

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

Anonymous author

Anonymous affiliation

Anonymous author

Anonymous affiliation

Abstract

The release of OCAML 5, which introduced parallelism into the language, drove the need for safe and efficient concurrent data structures. New libraries like SATURN [31] aim at addressing this need. From the perspective of formal verification, this is an opportunity to apply and further state-of-the-art techniques to provide stronger guarantees.

We present a framework for verifying fine-grained concurrent OCAML 5 algorithms. Following a pragmatic approach, we support a limited but sufficient fragment of the language whose semantics has been carefully formalized to faithfully express such algorithms. Source programs are translated to a deeply-embedded language living inside ROCQ where they can be specified and verified using the IRIS [28] concurrent separation logic.

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases ROCQ, program verification, separation logic

Digital Object Identifier 10.4230/LIPIcs.ITP.2025.23

1 Introduction

Designing concurrent algorithms, in particular fine-grained concurrent algorithms, is a notoriously difficult task. Similarly, the formal verification of such algorithms is also difficult. It typically involves finding and reasoning about non-trivial linearization points [21, 29, 53, 54, 11].

In recent years, concurrent separation logic [5] has enabled significant progress in this area. In particular, the development of IRIS [28], a state-of-the-art mechanized *higher-order* concurrent separation logic with *user-defined ghost state*, has nourished a rich and successful line of works [29, 53, 54, 11, 6, 27, 48, 38, 37, 17, 43, 41, 40], dealing with external [54] and future-dependent [29, 53, 11] linearization points, relaxed memory [38, 37, 17, 43] and automation [41, 40].

Most of these works [29, 53, 54, 6, 27, 48, 41, 40] and many others [19, 45, 52, 35] rely on HEAPLANG [51], the canonical IRIS language. HEAPLANG is a concurrent, imperative, untyped, call-by-value functional language. To the best of our knowledge, it is currently the closest language to OCAML 5 in the IRIS ecosystem—we review the existing frameworks in Section 2. It has been extended to handle weak memory [38] and algebraic effects [18].

Although HEAPLANG is theoretically expressive enough to represent OCAML programs, our experience showed that it is fairly impractical when it comes to verifying large OCAML libraries. Indeed, it lacks basic abstractions such as algebraic data types (tuples, mutable and immutable records, variants) and mutually recursive functions. It also has very few standard data structures that can be directly reused. This view, we believe, is shared by many people in the IRIS community. Our first motivation in this work is therefore to fill this gap by providing a more practical OCAML-like verification language: ZOOLANG. This language consists in a subset of OCAML 5 extended with atomic record fields and equipped with a formal semantics and a program logic based on IRIS. We were influenced by the



© 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. 23:1–23:19

Leibniz International Proceedings in Informatics



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

45 PERENNIAL [8, 9, 10, 11] framework, which achieved similar goals for the GO language with
 46 a focus on crash-safety. As in PERENNIAL, we also provide a translator from OCAML to
 47 ZOOLANG: `ocaml2zoo`. We call the resulting framework ZOO.

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

56 In summary, we make the following contributions:

- 57 1. We present ZOOLANG, a convenient subset of OCAML 5 formalized in ROCQ (Sections 3
 58 and 4). ZOOLANG comes with a program logic based on IRIS and supports proof
 59 automation through DIAFRAME [41, 40].
- 60 2. We provide a translator from OCAML to ZOOLANG: `ocaml2zoo` (Section 3).
- 61 3. We formalize physical equality (Section 5) and structural equality (Section 6) in a faithful
 62 way. The careful analysis of these notions suggests a new OCAML feature: *generative*
 63 *constructors*.
- 64 4. We extend OCAML with *atomic record fields* and *atomic arrays* to ease the development
 65 of fine-grained concurrent algorithms (Section 7).
- 66 5. We verify realistic use cases (Section 5) involving physical equality: (1) Treiber stack [7],
 67 (2) a thread-safe wrapper around a file descriptor using reference-counting from the
 68 Eio [36] library.

69 **2 Related work**

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

72 **2.1 Semi-automated verification**

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

77 In *non-foundational* automated verification, the tool and the external solvers it may
 78 rely on are part of the trusted computing base. It is the most common approach and has
 79 been widely applied in the literature [50, 42, 26, 20, 3, 22, 34, 46], including to OCAML by
 80 CAMELEER [44], which uses the GOSPEL specification language [13] and WHY3 [22].

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

84 **2.2 Non-automated verification**

85 The verified program is translated, manually or in an automated way, into a representation
 86 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` [49]. For imperative programs, this is more challenging. One solution is to use a monad, *e.g.* in `coq-of-ocaml` [14], but it does not support concurrency.

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

3 Zoo in practice

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

■ **Figure 1** ZOOLANG syntax (omitting mutually recursive toplevel functions)

```

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

97 In this section, we give an overview of our framework. We also provide a minimal example¹
 98 demonstrating its use.

99 3.1 Language

100 The core of ZOO is ZOOLANG: a concurrent, imperative, untyped, functional programming
 101 language fully formalized in ROCQ. Its semantics has been designed to match OCAML's.

102 ZOOLANG comes with a program logic based on IRIS: reasoning rules expressed in
 103 separation logic (including rules for the different constructs of the language) along with
 104 ROCQ tactics that integrate into the IRIS proof mode [33, 32]. In addition, it supports
 105 DIAFRAME [41, 40], enabling proof automation.

106 The ZOOLANG syntax is given in Figure 1², omitting mutually recursive toplevel functions
 107 that are treated specifically. Expressions include standard constructs like booleans, integers,
 108 anonymous functions (that may be recursive), **let** bindings, sequence, unary and binary
 109 operators, conditionals, **for** loops, tuples. In any expression, one can refer to a ROCQ term
 110 representing a ZOOLANG value (of type **val**) using its ROCQ identifier. ZOOLANG is a deeply

¹ Non-anonymous link

² More precisely, it is the syntax of the surface language, including ROCQ notations.

111 embedded language: variables (bound by functions and `let`) are quoted, represented as
 112 strings.

113 Data constructors (immutable memory blocks) are supported through two constructs : `$C`
 114 represents a constant constructor (e.g. `$None`), `'C (e1, ..., en)` represents a non-constant
 115 constructor (e.g. `'Some(e)`). Unlike OCAML, ZOOLANG has projections of the form
 116 `e.<proj>` (e.g. `(e1, e2).<1>`), that can be used to obtain a specific component of a tuple or
 117 data constructor. ZOOLANG supports shallow pattern matching (patterns cannot be nested)
 118 on data constructors with an optional fallback case.

119 Mutable memory blocks are constructed using either the untagged record syntax `{e1, ..., en}`
 120 or the tagged record syntax `'C {e1, ..., en}`. Reading a record field can be performed using
 121 `e.{fld}` and writing to a record field using `e1 <-{fld} e2`. Pattern matching can also be used
 122 on mutable tagged blocks provided that cases do not bind anything—in other words, only
 123 the tag is examined, no memory access is performed. References are also supported through
 124 the usual constructs : `ref e` creates a reference, `!e` reads a reference and `e1 <- e2` writes
 125 into a reference. The syntax seemingly does not include constructs for arrays but they are
 126 supported through the `Array` standard module (e.g. `array_make`).

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

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

132 3.2 Translation from OCaml to ZooLang

133 While ZOOLANG lives in ROCQ, we want to verify OCAML programs. To connect them, we
 134 provide a tool to automatically translate OCAML source files³ into ROCQ files containing
 135 ZOOLANG code: `ocaml2zoo`. This tool can process entire `dune` projects, including many
 136 libraries.

137 The supported OCAML fragment includes: tuples, variants, records (including inline
 138 records), shallow `match`, atomic record fields, unboxed types, toplevel mutually recursive
 139 functions.

140 Consider, for example, the OCAML implementation of a concurrent stack [7] in Figure 2.
 141 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_yield () ;;
      "push" "t" "v"
    ).
  
```

142 3.3 Specifications and proofs

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

³ Actually, `ocaml2zoo` processes binary annotation files (`.cmt` files).

```

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.

```

Here, we use a *logically atomic specification* [15], which has been proven [4] to be equivalent to *linearizability* [25] in sequentially consistent memory models.

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

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

4 Zoo features

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

4.1 Algebraic data types

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

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

```

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

```

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

```

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

171 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".
```

172 4.2 Mutually recursive functions

173 ZOO supports non-recursive (`fun: $x_1 \dots x_n \Rightarrow e$`) and recursive (`rec: $f \ x_1 \dots x_n \Rightarrow e$`)
 174 functions but only *oplevel* mutually recursive functions. Indeed, it is non-trivial to properly
 175 handle mutual recursion: when applying a mutually recursive function, a naive approach
 176 would replace the recursive functions by their respective bodies, but this typically makes
 177 the resulting expression unreadable. To prevent it, the mutually recursive functions have
 178 to know one another so as to replace by the names instead of the bodies. We simulate this
 179 using some boilerplate that can be generated by `ocaml2zoo`. For instance, one may define
 180 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.
```

181 4.3 Standard library

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

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

190 4.4 Concurrent primitives

191 ZOO supports concurrent primitives both on atomic references (from `Atomic`) and atomic
 192 record fields (from `Atomic.Loc`⁴) according to the table below. The OCAML expressions
 193 listed in the left-hand column translate into the ZOO expressions in the right-hand column.
 194 Notice that an atomic location `[%atomic.loc e.f]` (of type `_ Atomic.Loc.t`) translates
 195 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>
196 <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>

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

201 4.5 Prophecy variables

202 Lockfree algorithms exhibit complex behaviors. To tackle them, IRIS provides powerful
 203 mechanisms such as *prophecy variables* [29]. Essentially, prophecy variables can be used to
 204 predict the future of the program execution and reason about it. They are key to handle
 205 *future-dependent linearization points*: linearization points that may or may not occur at a
 206 given location in the code depending on a future observation.

207 ZOO supports prophecy variables through the `Proph` and `Resolve` expressions—as in
 208 HEAPLANG, the canonical IRIS language. In OCAML, these expressions correspond to
 209 `Zoo.proph` and `Zoo.resolve`, that are recognized by `ocaml2zoo`.

210 5 Physical equality

211 5.0.0.1 Example 1: physical equality.

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

217 Indeed, the semantics of `Atomic.compare_and_set` involves *physical equality*: if the
 218 content of the atomic reference is physically equal to the expected value, it is atomically
 219 updated to the new value. Comparing physical equality is tricky and can be dangerous—this
 220 is why *structural equality* is often preferred—because the programmer has few guarantees

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


```

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

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

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

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

```

■ **Figure 3** `Rcfd.close` function from Eio [36]

221 about the *physical identity* of a value. In particular, the physical identity of a list, or
 222 more generally of an inhabitant of an algebraic data type, is not really specified. The only
 223 guarantee is: if two values are physically equal, they are also structurally equal. Apparently,
 224 we don't learn anything interesting when two values are physically distinct. Going back
 225 to our example, this is fortunately not an issue, since we always retry the operation when
 226 `Atomic.compare_and_set` returns `false`.

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

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

236 **5.0.0.2 Example 2: when physical identity matters.**

237 Consider another example given in Figure 3: the `Rcfd.close`⁵ function from the `Eio` [36]
 238 library. Essentially, it consists in protecting a file descriptor using reference counting.
 239 Similarly, it relies on atomically updating the `state` field using `Atomic.Loc.compare_and_set`⁶.
 240 However, there is a complication. Indeed, we claim that the correctness of `close` derives from
 241 the fact that the `Open` state does not change throughout the lifetime of the data structure; it
 242 can be replaced by a `Closing` state but never by another `Open`. In other words, we want to
 243 say that 1) this `Open` is *physically unique* and 2) `Atomic.Loc.compare_and_set` therefore
 244 detects whether the data structure has flipped into the `Closing` state. In fact, this kind of
 245 property appears frequently in lock-free algorithms; it also occurs in the `Kcas` [30] library⁷.

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

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

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

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

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

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

⁵ https://github.com/ocaml-multicore/eio/blob/main/lib_eio/unix/rcfd.ml

⁶ Here, we make use of atomic record fields that were recently introduced in OCAML.

⁷ <https://github.com/ocaml-multicore/kcas/blob/main/doc/gkmz-with-read-only-cmp-ops.md>

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

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

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

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

293 6 Structural equality

294 7 OCaml extensions for fine-grained concurrent programming

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

301 7.1 Atomic record fields

302 7.1.1 Before

303 OCAML 5 offers a type `'a Atomic.t` of atomic references exposing sequentially-consistent
 304 atomic operations. Data races on non-atomic mutable locations has a much weaker semantics
 305 and is generally considered a programming error. For example, the Michael-Scott concurrent
 306 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 }
```

307 Performance-minded concurrency experts dislike this representation, because `'a Atomic.t`
 308 introduces an indirection in memory: it is represented as a pointer to a block containing the
 309 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
| (Next _) as record -> Some (Obj.magic record : 'a node Atomic.t)

```

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

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

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

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

7.1.2 Proposal(s)

We proposed a design for atomic record fields as an OCAML language change proposal: RFC #39⁸. 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];
}

```

⁸ Non-anonymous link

```

mutable back: int [@atomic];
}

```

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

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

- 341 1. Our first implementation⁹ introduces a built-in type `'a Atomic.Loc.t` for an atomic
 342 location that holds an element of type `'a`, with a syntax extension `[%atomic.loc <expr>.<field>]`
 343 to construct such locations. Atomic primitives operate on values of type `'a Atomic.Loc.t`
 344 and they are exposed as functions of the module `Atomic.Loc`. For example, the standard
 345 library exposes:

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

346 and users can write:

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

347 where `[%atomic.loc buf.front]` has type `int Atomic.Loc.t`. Internally, a value of
 348 type `'a Atomic.Loc.t` can be represented as a pair of a record and an integer offset for
 349 the desired field, and the `atomic.loc` construction builds this pair in a well-typed manner.
 350 When a primitive of the `Atomic.Loc` module is applied to an `atomic.loc` expression, the
 351 compiler can optimize away the construction of the pair—but it would happen if there
 352 was an abstraction barrier between the construction and its use.

- 353 2. Our second implementation¹⁰ introduces a built-in type `('r, 'a) Atomic.Field.t` that
 354 denotes a field/index of type `'a` within a record of type `'r`, with a syntax extension
 355 `[%atomic.loc <field>]` to construct such field description, and atomic primitives in a
 356 module `Atomic.Field`, that need both the record value of type `'r` and the field description.
 357 For example, the standard library exposes:

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

358 and users can write:

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

359 where `[%atomic.field front]` has type `('a bag, int) Atomic.Loc.t`. Internally, a
 360 value of type `('r, 'a) Atomic.Field.t` is just an integer offset locating the field within
 361 the record: in exchange for a more complex type, we get a simpler data representation,
 362 that does not rely on specific compiler optimizations to generate efficient code, even
 363 across abstraction boundaries. Note that the previous type `'a Atomic.Loc.t` can be
 364 reconstructed as a dependent pair of a `'r` and a `('r, 'a) Atomic.Field.t`, which is
 365 expressible in OCAML as a GADT:

⁹ Non-anonymous link

¹⁰ Non-anonymous link

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

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

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

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

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

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

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 arrays¹¹ 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:

¹¹ Non-anonymous link

```

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

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 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 Nikolai 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 Nikolai 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*

- 448 *Implementation, OSDI 2021, July 14-16, 2021*, pages 423–439. USENIX Association, 2021.
 449 URL: <https://www.usenix.org/conference/osdi21/presentation/chajed>.
- 450 10 Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nikolai Zeldovich.
 451 Verifying the daisynfs concurrent and crash-safe file system with sequential reasoning. In
 452 Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating
 453 Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages
 454 447–463. USENIX Association, 2022. URL: [https://www.usenix.org/conference/osdi22/
 455 presentation/chajed](https://www.usenix.org/conference/osdi22/presentation/chajed).
- 456 11 Yun-Sheng Chang, Ralf Jung, Upamanyu Sharma, Joseph Tassarotti, M. Frans Kaashoek,
 457 and Nikolai Zeldovich. Verifying vmvcc, a high-performance transaction library using multi-
 458 version concurrency control. In Roxana Geambasu and Ed Nightingale, editors, *17th USENIX
 459 Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA,
 460 July 10-12, 2023*, pages 871–886. USENIX Association, 2023. URL: [https://www.usenix.
 461 org/conference/osdi23/presentation/chang](https://www.usenix.org/conference/osdi23/presentation/chang).
- 462 12 Arthur Charguéraud. *A Modern Eye on Separation Logic for Sequential Programs. (Un
 463 nouveau regard sur la Logique de Séparation pour les programmes séquentiels)*. 2023. URL:
 464 <https://tel.archives-ouvertes.fr/tel-04076725>.
- 465 13 Arthur Charguéraud, Jean-Christophe Filliâtre, Cláudio Lourenço, and Mário Pereira.
 466 GOSPEL - providing ocaml with a formal specification language. In Maurice H. ter
 467 Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods - The Next 30
 468 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*,
 469 volume 11800 of *Lecture Notes in Computer Science*, pages 484–501. Springer, 2019. doi:
 470 10.1007/978-3-030-30942-8_29.
- 471 14 Guillaume Claret. coq-of-ocaml. URL: <https://github.com/formal-land/coq-of-ocaml>.
- 472 15 Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for
 473 time and data abstraction. In Richard E. Jones, editor, *ECOOP 2014 - Object-Oriented
 474 Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014.
 475 Proceedings*, volume 8586 of *Lecture Notes in Computer Science*, pages 207–231. Springer,
 476 2014. doi:10.1007/978-3-662-44202-9_9.
- 477 16 Arnaud Daby-Seesaram, Jean-Marie Madiot, François Pottier, Remy Seassau, and Irene Yoon.
 478 Osiris. URL: <https://gitlab.inria.fr/fpottier/osiris>.
- 479 17 Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Thuan Nguyen, William Mansky, Jeehoon
 480 Kang, and Derek Dreyer. Compass: strong and compositional library specifications in relaxed
 481 memory separation logic. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM
 482 SIGPLAN International Conference on Programming Language Design and Implementation,
 483 San Diego, CA, USA, June 13 - 17, 2022*, pages 792–808. ACM, 2022. doi:10.1145/3519939.
 484 3523451.
- 485 18 Paulo Emílio de Vilhena and François Pottier. A separation logic for effect handlers. *Proc.
 486 ACM Program. Lang.*, 5(POPL):1–28, 2021. doi:10.1145/3434314.
- 487 19 Paulo Emílio de Vilhena, François Pottier, and Jacques-Henri Jourdan. Spy game: verifying
 488 a local generic solver in iris. *Proc. ACM Program. Lang.*, 4(POPL):33:1–33:28, 2020. doi:
 489 10.1145/3371101.
- 490 20 Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: A foundry for
 491 the deductive verification of rust programs. In Adrián Riesco and Min Zhang, editors,
 492 *Formal Methods and Software Engineering - 23rd International Conference on Formal
 493 Engineering Methods, ICFEM 2022, Madrid, Spain, October 24-27, 2022, Proceedings*,
 494 volume 13478 of *Lecture Notes in Computer Science*, pages 90–105. Springer, 2022. doi:
 495 10.1007/978-3-031-17244-1_6.
- 496 21 Brijesh Dongol and John Derrick. Verifying linearisability: A comparative survey. *ACM
 497 Comput. Surv.*, 48(2):19:1–19:43, 2015. doi:10.1145/2796550.
- 498 22 Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In
 499 Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems -*

- 500 *22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint*
 501 *Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24,*
 502 *2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer,
 503 2013. doi:10.1007/978-3-642-37036-6_8.
- 504 23 Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer.
 505 Refinedrust: A type system for high-assurance verification of rust programs. *Proc. ACM*
 506 *Program. Lang.*, 8(PLDI):1115–1139, 2024. doi:10.1145/3656422.
- 507 24 Léon Gondelman, Jonas Kastberg Hinrichsen, Mário Pereira, Amin Timany, and Lars Birkedal.
 508 Verifying reliable network components in a distributed separation logic with dependent
 509 separation protocols. *Proc. ACM Program. Lang.*, 7(ICFP):847–877, 2023. doi:10.1145/
 510 3607859.
- 511 25 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent
 512 objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.
- 513 26 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and
 514 Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and java. In
 515 Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors,
 516 *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA,*
 517 *April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages
 518 41–55. Springer, 2011. doi:10.1007/978-3-642-20398-5_4.
- 519 27 Jaehwang Jung, Janggun Lee, Jaemin Choi, Jaewoo Kim, Sunho Park, and Jeehoon Kang.
 520 Modular verification of safe memory reclamation in concurrent separation logic. *Proc. ACM*
 521 *Program. Lang.*, 7(OOPSLA2):828–856, 2023. doi:10.1145/3622827.
- 522 28 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek
 523 Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation
 524 logic. *J. Funct. Program.*, 28:e20, 2018. doi:10.1017/S0956796818000151.
- 525 29 Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany,
 526 Derek Dreyer, and Bart Jacobs. The future is ours: prophecy variables in separation logic.
 527 *Proc. ACM Program. Lang.*, 4(POPL):45:1–45:32, 2020. doi:10.1145/3371113.
- 528 30 Vesa Karvonen. Kcas. URL: <https://github.com/ocaml-multicore/kcas>.
- 529 31 Vesa Karvonen and Carine Morel. Saturn. URL: [https://github.com/ocaml-multicore/](https://github.com/ocaml-multicore/saturn)
 530 [saturn](https://github.com/ocaml-multicore/saturn).
- 531 32 Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser,
 532 Amin Timany, Arthur Charguéraud, and Derek Dreyer. Mosel: a general, extensible modal
 533 framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.*, 2(ICFP):77:1–
 534 77:30, 2018. doi:10.1145/3236772.
- 535 33 Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order
 536 concurrent separation logic. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings*
 537 *of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL*
 538 *2017, Paris, France, January 18-20, 2017*, pages 205–217. ACM, 2017. doi:10.1145/3009837.
 539 3009855.
- 540 34 Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou,
 541 Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear
 542 ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1):286–315, 2023. doi:10.1145/3586037.
- 543 35 Anton Lorenzen, Daan Leijen, Wouter Swierstra, and Sam Lindley. The functional essence
 544 of imperative binary search trees. *Proc. ACM Program. Lang.*, 8(PLDI):518–542, 2024.
 545 doi:10.1145/3656398.
- 546 36 Anil Madhavapeddy and Thomas Leonard. Eio. URL: [https://github.com/](https://github.com/ocaml-multicore/eio)
 547 [ocaml-multicore/eio](https://github.com/ocaml-multicore/eio).
- 548 37 Glen Mével and Jacques-Henri Jourdan. Formal verification of a concurrent bounded queue in a
 549 weak memory model. *Proc. ACM Program. Lang.*, 5(ICFP):1–29, 2021. doi:10.1145/3473571.

- 550 38 Glen Mével, Jacques-Henri Jourdan, and François Pottier. Cosmo: a concurrent separation
551 logic for multicore ocaml. *Proc. ACM Program. Lang.*, 4(ICFP):96:1–96:29, 2020. doi:
552 10.1145/3408978.
- 553 39 Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking
554 concurrent queue algorithms. In James E. Burns and Yoram Moses, editors, *Proceedings of*
555 *the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia,*
556 *Pennsylvania, USA, May 23-26, 1996*, pages 267–275. ACM, 1996. doi:10.1145/248052.
557 248106.
- 558 40 Ike Mulder and Robbert Krebbers. Proof automation for linearizability in separation logic.
559 *Proc. ACM Program. Lang.*, 7(OOPSLA1):462–491, 2023. doi:10.1145/3586043.
- 560 41 Ike Mulder, Robbert Krebbers, and Herman Geuvers. Diaframe: automated verification
561 of fine-grained concurrent programs in iris. In Ranjit Jhala and Isil Dillig, editors, *PLDI*
562 *'22: 43rd ACM SIGPLAN International Conference on Programming Language Design and*
563 *Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 809–824. ACM, 2022.
564 doi:10.1145/3519939.3523432.
- 565 42 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure
566 for permission-based reasoning. In Alexander Pretschner, Doron Peled, and Thomas
567 Hutzelmann, editors, *Dependable Software Systems Engineering*, volume 50 of *NATO Science*
568 *for Peace and Security Series - D: Information and Communication Security*, pages 104–125.
569 IOS Press, 2017. doi:10.3233/978-1-61499-810-5-104.
- 570 43 Sunho Park, Jaewoo Kim, Ike Mulder, Jaehwang Jung, Janggun Lee, Robbert Krebbers, and
571 Jeehoon Kang. A proof recipe for linearizability in relaxed memory separation logic. *Proc.*
572 *ACM Program. Lang.*, 8(PLDI):175–198, 2024. doi:10.1145/3656384.
- 573 44 Mário Pereira and António Ravara. Cameleer: A deductive verification tool for ocaml.
574 In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd*
575 *International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part*
576 *II*, volume 12760 of *Lecture Notes in Computer Science*, pages 677–689. Springer, 2021.
577 doi:10.1007/978-3-030-81688-9_31.
- 578 45 François Pottier, Armaël Guéneau, Jacques-Henri Jourdan, and Glen Mével. Thunks and
579 debits in separation logic with time credits. *Proc. ACM Program. Lang.*, 8(POPL):1482–1508,
580 2024. doi:10.1145/3632892.
- 581 46 Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and
582 Neel Krishnaswami. CN: verifying systems C code with separation-logic refinement types.
583 *Proc. ACM Program. Lang.*, 7(POPL):1–32, 2023. doi:10.1145/3571194.
- 584 47 Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer,
585 and Deepak Garg. Refinedc: automating the foundational verification of C code with refined
586 ownership types. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM*
587 *SIGPLAN International Conference on Programming Language Design and Implementation,*
588 *Virtual Event, Canada, June 20-25, 2021*, pages 158–174. ACM, 2021. doi:10.1145/3453483.
589 3454036.
- 590 48 Thomas Somers and Robbert Krebbers. Verified lock-free session channels with linking. *Proc.*
591 *ACM Program. Lang.*, 8(OOPSLA2):588–617, 2024. doi:10.1145/3689732.
- 592 49 Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Total
593 haskell is reasonable coq. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th*
594 *ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los*
595 *Angeles, CA, USA, January 8-9, 2018*, pages 14–27. ACM, 2018. doi:10.1145/3167092.
- 596 50 Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and
597 Jean Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*,
598 23(4):402–451, 2013. doi:10.1017/S0956796813000142.
- 599 51 Iris Development Team. The coq mechanization of iris. URL: <https://gitlab.mpi-sws.org/iris/iris/>.
600

- 601 52 Amin Timany, Armaël Guéneau, and Lars Birkedal. The logical essence of well-bracketed
602 control flow. *Proc. ACM Program. Lang.*, 8(POPL):575–603, 2024. doi:10.1145/3632862.
- 603 53 Simon Friis Vindum and Lars Birkedal. Contextual refinement of the michael-scott queue
604 (proof pearl). In Catalin Hritcu and Andrei Popescu, editors, *CPP '21: 10th ACM SIGPLAN
605 International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January
606 17-19, 2021*, pages 76–90. ACM, 2021. doi:10.1145/3437992.3439930.
- 607 54 Simon Friis Vindum, Dan Frumin, and Lars Birkedal. Mechanized verification of a fine-
608 grained concurrent queue from meta’s folly library. In Andrei Popescu and Steve Zdancewic,
609 editors, *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs
610 and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, pages 100–115. ACM, 2022.
611 doi:10.1145/3497775.3503689.