# Developing an Iris-Based Program Verification Framework for OCaml
Research internship proposal, M2

Armaël Guéneau, Inria Saclay, armael.gueneau@inria.fr
François Pottier, Inria Paris, francois.pottier@inria.fr

2023

## 1 Overview

There exist a large number of approaches for program verification, which vary in the targeted language, the expressiveness of the logic, the amount of automation provided to the user, the integration between the verification tool and the targeted language, and in the trusted computing base of the approach.

In terms of expressiveness, the Iris Separation Logic framework lies at one end of this spectrum: it provides a powerful, expressive program logic that can be used to reason about a wide variety of subtle properties, for languages with higher-order stateful programs, concurrency, weak memory, distributed nodes communicating over the network, etc. To reason about concrete programs, one needs to *instantiate* the Iris framework with a given programming language, whose syntax and operational semantics must be defined in Coq. One then typically proves language-specific reasoning rules in Separation Logic, and defines custom tactics that help automate certain language-specific reasoning steps.

This instantiation work has been done for a number of minimalistic research languages, such as HeapLang, lambda-rust, Aneris, and more. These "toy" languages are useful for fundational work and experimentation within Iris, but are ill-suited for verification of programs that can be released "in the real world", to be compiled and run as libraries of a mainstream programming language. For instance, idiomatic OCaml programs make extensive use of algebraic datatypes and pattern matching, but HeapLang (which is intended to instantiate Iris with an "ML-like" language) only provides binary pairs and variants. This makes it cumbersome to translate a program from OCaml to HeapLang or from HeapLang to OCaml, and thus difficult to verify realistic OCaml programs using Iris.

We propose to address this issue by instantiating Iris with a large enough, realistic subset of a mainstream programming language, namely OCaml. For a large enough subset of OCaml, we wish to define its syntax so that it matches the syntax of real-world OCaml programs, and to define its operational semantics so that it matches what is implemented by the OCaml compiler. We also wish to develop tooling to easily translate between OCaml source files and their representation in Coq (as a deep embedding). Making it easy to translate between these two representations would be useful for actual program verification and would also serve as a testimony that our language definition inside Coq is close enough to real-world OCaml.

In this internship, we propose to work towards the definition of such a language, which we dub "OCamlLang", and towards an instantiation of Iris with this language. OCamlLang should, ultimately, support features including: algebraic datatypes and pattern matching, records, arrays, mutually recursive functions, exceptions. The task then is to define the syntax and operational semantics of OCamlLang, as well as reasoning rules in Separation Logic. Depending on time, one could also look at writing tools to translate between OCaml source files and OCamlLang programs, or automation in Coq to support verification of OCamlLang programs.

## 2 Roadmap

Broadly speaking, defining a program verification environment based on Iris involves the following steps:

- (in Coq) define the **syntax** and **operational semantics** of OCamlLang;
- (in Coq) define a **weakest-precondition** modality and prove a set of reasoning rules;
- (in Coq) set up **tactics** (or other infrastructure) that helps apply these rules in practice;
- (in OCaml) implement a **translation** of OCaml to OCamlLang;
- (in Coq; optional) set up infrastructure that reflects OCaml's **type discipline** in specifications;
- (in Coq) **validate** the previous steps by verifying a number of small OCaml modules.

Regarding OCamlLang and its operational semantics, a good starting point is the calculus defined by Arthur Charguéraud as part of CFML2 (Charguéraud, 2022). CakeML Kumar et al. (2014); Tan et al. (2019), its operational semantics, and its characteristic formula generator (Guéneau et al., 2017) can also serve as a reference. The tool developed by Léon Gondelman for Aneris (Krogh-Jespersen et al., 2020) may also be of interest. In addition, the MetaCoq project is developed a verified extraction procedure whose target language is Malfunction; they have a formal semantics of Malfunction in Coq. Some questions that we must address include:

- Should we use a (small-step) **substitution-based semantics** or a (small-step) **environment-based semantics**? The former is more common in the Iris literature. The latter may in principle reduce the cost of propagating substitutions during program verification. Perhaps a more important question is, which of the two offers the most readable goals?
- How should we deal with OCaml's **unspecified evaluation order**? The let/and construct can be given a non-deterministic semantics and a "parallel" reasoning rule that is sound with respect to this semantics. The other constructs whose evaluation order is unspecified (applications, tuples, and so on) could be either dealt with in the same way as let/and or reduced to let/and constructs, either via a preliminary translation step, or at runtime (during reduction).
- Should we have just one calculus and one semantics, or possibly one calculus and multiple semantics, or possibly multiple calculi? The desire to **desugar** certain constructs may be a motivation for distinguishing several calculi. As an example, the constructs whose evaluation order is unspecified could be desugared in terms of let/and. As another motivation, the question whether immutable objects (tuples, records, sums) should be regarded as values or as heap blocks might be answered both ways by offering two semantics.
- How should we model **names**, including names of variables, record fields, data constructors?
- What is the best way of modeling OCaml's **tuples**, **records** (with **mutable** fields), and **sums**?
- What is needed to support deep **pattern matching** in an elegant and practical way?
- What is the best way of modeling OCaml's **arrays**, whose length field is immutable?
- What is the best way of modeling **mutually recursive first-class functions**? What is the best way of dealing with (curried) functions of several arguments and with partial applications?

- How should we model OCaml's **modules and functors**? How do we model toplevel definitions that have a side effect? Perhaps the most natural approach is to view modules and functors simply as records and functions. In that case, how can we set up our specifications and proofs so as to be able to reason about one definition at a time?
- Can we model OCaml's polymorphic comparison operators?
- Can we deal with **exceptions** and **delimited control effects**? de Vilhena's work (de Vilhena and Pottier, 2021, 2022) can serve as a source of inspiration.

The features that we do **not** intend to support (at first) include concurrency (especially weak memory concurrency) and GADTs.

Regarding our instantiation of Iris for OCamlLang, the following questions may arise:

- What form should our points-to assertions take? Should we allow just **per-block** or also **per-field** ownership? Regarding arrays, should we allow just **whole-array** ownership or also ownership of **array slices**?
- Must every assertion be affine, or can we distinguish between **affine** and **linear** assertions? Examples of linear objects in OCaml include continuations and system resources (such as network sockets and file descriptors).
- Can we instantiate `diaframe` (Mulder et al., 2022) for OCamlLang?

Regarding the tool that translates OCaml to OCamlLang, the following questions may arise:

- Should the tool duplicate part of OCaml's front-end? Should it share code with the CFML tool? Can it rely on Frédéric Bour's library (ongoing work) which offers access to OCaml's abstract syntax trees?
- How might the tool be thoroughly **tested** so as to ensure semantic preservation?

## 3 Prerequisites

A solid programming background, including fluency in OCaml, is highly desirable. Familiarity with the operational semantics of programming languages (MPRI 2.4) and with Hoare logic and Separation Logic (MPRI 2.36.1) is essential. Familiarity with Coq or with another proof assistant (MPRI 2.7.1 and 2.7.2) is essential.

## 4 Practical details

This internship will be jointly supervised by Armaël Guéneau (Inria Saclay) and François Pottier (Inria Paris). It will take place at Inria Paris from March 2023 to August 2023 approximately. Our colleague Arthur Charguéraud (Inria Strasbourg) will provide expert advice.

## 5 Reading list

To learn about Separation Logic for sequential programs, we suggest reading the educational pearl by Charguéraud (2020), as well as the associated course material (Charguéraud, 2021). Learning about the logical foundations of Iris (Jung et al., 2018) is necessary. For broader information on Separation Logic and its many variants, the surveys by O'Hearn (2019) and by Brookes and O'Hearn (2016) are recommended.

## References

Stephen Brookes and Peter W. O'Hearn. Concurrent separation logic. *SIGLOG News*, 3(3):47–65, 2016. URL http://siglog.hosting.acm.org/wp-content/uploads/2016/07/siglog_news_9.pdf.

Arthur Charguéraud. Separation logic for sequential programs (functional pearl). *Proceedings of the ACM on Programming Languages*, 4(ICFP):116:1–116:34, 2020. URL https://doi.org/10.1145/3408998.

Arthur Charguéraud. *Separation Logic Foundations*, volume 6 of *Software Foundations*. 2021. http://softwarefoundations.cis.upenn.edu.

Arthur Charguéraud. CFML2. https://gitlab.inria.fr/charguer/cfml2, 2022.

Paulo Emílio de Vilhena and François Pottier. A separation logic for effect handlers. *Proceedings of the ACM on Programming Languages*, 5(POPL), January 2021. URL http://cambium.inria.fr/~fpottier/publis/de-vilhena-pottier-sleh.pdf.

Paulo Emílio de Vilhena and François Pottier. A type system for effect handlers and dynamic labels. Submitted, October 2022. URL http://cambium.inria.fr/~fpottier/publis/de-vilhena-pottier-tes.pdf.

Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified characteristic formulae for CakeML. In *European Symposium on Programming (ESOP)*, volume 10201 of *Lecture Notes in Computer Science*, pages 584–610. Springer, April 2017. URL http://cambium.inria.fr/~agueneau/publis/gueneau-myreen-kumar-norrish-cf-cakeml.pdf.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018. URL https://people.mpi-sws.org/~dreyer/papers/iris-ground-up/paper.pdf.

Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. Aneris: A mechanised logic for modular reasoning about distributed systems. In *European Symposium on Programming (ESOP)*, volume 12075 of *Lecture Notes in Computer Science*, pages 336–365. Springer, April 2020. URL https://iris-project.org/pdfs/2020-esop-aneris-final.pdf.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In *Principles of Programming Languages (POPL)*, pages 179–192, January 2014. URL https://cakeml.org/popl14.pdf.

Ike Mulder, Robbert Krebbers, and Herman Geuvers. Diaframe: automated verification of fine-grained concurrent programs in Iris. In *Programming Language Design and Implementation (PLDI)*, pages 809–824, June 2022. URL https://doi.org/10.1145/3519939.3523432.

Peter W. O'Hearn. Separation logic. *Communications of the ACM*, 62(2):86–95, 2019. URL https://doi.org/10.1145/3211968.

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29:e2, 2019. URL https://cakeml.org/jfp19.pdf.