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 et al. 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.

1.1 OCaml and ZooLang

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 et al. 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 et al. 2022] and future-dependent [Chang et al. 2023; Jung et al. 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 et al. 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 et al. 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 et al. 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.

1.2 Verification results

1.3 Contributions

2 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[1] are completely reimplemented in Zoo and axiom-free, including the `Array`[2] 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 3 and Section 5. 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 5.4.

Concurrent data structures. We provide verified implementations of various concurrent data structures: domain[3] (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.

3 Persistent data structures

4 `Rcfd`: parallelism-safe file descriptor

5 `Saturn`: A library of standard lock-free data structures

5.1 Stacks

5.2 List-based queues

5.3 Stack-based queues

5.4 Work-stealing queues

References

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

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

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

[1]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.

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

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

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

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