Full Title*

Subtitle†

Anonymous Author(s)

Abstract

Text of abstract ....

Keywords keyword1, keyword2, keyword3

1 Introduction

Part of showing full abstraction consists of showing that the expressive power of a compiled program, is exactly the expressive power as the source program. This becomes a challenge when the expressive power of the target language is bigger than the expressive power of the source language.

Take for instance the compilation from a high level functional language, to a low level machine language. Certain high level abstractions are taken for granted by the high level language, such as local state encapsulation and well bracketed control flow. At the lowest level, a machine may allow arbitrary jumps to instructions in memory, and arbitrarily manipulate pointers to access any local state otherwise hidden by the top level abstraction. It must therefore be up to the compiler to enforce such abstractions, if we want to show secure compilation.

Capability Machines allow for fine grained control over the *authority* of memory. At the machine level, pointers are replaced by capabilities, to which is attached a range of authority and a permission. The language then executes instructions by dynamically checking that the instruction uses a capability within its range of authority. Using a capability machine language as the target of a secure compiler allows us to take advantage these dynamic checks to enforce the high level abstractions mentioned above.

In [3] and [4], Skorstensgaard et. al. present two different calling conventions that enforce well bracketed control flow, and methods for defining and reasoning about capability machines. In each case, they define a step indexed Kripke Logical Relation to define a notion of capability safety. However, such logical relations are notoriously difficult to reason about [1]. A formalization of these definitions into a proof assistant could facilitate such reasoning, and enable easier exploration of extensions and variations to the language.

Iris is a higher-order concurrent separation logic framework. Its support of higher-order ghost state makes Iris a great candidate for complex reasoning about *state*. This is exactly what we need for reasoning about a machine, where computation consist of state manipulation. Furthermore, the model of Iris is itself step-indexed, and comes equipped with a later modality: ▷. The later modality enables elegant Iris definitions of step-indexed logical relations [2] [5].

The work in process formalization will make the following contributions:

- A formalization of a capability machine program logic in Iris, in which programs lie in memory, and execution is completely statefull.
- An Iris formalization of a Logical Relation to reason about programs in a capability machine language, which can be used to reason about local state while distinguishing between well-bracketed, and non well-bracketed calls.

2 Iris Embedding of a Capability Machine

The first step of formalizing the logical relation, is to embed the capability machine language into Iris. We want a formalization which can be used to show full abstraction from a high level language, to a low level capability machine. We are interested in the case where the target language does not natively enforce certain basic abstractions of a high level language: local state encapsulation of well bracketed control flow. With this goal in mind, it is important not to *accidentally* formalize a language where certain control flow abstractions are still enforced, such as implementing the language using frames and labels. Rather, we are interested in reasoning about a language with arbitrary jumps, and (almost) arbitrary capability manipulations (within its given range of authority).

Machine instructions operate through memory. Typically a machine will have a special register; the program counter, which contains a pointer to an address in memory. That address in turn will contain an integer, which can then be decoded to various machine dependent instructions, such as Load, Store, Jump, etc. Once an instruction has been executed, the program counter is incremented and will now point to the next instruction in memory. A jump now simply consist of a store into the program counter itself. Strictly speaking, such a language does have expressions, rather it executes given the configurations: the state of registers and the state of memory.

3 Logical Relation

3.1 Definition of the Logical Relation

The logical relation is a unary relation that relates a word to *capability safety*. Figure **??**) describes the Iris implementation

*Title note

†Subtitle note

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

$$\mathcal{E}(W)(pc) \triangleq \forall r, \mathcal{R}(W)(r) \ * \ \text{context}(W)(r[\text{PC} := pc]) \\ \twoheadrightarrow \text{WP Seq (Instr Executable)} \ \{v, v = \mathit{HaltedV} \implies \exists W' r', W' \sqsubseteq_{priv} W * \text{context}(W')(r')\}$$

Figure 1. Logical Relation for Expressions

$$\mathcal{R}(W)(r) \triangleq \forall (reg : \text{RegName} \backslash \text{PC}), \mathcal{V}(W)(r(reg))$$

Figure 2. Logical Relation for Register States

$$\begin{aligned}
\mathcal{V}(W)(z) &\triangleq \exists z' \in \mathbb{Z}. z = z' \\
\mathcal{V}(W)(((\text{O}, g), b, e, a)) &\triangleq \top \\
\left.\begin{array}{l}\mathcal{V}(W)(((\text{RO}, g), b, e, a)) \\ \mathcal{V}(W)(((\text{RW}, g), b, e, a)) \\ \mathcal{V}(W)(((\text{RWL}, g), b, e, a))\end{array}\right\} &\triangleq \exists\, p', p \sqsubseteq p' * \text{read_write_cond(p',b,e)} \qquad \textit{where p is RO, RW, RWL resp.} \\
\left.\begin{array}{l}\mathcal{V}(W)(((\text{RX}, g), b, e, a)) \\ \mathcal{V}(W)(((\text{RWX}, g), b, e, a)) \\ \mathcal{V}(W)(((\text{RWLX}, g), b, e, a))\end{array}\right\} &\triangleq \exists\, p', p \sqsubseteq p' * \text{read_write_cond(p',b,e)} \\ &\qquad * \ \square\ \text{exec_cond(W)(p,g,b,e)} \qquad \textit{where p is RX, RWX, RWLX resp.} \\
\mathcal{V}(W)(((\text{E}, g), b, e, a)) &\triangleq \square\ \text{enter_cond}(W)\text{(g,b,e,a)}
\end{aligned}$$

Figure 3. Logical Relation for Words

$$\begin{aligned}
\text{read_write_cond}(p, b, e) &\triangleq \mathop{\Large *}_{a \in [b,e]} \text{rel}(a, p, \mathcal{V}) \\
\text{exec_cond(W)(p,g,b,e)} &\triangleq \forall a \in [b\ e], W' \sqsubseteq^{g} W. \ \triangleright\ \mathcal{E}(W')(((p, g), b, e, a)) \\
\text{enter_cond(W)(g,b,e,a)} &\triangleq \forall W' \sqsubseteq^{g} W. \ \triangleright\ \mathcal{E}(W)(((\text{RX}, g), b, e, a))
\end{aligned}$$

$$\text{where } W' \sqsubseteq^{Global} W \iff W' \sqsubseteq_{priv} W \\ \text{and } W' \sqsubseteq^{Local} W \iff W' \sqsubseteq_{pub} W$$

Figure 4. Capability Conditions

of the logical relation defined by Skorstensgaard et. al. [3], in which the authors define a capability machine with Local capabilities, which imposes certain restrictions on where to store such capabilities. The value relation $\mathcal{V}$ is an Iris relation of type $World \to Word \to iProp$, where the World is a collection of state transition systems used to reason about local state, and a Word can be an integer of a capability. We then define the register relation $\mathcal{R}$ as the validity of each word that the register state maps to.

The expression relation should capture what it means for programs to be capability safe. However, as previously discussed, programs in a machine lie in memory, pointed to by the program counter. Strictly speaking, there are no "expressions" in such a machine language. Instead, the expression relation relates Words to the notion of program capability safety, where the word in question will be the program counter state.

3.2 The Context

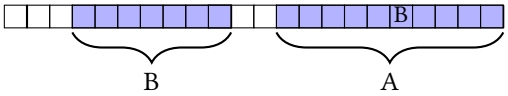
In the expression relation, the context contains the ghost resources necessary to run a program pointed to by the program counter. When the program has finished running, ownership of these resources is given back. To start out, the running program gets full ownership of all machine registers. A ghost register will be represented by the full ownership of

the map from RegName to Word. We write

$$r \mapsto_r w$$

to indicate that registers r currently contains the word w. The context will contain such a statement for each register in the register map.

A program may also need the ghost resources for the pieces of memory it has authority over. However, we want to respect the principle of least privilege. A program should only be able to gain ownership over the ghost resources for memory pointed to by a capability. Say a register contains a capability A. Consider the following memory layout.



In that case, the program may have authority over the region given by A, as well as the region given by B. However if no other capabilities are given, the program should not be given access to the ghost resources denoted by the white regions.

One way to enforce this, is to define the value relation of a capability as an invariant containing the resources for its range of authority, which at the same time guarantees that all the words that these words point to are also valid. However, as the name indicates, an invariant can never change. This is not a problem when the only variant is the current state of a location. In this case however, the value relation is indexed over a particular *World*. This World will capture the behaviour of local state. The validity of an executable capability will consider any possible future world. This becomes crucial when showing safety of programs that depend on well bracketed control flow.

3.3 The World

References

[1] Amal G Ahmed. 2004. Semantics of types for mutable state. Ph.D. DIssertation. Princeton University.

[2] 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

[3] 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.

[4] 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

[5] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2017. A Logical Relation for Monadic Encapsulation of State: Proving Contextual Equivalences in the Presence of runST. *Proc. ACM Program. Lang.* 2, POPL, Article 64 (Dec. 2017), 28 pages. https://doi.org/10.1145/3158152

A Appendix

Text of appendix …