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

ANONYMOUS AUTHOR(S)

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

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

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

1 INTRODUCTION

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

Shared-memory concurrency is a difficult programming domain, and existing ecosystems (C++, Java, Haskell, Rust, Go...) took decades to evolve comprehensive libraries of concurrent abstractions and data structures. In the last couple years, a handful of contributors to the OCaml system have been implementing basic libraries for concurrent and parallel programs in OCaml, in particular `Saturn` [Karvonen and Morel 2024], a library of lock-free thread-safe data structures (stacks, queues, a work-stealing dequeue, a skip list, a bag and a hash table), `Eio` [Madhavapeddy and Leonard 2024], a library of asynchronous IO and structured concurrency, and `Kcas` [Karvonen 2024], a library offering a software-transactional-memory abstraction for users to build safe yet efficient thread-safe data structures.

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

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

OCaml language features. When studying the new codebases of concurrent and parallel data structures, we found a variety of unsafe idioms, working around expressivity or performance limitations with the OCaml language support for lock-free concurrent data structures. In particular, the support for *atomic references* in the OCaml library proved inadequate, as idiomatic concurrent

data-structures need the more expressive feature of *atomic record fields*. We designed an extension of OCaml with atomic record fields, implemented it as a an experimental compiler variant, and succeeded in getting it integrated in the upstream OCaml compiler: it should be available as part of OCaml 5.4, which is not yet released at the time of writing.

Verification tools for concurrent programs. The state-of-the-art approach for mechanized verification of fine-grained concurrent algorithms is to use Iris [Jung, Krebbers, Jourdan, Bizjak, Birkedal and Dreyer 2018], a state-of-the-art mechanized *higher-order* concurrent separation logic with *user-defined ghost state*. Its expressivity allows to precisely capture delicate invariants, and to reason about the linearization points of fine-grained concurrent algorithms (including external [Vindum, Frumin and Birkedal 2022] and future-dependent [Chang, Jung, Sharma, Tassarotti, Kaashoek and Zeldovich 2023; Jung, Lepigre, Parthasarathy, Rapoport, Timany, Dreyer and Jacobs 2020; Vindum and Birkedal 2021] linearization). Iris provides a generic mechanism to define programming languages and program logics for them. Much of the existing Iris concurrent verification work has been performed in HeapLang, the exemplar Iris language; a concurrent, imperative, untyped, call-by-value functional language.

To the best of our knowledge, it is currently the closest language to OCaml 5 in the Iris ecosystem — we review the existing frameworks in **??**. We started our verification effort in HeapLang, but it eventually proved impractical to verify realistic OCaml libraries. Indeed, it lacks basic abstractions such as algebraic data types (tuples, mutable and immutable records, variants) and mutually recursive functions. Verifying OCaml programs in HeapLang requires difficult translation choices and introduces various encodings, to the point that the relation between the source and verified programs can become difficult to maintain and reason about. It also has very few standard data structures that can be directly reused. This view, we believe, is shared by many people in the Iris community.

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

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

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

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

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

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

Contributions. In summary, we claim the following contributions:

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

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

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

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

2 ZOO IN PRACTICE

2.1 Language

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

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

The ZooLang syntax is given in Figure 1[1], omitting mutually recursive toplevel functions that are treated specially. Expressions include standard constructs like booleans, integers, anonymous functions (that may be recursive), applications, **let** bindings, sequence, unary and binary operators, conditionals, **for** loops, tuples. In any expression, one can refer to a Rocq term representing a ZooLang value (of type `val`) using its Rocq identifier. ZooLang is deeply embedded: variables (bound by functions and **let**) are quoted as strings.

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

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

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

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

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

2.2 Translation from OCaml to ZooLang

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

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

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

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

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

```
type 'a t = 'a list Atomic.t
let create () = Atomic.make []

let rec push t v =
  let old = Atomic.get t in
  let new_ = v :: old in
  if not (Atomic.compare_and_set t old new_) then (
    Domain.cpu_relax () ;
    push t v
  )

let rec pop t =
  match Atomic.get t with
  | [] -> None
  | v :: new_ as old ->
      if Atomic.compare_and_set t old new_ then (
        Some v
      ) else (
        Domain.cpu_relax () ;
        pop t
      )
```

Fig. 2. Implementation of a concurrent stack

```
    domain_cpu_relax () ;;
    "push" "t" "v" ).
```

2.3 Specifications and proofs

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

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

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

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

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

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

3 ZOO FEATURES

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

3.1 Algebraic data types

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

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

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

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

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

```
Definition map : val :=
  rec: "map" "fn" "t" =>
    match: "t" with
    | Nil => §Nil
    | Cons "x" "t" =>
        let: "y" := "fn" "x" in
        'Cons( "y", "map" "fn" "t" )
    end.
```

Similarly, one may define a record-like type with two mutable fields `f1` and `f2`:

```
Notation "'f1'" := (in_type "t" 0) (in custom zoo_field).
Notation "'f2'" := (in_type "t" 1) (in custom zoo_field).

Definition swap : val :=
  fun: "t" =>
    let: "f1" := "t".{f1} in
    "t" <-{f1} "t".{f2} ;; "t" <-{f2} "f1".
```

3.2 Mutually recursive functions

Zoo supports non-recursive (`fun:` $x_1 \ldots x_n$ `=>` e) and recursive (`rec:` $f\ x_1 \ldots x_n$ `=>` e) functions but only *toplevel* mutually recursive functions. It is non-trivial to properly handle mutual recursion: when applying a mutually recursive function, a naive approach would replace calls to sibling functions by their respective bodies, but this typically makes the resulting expression unreadable. To prevent it, the mutually recursive functions have to know one another to preserve their names

during β-reduction. We simulate this using some boilerplate that can be generated by `ocaml2zoo`. For instance, one may define two mutually recursive functions `f` and `g` as follows:

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

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

3.3 Standard library

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

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

3.4 Concurrent primitives

Zoo supports concurrent primitives both on atomic references (from **`Atomic`**) and atomic record fields (from **`Atomic.Loc`**[3]) according to the table below. The OCaml expressions listed in the left-hand column translate into the Zoo expressions in the right-hand column. Notice that an atomic location `[%atomic.loc` e`.`f`]` (of type `_ Atomic.Loc.t`) translates directly into e`.[`f`]`.

OCaml	Zoo
`Atomic``.get` e	`!`e
`Atomic``.set` e_1 e_2	e_1 `<-` e_2
`Atomic``.exchange` e_1 e_2	`Xchg` e_1`.[contents]` e_2
`Atomic``.compare_and_set` e_1 e_2 e_3	`CAS` e_1`.[contents]` e_2 e_3
`Atomic``.fetch_and_add` e_1 e_2	`FAA` e_1`.[contents]` e_2
`Atomic.Loc``.exchange [%atomic.loc` e_1`.`f`]` e_2	`Xchg` e_1`.[`f`]` e_2
`Atomic.Loc``.compare_and_set [%atomic.loc` e_1`.`f`]` e_2 e_3	`CAS` e_1`.[`f`]` e_2 e_3
`Atomic.Loc``.fetch_and_add [%atomic.loc` e_1`.`f`]` e_2	`FAA` e_1`.[`f`]` e_2

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

3.5 Prophecy variables

Lock-free algorithms exhibit complex behaviors. To tackle them, Iris provides powerful mechanisms such as *prophecy variables* [Jung, Lepigre, Parthasarathy, Rapoport, Timany, Dreyer and Jacobs 2020]. Essentially, prophecy variables can be used to predict the future of the program execution

[3]The **`Atomic.Loc`** module is part of the PR that implements atomic record fields (see Section 8).

and reason about it. They are key to handle *future-dependent linearization points*: linearization points that may or may not occur at a given location in the code depending on a future observation.

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

4 STANDARD DATA STRUCTURES

To save users from reinventing the wheel, we provide a library of verified standard data structures — more or less a subset of the OCaml standard library. Most of these data structures[4] are completely reimplemented in Zoo and axiom-free, including the `Array`[5] module.

Sequential data structures. We provide verified implementations of various sequential data structures: array, dynamic array (vector), list, stack, queue (bounded and unbounded), double-ended queue. We claim that the proven specifications are modular and practical. In fact, most of these data structures have already been used to verify more complex ones — we present some in Section 5 and Section 7. Especially, we developed an extensive collection of flexible specifications for the iterators of the `Array` and `List` modules. Remarkably, our formalization of `Array` features different (fractional) predicates to express the ownership of either an entire array, a slice or even a circular slice — we use it to verify algorithms involving circular arrays, *e.g.* Chase-Lev working-stealing queue [Chase and Lev 2005] as presented in Section 7.4.

Concurrent data structures. We provide verified implementations of various concurrent data structures: domain[6] (including domain-local storage), mutex, semaphore, condition variable, write-once variable (also known as *ivar*), atomic array. Note that there is currently no `Atomic_array` module in the OCaml standard library, but we are planning to propose it.

5 PERSISTENT DATA STRUCTURES

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

Currently, verification of purely functional programs is conducted in ZooLang, which is deeply embedded inside Rocq. However, in the future, it would be desirable to be able to verify them directly in Rocq, through a translation to Gallina. Similarly to Hacspec [Haselwarter, Hvass, Hansen, Winterhalter, Hritcu and Spitters 2024], these two translations would come with a generated proof of equivalence.

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

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

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

8 OCAML EXTENSIONS FOR FINE-GRAINED CONCURRENT PROGRAMMING

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

8.1 Atomic record fields

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

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

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

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

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

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

It is possible to statically forbid passing `Nil` to `as_atomic` to avoid error handling, by turning `'a node` into a GADT indexed over a type-level representation of its head constructor. Examples of this pattern can be found in the `Kcas` [Karvonen 2024] 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.

8.1.1 Our atomic fields proposal. We proposed a design for atomic record fields as an OCaml language change proposal: RFC #39[7]. Declaring a record field atomic simply requires an `[@atomic]` attribute—and could eventually become a proper keyword of the language.

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

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

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

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

Our proposed implementation[8] introduces a built-in type `'a Atomic.Loc.t` for an atomic location that holds an element of type `'a`, with a syntax extension `[%atomic.loc <expr>.<field>]` to construct such locations. Atomic primitives operate on values of type `'a Atomic.Loc.t`, and they are exposed as functions of the module `Atomic.Loc`.

For example, the standard library exposes

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

and users can write:

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

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

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

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

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

8.2 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

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

concurrent bag of type `'a bag` used a backing array of type `'a Atomic.t array`, which contains more indirections than may be desirable, as each array element is a pointer to a block containing the value of type `'a`, instead of storing the value of type `'a` directly in the array.

Our implementation of atomic arrays[9] builds on top of the type `'a Atomic.Loc.t` we described in the previous section, and it relies on two new low-level primitives provided by the compiler:

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

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

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

REFERENCES

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

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

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

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

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

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

Philipp G. Haselwarter, Benjamin Salling Hvass, Lasse Letager Hansen, Théo Winterhalter, Catalin Hritcu, and Bas Spitters. 2024. The Last Yard: Foundational End-to-End Verification of High-Speed Cryptography. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2024, London, UK, January 15-16, 2024*, Amin Timany, Dmitriy Traytel, Brigitte Pientka, and Sandrine Blazy (Eds.). ACM, 30–44. doi:10.1145/3636501.3636961

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

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

Vesa Karvonen. 2024. Kcas. https://github.com/ocaml-multicore/kcas

Vesa Karvonen and Carine Morel. 2024. Saturn. https://github.com/ocaml-multicore/saturn

Anil Madhavapeddy and Thomas Leonard. 2024. Eio. https://github.com/ocaml-multicore/eio

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

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

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

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

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