# Mechanized Reasoning about a Capability Machine

Aïna Linn Georges
Aarhus University
ageorges@cs.au.dk

Alix Trieu
Aarhus University
alix.trieu@cs.au.dk

Lars Birkedal
Aarhus University
birkedal@cs.au.dk

## Abstract

Capability machines are promising targets for secure compilers since capabilities can be used to enforce abstractions that are usually expected for high-level languages, such as well-bracketed control-flow (WBCF) and local state encapsulation (LSE). We present the first formalization of a capability machine that supports mechanized reasoning about deep semantic properties, including WBCF and LSE. Our formalization is done in the Coq implementation of Iris, a state-of-the-art concurrent higher-order separation logic, and includes a formalization of the logical relation defined by Skorstensgaard et al. [15], which can used to prove WBCF and LSE.

***Keywords*** keyword1, keyword2, keyword3

## 1 Introduction

Capability machines allow for fine grained control over the authority of memory [14, 18]. At the machine level, pointers are replaced by capabilities, to which is attached a range of authority and a permission. When the machine executes an instruction it dynamically checks that the instruction uses a capability within its range of authority. Capability machines are promising targets for secure compilers because these dynamic checks can be used to enforce abstractions that are usually expected for high-level languages, such as well-bracketed control-flow (WBCF) and local state encapsulation (LSE).

We present the first formalization of a capability machine that supports mechanized reasoning about deep semantic properties, including WBCF and LSE.

Our formalization builds upon earlier work of Skorstengaard et al. [15] and [16], who present two different capability machines and calling conventions that enforce well-bracketed control flow, and methods for defining and reasoning about capability machines. In each case, they define a logical relation to capture a semantic notion of capability safety and use it to prove WBCF and LSE.

The logical relations in [15] and [16] are so-called step-indexed Kripke logical relations, which means that they are indexed over recursively-defined worlds, which contain descriptions of invariants of the memory of the machine. It is well-known that it is non-trivial to define and work with such step-indexed Kripke logical relations [1, 2]. Therefore we do not formalize the logical relations of Skorstengaard et al. directly, but rather give a more abstract *logical* definition of the logical relations in the Iris program logic framework [7–10], which comes with built-in support for abstract reasoning about recursion (qua the later modality and Löb induction) and invariants. Such a logical approach to defining logical relations has been used successfully before for logical relations for typed high-level languages (e.g., [3, 6, 13, 17]); here we use it for the first time for a low-level untyped machine language. Another reason for using Iris is that we can use the Coq implementation of Iris and the Iris proof mode [11] to mechanize our development.

In summary, we present Iris formalizations of

- a program logic for reasoning about capability machine programs.
- the logical relation from [15], which captures capability safety and which can be used to reason about examples that rely on WBCF and LSE.

Currently, almost all of the technical development is mechanized in Coq using the Coq implementation of Iris. The mechanization presently consists of 5.8K lines of spec and 15K lines of proof as reported by coqwc. Its substantial size can probably be reduced by better use of tactics. However, such a mechanization will always be non-trivial due to the nature of the capability machine with dynamic checks and multiple ways in which instructions can fail.

## 2 A Program Logic for a Capability Machine

In this section we give a brief overview of how we define a program logic for reasoning about the capability machine from [15] in the Iris framework.

While Iris is a framework and supports many languages, it is geared towards models of higher-level languages, which abstract from the fact that programs are stored in memory, and hence come equipped with notions of expressions and values, in addition to the program memory. In contrast, our low-level capability machine model has no notion of expression and values, it just consists of a memory and a register file. The program counter register contains a pointer to an address in memory. That address in turn will contain an integer, which can then be decoded to a machine instruction, such as Load, Store, Jump, etc. Once an instruction has been executed, the program counter is updated and will then point to the next instruction in memory.

*PL'18, January 01–03, 2018, New York, NY, USA*
2018.

To capture the semantics of the capability machine in the Iris framework, we introduce an abstract notion of an instruction, whose operational meaning is to execute the instruction the program counter points to, and abstract notions of values, for halted and failed configurations.

Next we use the Iris framework to *prove* Hoare triples for each instruction. As in [5], we define a points-to predicate for registers, denoted $r \mapsto_r w$. Since a capability machine replaces pointers with capabilities, we replace the conventional points-to predicate of separation logic for pointers with a points-to predicate with a permission attached to it, denoted $a \mapsto_a [p]\ w$. This predicate states that address $a$ points to word $w$ with permission $p$. The permission restricts how the memory may be updated at address $a$. For instance, if $a \mapsto_a [\textsc{rx}]\ w$, then the RX (ReadExecute) permission gives us permission to read from address $a$, to execute w, but it does not permit us to write to address $a$.

The Hoare triples for basic instructions take the following form

$$\textit{run time conditions} \wedge \textit{decode}(w) = \textit{instr} \Rightarrow$$
$$\{\{\{\mathsf{PC} \mapsto_r ((p,g),b,e,a) * a \mapsto_a [p] w * \ldots\}\}\}$$
$$\mathsf{Instr\ Executable}$$
$$\{\{\{\mathsf{PC} \mapsto_r ((p,g),b,e,a+1) * a \mapsto_a [p] w * \ldots\}\}\}$$

Here, the runtime conditions correspond to the dynamic checks done by the capability machine, and Instr Executable is the abstract expression for executing the next instruction in memory. This form of Hoare triple is similar to the one used in [5] but unlike [5], the decoding function is in our case assumed.

Here we have only described the format for Hoare triples for individual instructions; we use a trick involving the standard bind rule of Iris to reason about programs consisting of many instructions, but we omit the description of that from this extended abstract.

## 3 Logical Relation

We now outline how we define a unary logical relation in Iris that captures capability safety. We define a value relation $\mathcal{V}$ as an Iris relation of type World $\rightarrow$ Word $\rightarrow$ *iProp*, where World is a collection of state transition systems used to reason about local state and Word is the type of capability machine words. It is well-known how state transistion systems can be defined in Iris via Iris' notion of monoids and ghost state [9].

Our notion of World is simpler and more abstract than the one used in the concrete logical relation given by [15], where the World is a collection of invariants describing the behaviour of all memory, not just local state. We can use a simpler more abstract notion of world because the Iris model takes care of the world circularity problem. In particular, we make use of Iris' higher-order ghost state — the ability to store arbitrary higher-order separation-logic predicates in ghost variables — to define the validity of the regions a capability has authority over. The semantics of Iris' higher-order ghost state involves solving a recursive equation [7].

Logical accounts of logical relations for high level languages with references have used Iris invariants to define semantic validity of reference locations, see, e.g., [12]. This approach suffices for reasoning about examples involving local state encapsulation in languages where all calls are well-bracketed.

In our case, however, calls are not always well-bracketed (since the capability machine includes general jump instructions). Skorstengaard et al. [15] distinguishes between well-bracketed and non well-bracketed calls by using notions of public- and private future worlds, following [4]. Iris invariants do not make such a distinction and thus invariants alone are not sufficient for our purposes. Instead we *explicitly* define notions of public and private future world relations, rather than relying on Iris' *implicit* future world, and use these notions to distinguish between well-bracketed and non well-bracketed calls. We use these relations in combination with Iris' higher-order ghost state, which allow us to save a predicate by associating it to a unique ghost name. Using that unique name, we are then able to refer to the saved predicate somewhere else, and apply it to an appropriate future world argument.

We prove the Fundamental Theorem of Logical Relations, which roughly says that if we have a read-execute capability, and it is capability safe to read it, then it is capability safe to execute it (use it as a program counter). We then use the fundamental theorem to prove functional correctness of examples with calls to unknown adversary code and whose correctness relies on local state encapsulation and well-bracketed control flow.

In the future we plan to finish the remaining details of the implementation, then use it as a starting point to prove full abstraction of a compiler from a high level language to our capability machine. Furthermore, it can also be used as a starting point for exploring different kinds of capabilities.

## Acknowledgments

This research was supported in part by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU) and by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation.

## References

[1] A. Ahmed, A. Appel, and R. Virga. 2002. A Stratified Semantics of General References. In *LICS*.

[2] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. 2011. Step-Indexed Kripke models over recursive worlds. In *POPL*.

[3] D. Dreyer, A. Ahmed, and L. Birkedal. 2011. Logical Step-Indexed Logical Relations. *LMCS* 7, 2:16 (2011).

[4] Derek Dreyer, Georg Neis, and Lars Birkedal. 2012. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming* 22, 4-5 (2012), 477–528. https://doi.org/10.1017/S095679681200024X

[5] Jonas B. Jensen, Nick Benton, and Andrew Kennedy. 2013. High-level Separation Logic for Low-level Code. *SIGPLAN Not.* 48, 1 (Jan. 2013), 301–314. https://doi.org/10.1145/2480359.2429105

[6] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the Foundations of the Rust Programming Language. In *POPL*.

[7] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. 256–269.

[8] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[9] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. 637–650.

[10] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The essence of higher-order concurrent separation logic. In *European Symposium on Programming (ESOP)*.

[11] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-order Concurrent Separation Logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 205–217. https://doi.org/10.1145/3009837.3009855

[12] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *POPL*.

[13] Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. 2017. A Logical Account of a Type-and-Effect System. In *POPL*.

[14] Henry M. Levy. 1984. *Capability-Based Computer Systems*. Digital Press. https://homes.cs.washington.edu/~levy/capabook/

[15] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2018. Reasoning About a Machine with Local Capabilities. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 475–501.

[16] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019. StkTokens: Enforcing Well-bracketed Control Flow and Stack Encapsulation Using Linear Capabilities. *Proc. ACM Program. Lang.* 3, POPL, Article 19 (Jan. 2019), 28 pages. https://doi.org/10.1145/3290332

[17] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2018. A Logical Relation for Monadic Encapsulation of State: Proving contextual equivalences in the presence of runST. *Proc. ACM Program. Lang.* 2, POPL (Jan. 2018), to appear.

[18] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *IEEE Symposium on Security and Privacy*. IEEE, 20–37. https://doi.org/10.1109/SP.2015.9