zoo_tag` custom entry,
 186 on which the notations for data constructors rely. The `in_type` term is needed to distinguish
 187 the tags of distinct data types; crucially, it cannot be simplified away by ROCQ, as this could
 188 lead to confusion during the reduction of expressions.

189 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".

```

190 4.2 Mutually recursive functions

191 ZOO supports non-recursive (`fun: $x_1 \dots x_n \Rightarrow e$`) and recursive (`rec: $f \ x_1 \dots x_n \Rightarrow e$`)
 192 functions but only *toplevel* mutually recursive functions. Indeed, it is non-trivial to properly
 193 handle mutual recursion: when applying a mutually recursive function, a naive approach
 194 would replace the recursive functions by their respective bodies, but this typically makes
 195 the resulting expression unreadable. To prevent it, the mutually recursive functions have
 196 to know one another so as to replace by the names instead of the bodies. We simulate this
 197 using some boilerplate that can be generated by `ocaml2zoo`. For instance, one may define
 198 two mutually recursive functions `f` and `g` as follows:

```

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

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

```


4.3 Standard library

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

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

4.4 Physical equality

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

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

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

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

If physical comparison returns `false`, the semantics of OCAML tells us essentially nothing: two immutable blocks may have distinct identities but same content. However, given this semantics, we cannot verify the `Rcfd` example of Section 1. To see why, consider the first `Atomic.compare_and_set` in the `close` function. If it fails, we expect to see a `Closing` state because we know there is only one `Open` state ever created, but we cannot prove it. To address it, we take another step back from OCAML's semantics by introducing the `Reveal` construct. When applied to an immutable memory block, `Reveal` yields the same block annotated with a logical identifier that can be interpreted as its abstract identity. The meaning of this identifier is: if physical comparison of two identified blocks returns `false`, the two identifiers are necessarily distinct. The underlying assumption that we make here—which is hopefully also correct in the current implementation of OCAML—is that the compiler may introduce sharing but not unsharing.

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

```

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

```

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

Structural equality is also supported. Due to space limitations, we do not describe it here but interested readers may refer to the ROCQ mechanization⁸.

4.5 Concurrent primitives

ZOO supports concurrent primitives both on atomic references (from `Atomic`) and atomic record fields (from `Atomic.Loc`⁹) according to the table below. The OCAML expressions listed in the left-hand column translate into the ZOO expressions in the right-hand column. Notice that an atomic location `[%atomic.loc e.f]` (of type `_ Atomic.Loc.t`) translates directly into `e.[f]`.

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

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

4.6 Prophecy variables

Lockfree algorithms exhibit complex behaviors. To tackle them, IRIS provides powerful mechanisms such as *prophecy variables* [19]. 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`.

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

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

272 5 Improving OCaml for concurrent lock-free programming

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

279 5.0.0.1 A reminder on OCaml attributes and extension points.

280 TODO

281 5.1 Atomic record fields

282 5.1.1 Before

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

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

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

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

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

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

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

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

5.1.2 Proposal(s)

We proposed a design for atomic record fields as an OCAML language change proposal: RFC #39¹⁰. Declaring a record field atomic simply requires an `[@atomic]` attribute – and could eventually become a proper keyword of the language.

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

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

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

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

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

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

¹⁰<https://github.com/ocaml/RFCs/pull/39>. Warning: this link is not anonymized.

- 323 1. Our first implementation¹¹ introduces a built-in type 'a Atomic.Loc.t for an atomic
 324 location that holds an element of type 'a, with a syntax extension [%atomic.loc <expr>.<field>]
 325 to construct such locations. Atomic primitives operate on values of type 'a Atomic.Loc.t,
 326 and they are exposed as functions of the module Atomic.Loc.
 327 For example, the standard library exposes

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

328 and users can write

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

329 where [%atomic.loc buf.front] has type int Atomic.Loc.t.

330 Internally, a value of type 'a Atomic.Loc.t can be represented as a pair of a record and
 331 an integer offset for the desired field, and the atomic.loc construction builds this pair
 332 in a well-typed manner. When a primitive of the Atomic.Loc module is applied to an
 333 atomic.loc expression, the compiler can optimize away the construction of the pair –
 334 but it would happen if there was an abstraction barrier between the construction and its
 335 use.

- 336 2. Our second implementation¹² introduces a built-in type ('r, 'a) Atomic.Field.t that
 337 denotes a field/index of type 'a within a record of type 'r, with a syntax extension
 338 [%atomic.loc <field>] to construct such field description, and atomic primitives in
 339 a module Atomic.Field, that need both the record value of type 'r and the field
 340 description.

341 For example, the standard library exposes

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

342 and users can write

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

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

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

348 Note that the previous type 'a Atomic.Loc.t can be reconstructed as a dependent pair
 349 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
```

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

¹¹<https://github.com/ocaml/ocaml/pull/13404>. Warning: this link is not anonymized.

¹²<https://github.com/ocaml/ocaml/pull/13707>. Warning: this link is not anonymized.

358 with those inline records. We have been working with OCAML maintainers to try to lift
 359 this limitation.

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

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

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

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

371 5.2 Atomic arrays

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

378 Our implementation of atomic arrays¹³ builds on top of the type `'a Atomic.Loc.t` we
 379 described in the previous section, and it relies on two new low-level primitives provided by
 380 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
```

381 The function `index` takes an array and an integer index within the array, and returns an
 382 atomic location into the corresponding element after performing a bound check. `unsafe_index`
 383 omits the boundcheck – additional performance at the cost of memory-safety – and allows to
 384 express the atomic counterpart of the unsafe operations `Array.unsafe_get` and `Array.unsafe_set`.
 385 The atomic primitives of the module `Atomic.Loc` can then be used on these indices; our
 386 implementation implements a library module on top of these primitives to provide a higher-
 387 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
```

388 5.3 Generative immutable constructors

389 TODO

¹³https://github.com/clef-men/ocaml/tree/atomic_array. Warning: this link is not anonymized.

6 Conclusion and future work

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

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

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

References

- 1 Clément Allain, Basile Clément, Alexandre Moine, and Gabriel Scherer. Snapshottable stores. *Proc. ACM Program. Lang.*, 8(ICFP):338–369, 2024. doi:10.1145/3674637.
- 2 Clément Allain, Vesa Karvonen, and Carine Morel. Saturn: a library of verified concurrent data structures for OCaml 5. In *OCaml Workshop 2024 - ICFP 2024*, Milan, Italy, September 2024. Armaël Guéneau and Sonja Heinze. URL: <https://inria.hal.science/hal-04681703>.
- 3 Vytautas Astrauskas, Aurel Bilý, Jonás Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. The prusti project: Formal verification for rust. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, volume 13260 of *Lecture Notes in Computer Science*, pages 88–108. Springer, 2022. doi:10.1007/978-3-031-06773-0_5.
- 4 Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. Theorems for free from separation logic specifications. *Proc. ACM Program. Lang.*, 5(ICFP):1–29, 2021. doi:10.1145/3473586.
- 5 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>.
- 6 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.
- 7 Arthur Charguéraud. *A Modern Eye on Separation Logic for Sequential Programs. (Un nouveau regard sur la Logique de Séparation pour les programmes séquentiels)*. 2023. URL: <https://tel.archives-ouvertes.fr/tel-04076725>.
- 8 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.
- 9 Guillaume Claret. `coq-of-ocaml`. URL: <https://github.com/formal-land/coq-of-ocaml>.
- 10 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.

- 439 11 Arnaud Daby-Seesaram, Jean-Marie Madiot, François Pottier, Remy Seassau, and Irene Yoon.
440 Osiris. URL: <https://gitlab.inria.fr/fpottier/osiris>.
- 441 12 Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: A foundry for
442 the deductive verification of rust programs. In Adrián Riesco and Min Zhang, editors,
443 *Formal Methods and Software Engineering - 23rd International Conference on Formal*
444 *Engineering Methods, ICFEM 2022, Madrid, Spain, October 24-27, 2022, Proceedings*,
445 volume 13478 of *Lecture Notes in Computer Science*, pages 90–105. Springer, 2022. doi:
446 10.1007/978-3-031-17244-1_6.
- 447 13 Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In
448 Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems -*
449 *22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint*
450 *Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24,*
451 *2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer,
452 2013. doi:10.1007/978-3-642-37036-6_8.
- 453 14 Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer.
454 Refinedrust: A type system for high-assurance verification of rust programs. *Proc. ACM*
455 *Program. Lang.*, 8(PLDI):1115–1139, 2024. doi:10.1145/3656422.
- 456 15 Léon Gondelman, Jonas Kastberg Hinrichsen, Mário Pereira, Amin Timany, and Lars Birkedal.
457 Verifying reliable network components in a distributed separation logic with dependent
458 separation protocols. *Proc. ACM Program. Lang.*, 7(ICFP):847–877, 2023. doi:10.1145/
459 3607859.
- 460 16 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent
461 objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.
- 462 17 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and
463 Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and java. In
464 Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors,
465 *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA,*
466 *April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages
467 41–55. Springer, 2011. doi:10.1007/978-3-642-20398-5_4.
- 468 18 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek
469 Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation
470 logic. *J. Funct. Program.*, 28:e20, 2018. doi:10.1017/S0956796818000151.
- 471 19 Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany,
472 Derek Dreyer, and Bart Jacobs. The future is ours: prophecy variables in separation logic.
473 *Proc. ACM Program. Lang.*, 4(POPL):45:1–45:32, 2020. doi:10.1145/3371113.
- 474 20 Vesa Karvonen. Kcas. URL: <https://github.com/ocaml-multicore/kcas>.
- 475 21 Vesa Karvonen and Carine Morel. Saturn. URL: [https://github.com/ocaml-multicore/](https://github.com/ocaml-multicore/saturn)
476 [saturn](https://github.com/ocaml-multicore/saturn).
- 477 22 Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser,
478 Amin Timany, Arthur Charguéraud, and Derek Dreyer. Mosel: a general, extensible modal
479 framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.*, 2(ICFP):77:1–
480 77:30, 2018. doi:10.1145/3236772.
- 481 23 Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order
482 concurrent separation logic. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings*
483 *of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL*
484 *2017, Paris, France, January 18-20, 2017*, pages 205–217. ACM, 2017. doi:10.1145/3009837.
485 3009855.
- 486 24 Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou,
487 Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear
488 ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1):286–315, 2023. doi:10.1145/3586037.
- 489 25 Anil Madhavapeddy and Thomas Leonard. Eio. URL: [https://github.com/](https://github.com/ocaml-multicore/eio)
490 [ocaml-multicore/eio](https://github.com/ocaml-multicore/eio).

- 491 **26** Ike Mulder, Robbert Krebbers, and Herman Geuvers. Diaframe: automated verification
 492 of fine-grained concurrent programs in iris. In Ranjit Jhala and Isil Dillig, editors, *PLDI*
 493 *'22: 43rd ACM SIGPLAN International Conference on Programming Language Design and*
 494 *Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 809–824. ACM, 2022.
 495 doi:10.1145/3519939.3523432.
- 496 **27** Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure
 497 for permission-based reasoning. In Alexander Pretschner, Doron Peled, and Thomas
 498 Hutzelmann, editors, *Dependable Software Systems Engineering*, volume 50 of *NATO Science*
 499 *for Peace and Security Series - D: Information and Communication Security*, pages 104–125.
 500 IOS Press, 2017. doi:10.3233/978-1-61499-810-5-104.
- 501 **28** Mário Pereira and António Ravara. Cameleer: A deductive verification tool for ocaml.
 502 In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd*
 503 *International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part*
 504 *II*, volume 12760 of *Lecture Notes in Computer Science*, pages 677–689. Springer, 2021.
 505 doi:10.1007/978-3-030-81688-9_31.
- 506 **29** Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and
 507 Neel Krishnaswami. CN: verifying systems C code with separation-logic refinement types.
 508 *Proc. ACM Program. Lang.*, 7(POPL):1–32, 2023. doi:10.1145/3571194.
- 509 **30** Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer,
 510 and Deepak Garg. Refinedc: automating the foundational verification of C code with refined
 511 ownership types. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM*
 512 *SIGPLAN International Conference on Programming Language Design and Implementation,*
 513 *Virtual Event, Canada, June 20-25, 2021*, pages 158–174. ACM, 2021. doi:10.1145/3453483.
 514 3454036.
- 515 **31** Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Total
 516 haskell is reasonable coq. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th*
 517 *ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los*
 518 *Angeles, CA, USA, January 8-9, 2018*, pages 14–27. ACM, 2018. doi:10.1145/3167092.
- 519 **32** Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and
 520 Jean Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*,
 521 23(4):402–451, 2013. doi:10.1017/S0956796813000142.
- 522 **33** Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce,
 523 and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq.
 524 *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020. doi:10.1145/3371119.