ZOO: A framework for the verification of concurrent OCAML 5 programs using separation logic

Clément Allain

INRIA

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 [28] 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 COQ where they can be specified and verified using the IRIS [8] concurrent separation logic.

1 Introduction

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

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

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

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

```ocaml
type 'a t =
  'a list Atomic.t

let create () =
  Atomic.make []

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

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

Figure 1. Implementation of a concurrent stack

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

Example 2: when physical identity matters. Consider another example given in Figure 2: the `Rcfd.close`[1] function from the Eio [29] library. Essentially, it consists in protecting a file descriptor using reference counting. Similarly, it relies on atomically updating the `state` field using `Atomic.Loc.compare_and_set`[2]. However, there is a complication. Indeed, we claim that the correctness of `close` derives from the fact that the `Open` state does not change throughout the lifetime of the data structure; it can be replaced by a `Closing` state but never by another `Open`. In other words, we want to say that 1) this `Open` is *physically unique* and 2) `Atomic.Loc.compare_and_set` therefore detects whether the data structure has flipped into the `Closing` state. In fact, this kind of property appears frequently in lockfree algorithms; it also occurs in the Kcas [27] library[3].

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

[1] https://github.com/ocaml-multicore/eio/blob/main/lib_eio/unix/rcfd.ml
[2] Here, we make use of atomic record fields that were recently introduced in OCaml.
[3] https://github.com/ocaml-multicore/kcas/blob/main/doc/gkmz-with-read-only-cmp-ops.md

```
type state =
  | Open of Unix.file_descr
  | Closing of (unit -> unit)
type t =
  { mutable ops: int [@atomic];
    mutable state: state [@atomic];
  }

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

Figure 2. `Rcfd.close` function from the `Eio` [29] library

A formalized OCaml fragment for the verification of concurrent algorithms. These subtle aspects, illustrated through two realistic examples, justify the need for a faithful formal semantics of a fragment of OCaml tailored for the verification of concurrent algorithms. Ideally, of course, this fragment would include most of the language. However, the direct practical aim of this work—the verification of real-life libraries like Saturn [28]—led us to the following design philosophy: only include what is actually needed to express and reason about concurrent algorithms in a convenient way.

In this paper, we show how we have designed a practical framework, Zoo, following this guideline. We review the works related to the verification of OCaml programs in Section 2; we describe our framework in Section 3; we detail the important features, including the treatment of physical equality, in Section 4 before concluding.

2 Related work

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

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

In *non-foundational* automated verification, the tool and the external solvers it may rely on are part of the trusted computing base. It is the most common approach and has been widely applied in the literature [5, 7, 3, 19, 18, 4, 22, 23], including to OCaml by Cameleer [16], which uses the Gospel specification language [12] and Why3 [4].

In *foundational* automated verification, the proofs are checked by a proof assistant like Coq, meaning the automation does not have to be trusted. To our knowledge, it has been applied to C [17] and Rust [26].

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

The representation may be primitive, like Gallina for Coq. For pure programs, this is rather straightforward, *e.g.* in `hs-to-coq` [10]. For imperative programs, this is more challenging. One solution is to use a monad, *e.g.* in `coq-of-ocaml` [24], 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 [20, 21, 11] harnessing the power of separation logic, in particular the Iris [8] concurrent separation logic. Iris is a very important work for the verification of concurrent algorithms. It allows for a rich, customizable ghost state that makes it possible to design complex *concurrent protocols*. In our experience, for the lockfree algorithms we considered, there is simply no alternative.

The tool closest to our needs so far is CFML [20], which targets OCaml. However, CFML does not support concurrency and is not based on Iris. The Osiris [25] framework, still under development, also targets OCaml and is based on Iris. However, it does not support concurrency and it is arguably non-trivial to introduce it since the semantics uses interaction trees [14]—the question of how to handle concurrency in this context is a research subject. Furthermore, Osiris is not usable yet; its ambition to support a large fragment of OCaml makes it a challenge.

3 Zoo in practice

Before describing the salient features of our language, Zoo, in Section 4, we give an overview of the framework.

From OCaml to Zoo. First, OCaml source files are translated into Zoo by the `ocaml2zoo` tool. The Zoo syntax is given in Figure 3[4], omitting mutually recursive toplevel functions that are treated specifically. Essentially, Zoo is an untyped, ML-like, imperative, concurrent programming language. The supported OCaml fragment includes: shallow `match`, ADTs, records, inline records, atomic record fields, unboxed types, toplevel mutually recursive functions.

For instance, the `push` function from Section 1 is translated into:

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

Specifications and proofs. Second, the user writes specifications for the translated functions and prove them using the Iris proof mode [9].

For instance, the specification for the `stack_push` function would be:

[4] More precisely, it is the syntax of the surface language, including many Coq notations.

$$
\begin{array}{llcl}
\textrm{Coq term} & t & & \\
\textrm{constructor} & C & & \\
\textrm{projection} & \mathit{proj} & & \\
\textrm{record field} & \mathit{fld} & & \\
\textrm{identifier} & s, f & \in & \textrm{String} \\
\textrm{integer} & n & \in & \mathbb{Z} \\
\textrm{boolean} & b & \in & \mathbb{B} \\
\textrm{binder} & x & ::= & \texttt{<>} \mid s \\
\textrm{unary operator} & \oplus & ::= & \sim \mid \texttt{-} \\
\textrm{binary operator} & \otimes & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{`quot`} \mid \texttt{`rem`} \\
 & & \mid & \texttt{<=} \mid \texttt{<} \mid \texttt{>=} \mid \texttt{>} \mid \texttt{=} \mid \neq \mid \texttt{==} \mid \texttt{!=} \\
 & & \mid & \texttt{and} \mid \texttt{or} \\
\textrm{expression} & e & ::= & t \mid s \mid \texttt{\#}n \mid \texttt{\#}b \\
 & & \mid & \texttt{fun:}\ x_1 \ldots x_n \texttt{ => } e \mid \texttt{rec:}\ f\ x_1 \ldots x_n \texttt{ => } e \\
 & & \mid & \texttt{let:}\ x \texttt{ := } e_1 \texttt{ in } e_2 \mid e_1 \texttt{ ;; } e_2 \\
 & & \mid & \texttt{let:}\ f\ x_1 \ldots x_n \texttt{ := } e_1 \texttt{ in } e_2 \mid \texttt{letrec:}\ f\ x_1 \ldots x_n \texttt{ := } e_1 \texttt{ in } e_2 \\
 & & \mid & \texttt{let: `}C\ x_1 \ldots x_n \texttt{ := } e_1 \texttt{ in } e_2 \mid \texttt{let:}\ x_1\texttt{,}\ldots\texttt{,}x_n \texttt{ := } e_1 \texttt{ in } e_2 \\
 & & \mid & \oplus e \mid e_1 \otimes e_2 \\
 & & \mid & \texttt{if:}\ e_0 \texttt{ then } e_1\ (\texttt{else } e_2)^? \mid \texttt{ifnot:}\ e_0 \texttt{ then } e_1 \\
 & & \mid & \texttt{for:}\ x \texttt{ := } e_1 \texttt{ to } e_2 \texttt{ begin } e_3 \texttt{ end} \\
 & & \mid & \S C \mid \texttt{`}C\ \texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} \mid \texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} \mid e\texttt{.<}\mathit{proj}\texttt{>} \\
 & & \mid & \texttt{[]} \mid e_1 \texttt{ :: } e_2 \\
 & & \mid & \texttt{Alloc}\ e_1\ e_2 \mid \texttt{ref}\ e \mid \texttt{!}e \mid e_1 \texttt{ <- } e_2 \\
 & & \mid & \texttt{`}C\ \texttt{\{}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{\}} \mid \texttt{\{}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{\}} \mid e\texttt{.\{}\mathit{fld}\texttt{\}} \mid e_1 \texttt{ <-\{}\mathit{fld}\texttt{\} } e_2 \\
 & & \mid & \texttt{Reveal}\ e \mid \texttt{GetTag}\ e \mid \texttt{GetSize}\ e \\
 & & \mid & \texttt{match:}\ e_0 \texttt{ with } br_1 \texttt{|}\ldots\texttt{|}\ br_n\ (\texttt{|\_}\ (\texttt{as } s)^? \texttt{ => } e)^? \texttt{ end} \\
 & & \mid & \texttt{Fork}\ e \mid \texttt{Yield} \\
 & & \mid & e\texttt{.[}\mathit{fld}\texttt{]} \mid \texttt{Xchg}\ e_1\ e_2 \mid \texttt{CAS}\ e_1\ e_2\ e_3 \mid \texttt{FAA}\ e_1\ e_2 \\
 & & \mid & \texttt{Proph} \mid \texttt{Resolve}\ e_0\ e_1\ e_2 \\
\textrm{branch} & br & ::= & C\ (x_1 \ldots x_n)^?\ (\texttt{as } s)^? \texttt{ => } e \\
 & & \mid & \texttt{[]}\ (\texttt{as } s)^? \texttt{ => } e \mid x_1 \texttt{ :: } x_2\ (\texttt{as } s)^? \texttt{ => } e \\
\textrm{toplevel value} & v & ::= & t \mid \texttt{\#}n \mid \texttt{\#}b \\
 & & \mid & \texttt{fun:}\ x_1 \ldots x_n \texttt{ => } e \mid \texttt{rec:}\ f\ x_1 \ldots x_n \texttt{ => } e \\
 & & \mid & \S C \mid \texttt{`}C\ \texttt{(}v_1\texttt{,}\ldots\texttt{,}v_n\texttt{)} \mid \texttt{(}v_1\texttt{,}\ldots\texttt{,}v_n\texttt{)} \\
 & & \mid & \texttt{[]} \mid v_1 \texttt{ :: } v_2
\end{array}
$$

Figure 3. Zoo syntax (omitting mutually recursive toplevel functions)

```
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* [6], which has been proven [15] to be equivalent to *linearizability* [2] in sequentially consistent memory models.

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).

Definition map : val :=
  rec: "map" "fn" "t" =>
    match: "t" with
    | Nil =>
        §Nil
    | Cons "x" "t" =>
        let: "y" := "fn" "x" in
        ‘Cons( "y", "map" "fn" "t" )
    end.
```

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

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

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

4.2 Mutually recursive functions

ZOO supports non-recursive (`fun:` $x_1 \ldots x_n$ `=>` e) and recursive (`rec:` f $x_1 \ldots x_n$ `=>` e) functions but only *toplevel* mutually recursive functions. Indeed, it is non-trivial to properly handle mutual recursion: when applying a mutually recursive function, a naive approach would replace the recursive 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 so as to replace by the names instead of the bodies. 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.
Definition f := ValRecs 0 f_g.
Definition g := ValRecs 1 f_g.
Instance : AsValRecs' f 0 f_g [f;g]. Proof. done. Qed.
Instance : AsValRecs' g 1 f_g [f;g]. Proof. done. Qed.
```

4.3 Standard library

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

4.4 Physical equality

In ZOO, a value is either a bool, an integer, a memory location, a function or an immutable block. To deal with physical equality in the semantics, we have to specify what guarantees we get when 1) physical comparison returns `true` and 2) when it returns `false`. We assume that the program is semantically well typed, if not syntactically well typed, in the sense that compared values are loosely compatible: a boolean may be compared with another boolean or a location, an integer may be compared with another integer or a location, an immutable block may be compared with another immutable block or a location. This means we never physically compare, *e.g.*, a boolean and an integer, an integer and an immutable block. If we wanted to allow it, we would have to extend the semantics of physical comparison to account for conflicts in the memory representation of values.

For booleans, integers and memory locations, the semantics of physical equality is plain equality. For abstract values (functions and immutable blocks), the semantics is relaxed: `true` means the values are structurally equal, hence they are equal in COQ; `false` means basically nothing, we do not know because, *e.g.*, two immutable blocks may have distinct identities but same content.

To address the second example of Section 1, we add a twist. By using the `Reveal` primitive on an immutable block, we get the same block annotated with an abstract identifier. The meaning is this identifier is: if physical comparison on two identified blocks returns `false`, the two identifiers are necessarily distinct. The underling assumption that we make here, which is hopefully correct in OCAML, is that the compiler may only introduce sharing. Thanks to this trick, the example can be verified.

4.5 Structural equality

Structural equality is also supported. More precisely, it is not part of the semantics of the language but axiomatized on top of it[5]. The reason is that it is in fact difficult to specify for arbitrary values. Indeed, we have to handle not only abstract tree-like values (booleans, integers, immutable blocks) but also pointers to memory blocks for records. In general, we basically have to compare graphs—which implies structural comparison may diverge.

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

```
Axiom structeq_spec : ∀ `{zoo_G : !ZooG Σ} {v1 v2} footprint,
  val_traversable footprint v1 →
  val_traversable footprint v2 →
  {{{ structeq_footprint footprint }}}
    v1 = v2
  {{{ b, RET #b;
    structeq_footprint footprint ∗
    ⌜ if b then val_structeq footprint v1 v2
      else val_structne footprint v1 v2 ⌝
  }}}.
```

Obviously, this general specification is not very convenient to work with. Fortunately, for abstract tree-like values, we get a much simpler variant:

```
Lemma structeq_spec_abstract `{zoo_G : !ZooG Σ} v1 v2 :
  val_is_abstract v1 →
  val_is_abstract v2 →
  {{{ True }}}
    v1 = v2
  {{{ RET #(bool_decide (v1 = v2)); True }}}
Proof.
  ...
Qed.
```

4.6 Concurrent primitives

Zoo supports concurrent primitives both on atomic references (from `Atomic`) and atomic record fields (from `Atomic.Loc`[6]) 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

[5] We could also have implemented it in Zoo, but that would require more low-level primitives.
[6] The `Atomic.Loc` module is part of the PR that implements atomic record fields.

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

4.7 Prophecy variables

Lockfree algorithms exhibit complex behaviors. To tackle them, IRIS provides powerful mechanisms such as *prophecy variables* [13]. Essentially, prophecy variables can be used to predict the future of the program execution and reason about it. They are key to handle *future-dependent linearization points*: linearization points that may or may not occur at a given location in the code depending on a future observation.

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

5 Conclusion and future work

The development of ZOO is still ongoing. It 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.

ZOO is not yet available on `opam` but can be installed and used in other COQ projects. We provide a minimal example demonstrating its use. We are also working on integrating `ocaml2zoo` with `dune`.

Acknowledgments

We would like to thank Gabriel Scherer for his support and valuable feedback, Vesa Karvonen and Carine Morel for their work on the SATURN [28] library and the discussions it enabled, Thomas Leonard for suggesting verifying the `Rcfd` module from the EIO [29] library, Vincent Laviron and Oliver Nicole for their review of the "Atomic record fields" pull request to the OCAML compiler.

References

[1] 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.

[2] Maurice Herlihy and Jeannette M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (1990), pp. 463–492. URL: https://doi.org/10.1145/78969.78972.

[3] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java". In: *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*. Ed. by Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Vol. 6617. Lecture Notes in Computer Science. Springer, 2011, pp. 41–55. URL: https://doi.org/10.1007/978-3-642-20398-5%5C_4.

[4] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3 - Where Programs Meet Provers". In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Ed. by Matthias Felleisen and Philippa Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 125–128. URL: https://doi.org/10.1007/978-3-642-37036-6%5C_8.

[5] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. "Secure distributed programming with value-dependent types". In: *J. Funct. Program.* 23.4 (2013), pp. 402–451. URL: https://doi.org/10.1017/S0956796813000142.

[6] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. "TaDA: A Logic for Time and Data Abstraction". In: *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*. Ed. by Richard E. Jones. Vol. 8586. Lecture Notes in Computer Science. Springer, 2014, pp. 207–231. URL: https://doi.org/10.1007/978-3-662-44202-9%5C_9.

[7] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Dependable Software Systems Engineering*. Ed. by Alexander Pretschner, Doron Peled, and Thomas Hutzelmann. Vol. 50. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2017, pp. 104–125. URL: https://doi.org/10.3233/978-1-61499-810-5-104.

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

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

[10] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. "Total Haskell is reasonable Coq". In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*. Ed. by June Andronick and Amy P. Felty. ACM, 2018, pp. 14–27. URL: https://doi.org/10.1145/3167092.

[11] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. "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*. Ed. by Tim Brecht and Carey Williamson. ACM, 2019, pp. 243–258. URL: https://doi.org/10.1145/3341301.3359632.

[12] Arthur Charguéraud, Jean-Christophe Filliâtre, Cláudio Lourenço, and Mário Pereira. "GOSPEL - Providing OCaml with a Formal Specification Language". In: *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*. Ed. by Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira. Vol. 11800. Lecture Notes in Computer Science. Springer, 2019, pp. 484–501. URL: https://doi.org/10.1007/978-3-030-30942-8%5C_29.

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

[14] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. "Interaction trees: representing recursive and impure programs in Coq". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 51:1–51:32. URL: https://doi.org/10.1145/3371119.

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

[16] Mário Pereira and António Ravara. "Cameleer: A Deductive Verification Tool for OCaml". In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12760. Lecture Notes in Computer Science. Springer, 2021, pp. 677–689. URL: https://doi.org/10.1007/978-3-030-81688-9%5C_31.

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

[18] 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: *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*. Ed. by Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez. Vol. 13260. Lecture Notes in Computer Science. Springer, 2022, pp. 88–108. URL: https://doi.org/10.1007/978-3-031-06773-0%5C_5.

[19] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. "Creusot: A Foundry for the Deductive Verification of Rust Programs". In: *Formal Methods and Software Engineering - 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24-27, 2022, Proceedings*. Ed. by Adrián Riesco and Min Zhang. Vol. 13478. Lecture Notes in Computer Science. Springer, 2022, pp. 90–105. URL: https://doi.org/10.1007/978-3-031-17244-1%5C_6.

[20] Arthur Charguéraud. *A Modern Eye on Separation Logic for Sequential Programs. (Un nouveau regard sur la Logique de Séparation pour les programmes séquentiels)*. 2023. URL: https://tel.archives-ouvertes.fr/tel-04076725.

[21] Léon Gondelman, Jonas Kastberg Hinrichsen, Mário Pereira, Amin Timany, and Lars Birkedal. "Verifying Reliable Network Components in a Distributed Separation Logic with Dependent Separation Protocols". In: *Proc. ACM Program. Lang.* 7.ICFP (2023), pp. 847–877. URL: https://doi.org/10.1145/3607859.

[22] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. "Verus: Verifying Rust Programs using Linear Ghost Types". In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (2023), pp. 286–315. URL: https://doi.org/10.1145/3586037.

[23] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. "CN: Verifying Systems C Code with Separation-Logic Refinement Types". In: *Proc. ACM Program. Lang.* 7.POPL (2023), pp. 1–32. URL: https://doi.org/10.1145/3571194.

[24] Guillaume Claret. *coq-of-ocaml*. 2024. URL: https://github.com/formal-land/coq-of-ocaml.

[25] Arnaud Daby-Seesaram, Jean-Marie Madiot, François Pottier, Remy Seassau, and Irene Yoon. *Osiris*. 2024. URL: https://gitlab.inria.fr/fpottier/osiris.

[26] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. "RefinedRust: A Type System for High-Assurance Verification of Rust Programs". In: *Proc. ACM Program. Lang.* 8.PLDI (2024), pp. 1115–1139. URL: https://doi.org/10.1145/3656422.

[27] Vesa Karvonen. *Kcas*. 2024. URL: https://github.com/ocaml-multicore/kcas.

[28] Vesa Karvonen and Carine Morel. *Saturn*. 2024. URL: https://github.com/ocaml-multicore/saturn.

[29] Anil Madhavapeddy and Thomas Leonard. *Eio*. 2024. URL: https://github.com/ocaml-multicore/eio.