# Verified efficient libraries for CakeML

## HRUTVIK KANABAR\*, University of Kent, UK

The CakeML project has matured since 2012, and is now a practical language with an optimising, end-to-end verified compiler. We turn our attention to performance and usability, aiming to enable useful applications in CakeML which maintain the guarantees of the verified framework.

We are currently investigating a logical relation which will give CakeML a semantic type system, following the approach of projects such as RustBelt. This has several applications, including improved performance in imperative compiler benchmarks, and stronger reasoning about CakeML types.

We first apply the technique to a version of System F with CakeML-like semantics, augmented by existential, general isorecursive, sum, and product types. As in the rest of the CakeML project, our work is formalised in HOL4.

Additional Key Words and Phrases: semantic types, logical relations, formal verification, HOL4

### **1 INTRODUCTION**

CakeML is a formally-specified programming language, with a verified end-to-end correct optimising compiler. Using the interactive theorem-prover HOL4, the language semantics are specified in the functional big-step style: as a *clocked* (or *fuelled*) recursive interpreter [15].

The compiler correctness theorems state that input is compiled to machine code with the same semantics, provided this semantics does not get "stuck" – i.e. the input is *safe*. For user-written code, this condition is enforced by invoking a sound and complete type inferencer, and appealing to our type soundness theorem. However, syntactic type soundness is a whole-program property, and so CakeML libraries must syntactically type-check to be usable in user code. This imposes strong restrictions: for example, unchecked array accesses cannot syntactically be assigned a type in a type-safe language, and so CakeML does not have efficient libraries operating on arrays. This contributes to CakeML's slow performance in imperative benchmarks [18].

We aim to provide a more general method to ensure safety in CakeML. This should be compositional with little added effort from users, and should permit reasoning about almost any CakeML expression (including those that do not syntactically type-check). We are pursuing a unary logical relation to give a semantic type system for CakeML (*§2*), following the approach of projects such as RustBelt [9, 10]. We have first applied the technique to a version of System F with existential, general isorecursive, sum, and product types, as well as a CakeML-like semantics. We are exploring some use cases (*§3*), which we plan to transfer into CakeML in future work.

## 2 A LOGICAL RELATION FOR SEMANTIC TYPING IN CAKEML

We define unary logical relations on System F values and expressions in the usual way [2], using step-indexing to handle general isorecursive types. The relations are parametrised by a type, to which they give a semantics: a value or expression is in the respective relation if it is safe to use at the type for the number of computational steps given by the step-index.

We lift these relations to term variable contexts ( $\Gamma$ ), to give a notion of semantic typing:

 $\Gamma; \ \rho \vDash e: \tau \implies e \text{ is safe to use as if it has type } \tau, \text{ in any environment}$ whose values are safe to use as if they have the types given by  $\Gamma$ . (1)

<sup>\*</sup>Supported by the UK Research Institute in Verified Trustworthy Software Systems (VeTSS).

Author's address: Hrutvik Kanabar, hk324@kent.ac.uk, University of Kent, Canterbury, UK, CT2 7NZ.

Here,  $\rho$  is a semantic type variable context, a collection of relations which give semantics to the free type variables in  $\tau$ . The semantic typing judgement must reason about safety for any number of computational steps, so its definition universally quantifies over the step-index.

This approach is both compositional and general enough to be appropriate for our initial motivation of efficient libraries (§3.1). It further enables applications in reasoning about data abstraction, and verified compilation to CakeML (§3.2, §3.3).

#### **3 APPLICATIONS**

#### 3.1 Verified efficient libraries

Efficient libraries often do not syntactically type-check, so languages provide "escape hatches" to circumvent checks (e.g. OCaml has unsafe\_get for non-bounds-checked array accesses, and Obj.magic for unsafe casts; Rust has unsafe for freely aliasing pointers). However, users of such libraries lose the ability to assure safety of their code by syntactically type-checking it – syntactic type-soundness requires that *every* component of a program must type-check:

$$\Gamma \vdash e : \tau \land (\forall x_i \in \operatorname{dom}(\Gamma) : \emptyset \vdash e_i : \Gamma(x_i)) \implies \neg\operatorname{crash} (e^{[e_i/x_i]})$$

Though type-checking is compositional, and can consider each component  $(e, e_i)$  of the whole program  $(e^{[e_i/x_i]})$  independently, safety  $(\neg \operatorname{crash}(\cdot))$  is only guaranteed if all checks succeed.

We observe that in CakeML all libraries are verified, so we only have to compose verified library code with syntactically typeable (but perhaps unverified) user code safely. We take inspiration from the RustBelt project – for each syntactic typing rule ( $\Gamma$ ;  $\Delta \vdash e : \tau$  for term context  $\Gamma$  and type variable context  $\Delta$ ) of our version of System F, we have proven a semantic equivalent ( $\Gamma$ ;  $\rho \vDash e : \tau$  for semantic type variable context  $\rho$ , as in *Equation 1*):

$$\frac{p_1 \cdots p_n}{\Gamma; \operatorname{dom}(\rho) \vdash e : \tau} \implies \left[ \left[ \frac{p_1 \cdots p_n}{\Gamma; \operatorname{dom}(\rho) \vdash e : \tau} \right] \right]$$
where
$$\left[ \left[ \frac{p_1 \cdots p_n}{\Gamma; \operatorname{dom}(\rho) \vdash e : \tau} \right] \stackrel{\text{def}}{=} \frac{\llbracket p_1 \rrbracket \cdots \llbracket p_n \rrbracket}{\Gamma; \rho \vDash e : \tau} \right]$$

This allows us to compose syntactic and semantic typing derivations, with a verification obligation on the library writer only.

#### 3.2 Reasoning about data abstraction

Like many high-level languages, CakeML uses modules to provide data abstraction: module-writers can restrict implementation exposure to ensure that internal invariants are maintained. Strong type systems implicitly enforce this abstraction; however they do not provide appropriate formal guarantees [7].

Semantic typing allows us to embed the invariants into the type variable environment of the semantic typing judgement (*Equation 1*). To demonstrate this approach, we have encoded simple modules in the existential type constructs of our version of System F. We can then verify that well-typed uses of these modules must preserve their invariants. In future work, we plan to transfer this to the CakeML module system by proving a general Reynolds-style abstraction theorem, taking inspiration from Crary [7].

*Candle: a HOL prover implemented in CakeML.* We plan to apply this abstraction theorem to Candle, a HOL theorem-prover implemented in CakeML [13]. Following the LCF-style of theorem-provers, Candle implements a theorem datatype and primitive inference rules in a trusted module. Exposure of the datatype is restricted so that only the trusted module can construct theorems; thus

Verified efficient libraries for CakeML

all theorems can only be constructed using the inference rules. To show that Candle produces only valid theorems, we must prove that the abstraction is correctly enforced.

## 3.3 Verified compilation to CakeML

We believe that semantic typing will enable the use of code written directly in CakeML alongside Coq-verified code which is then compiled to CakeML.

Verified extraction from Coq to CakeML could compose with the existing CakeML verified compiler, enabling an end-to-end verification toolchain for Coq programs with CakeML at its core. However, the specification language of Coq is based on the Calculus of Inductive Constructions, and has a more expressive type system (allowing for features such as dependent types) than that of CakeML. Therefore, CakeML code compiled from Coq will not always syntactically type-check, even though Coq's type system assures safety of the source code. Semantic typing allows us to reason about safety at the CakeML level, and so enable both composition with user code and invocation of the CakeML compiler correctness theorem.

Currently, Coq provides unsafe extraction to OCaml, with the insertion of arbitrary casts in generated code to pass the OCaml type-checker.

#### 4 RELATED WORK

The proof technique of (step-indexed) logical relations is well-studied [1, 2, 5, 16], especially when applied to the problem of contextual equivalence [8, 17, 19]. It has also successfully been used in various semantic type soundness proofs, particularly recently [9].

The RustBelt project gives a semantics to Rust types to show safety of expressions with respect to the data-sharing disciplines of Rust [10]. The project (like much of the recent literature) uses the general Iris framework for higher-order concurrent separation logics [11]. The Iris logic is derived from a small set of primitives and is shallowly-embedded in Coq. This embedding gives Iris much of its generality and extensibility, but relies on the dependently-typed meta-logic of Coq. Therefore, it seems we cannot express the Iris logic in HOL. As a result, we are restricted to using more primitive mathematical constructs, similar to the literature pre-Iris, and it is well-known that this is intricate work. We could alternatively take inspiration from earlier versions of Iris, which relied on a greater set of primitives and so were less general [12] – these could be expressible in HOL, but further exploration is required.

To our knowledge, our semantic type system is novel in that it will be the first such system to reason about a real-world, formally-specified language: though RustBelt has successfully found bugs in Rust, it relies on a lambda calculus intended to model core Rust features, and no official formal specification exists for the real-world language. Our system is also formalised outside of Iris, and backed by a functional big-step semantics.

CertiCoq [3, 6] provides verified compilation of Coq to Clight, the front-end language of the CompCert verified compiler [14]. The project plans to allow linking of CertiCoq-compiled code to written Clight code, mirroring our goal in CakeML. It also intends to enable linking of the proof theory of Coq and that of Clight; the latter is the Verified Software Toolchain (VST), a separation logic for Clight [4]. The disparities between the program logics for the functional language of Coq and the imperative Clight pose problems here, but this may not be the case with a verified extraction to CakeML. This could allow for code to be partly verified in Coq and partly using CakeML frameworks – further research is required to determine feasibility.

#### REFERENCES

- Amal Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In Proceedings of the 15th European Conference on Programming Languages and Systems (ESOP'06), Peter Sestoft (Ed.). Springer, Berlin, Heidelberg, Germany, 69–83. https://doi.org/10.1007/11693024\_6
- [2] Amal J. Ahmed. 2004. Semantics of Types for Mutable State. Ph.D. Dissertation. Princeton, NJ, USA. Advisor(s) Andrew Appel.
- [3] Abhishek Anand, Andrew W. Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Bélanger, Matthieu Sozeau, and Matthew Weaver. 2016. CertiCoq : A verified compiler for Coq. In Proceedings of the 3rd International Workshop on Coq for Programming Languages (CoqPL'17), Sandrine Blazy and Emilio Jesús Gallego Arias (Eds.). ACM, New York, NY, USA.
- [4] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. Program Logics for Certified Compilers. Cambridge University Press, New York, NY, USA.
- [5] Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-carrying Code. ACM Trans. Program. Lang. Syst. 23, 5 (Sept. 2001), 657–683. https://doi.org/10.1145/504709.504712
- [6] Oliver Savary Belang/er. 2019. Verified Extraction for Coq. Ph.D. Dissertation. Princeton, NJ, USA. Advisor(s) Andrew Appell. ftp://ftp.cs.princeton.edu/techreports/2019/011.pdf
- [7] Karl Crary. 2017. Modules, Abstraction, and Parametric Polymorphism. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17). ACM, New York, NY, USA, 100–113. https://doi.org/ 10.1145/3009837.3009892
- [8] Derek Dreyer, Georg Neis, and Lars Birkedal. 2010. The Impact of Higher-order State and Control Effects on Local Relational Reasoning. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (*ICFP'10*), Paul Hudak and Stephanie Weirich (Eds.). ACM, New York, NY, USA, 143–156. https://doi.org/10.1145/ 1863543.1863566
- [9] Derek Dreyer, Amin Timany, Robbert Krebbers, Lars Birkedal, and Ralf Jung. 2017. What type soundness theorem do you really want to prove? (October 2017). Retrieved November 7, 2019 from https://blog.sigplan.org/2019/10/17/whattype-soundness-theorem-do-you-really-want-to-prove/
- [10] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154
- [11] 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. *Journal of Functional Programming* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151
- [12] 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 Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15). ACM, New York, NY, USA, 637–650. https://doi.org/10.1145/2676726.2676980
- [13] Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. 2014. HOL with Definitions: Semantics, Soundness, and a Verified Implementation. In *Interactive Theorem Proving (ITP) (Lecture Notes in Computer Science)*, Gerwin Klein and Ruben Gamboa (Eds.), Vol. 8558. Springer, 308–324. https://doi.org/10.1007/978-3-319-08970-6\_20
- [14] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert: a formally verified optimizing compiler. In *Proceedings of the 8th European Congress on Embedded Real Time Software and Systems (ERTS'16)*, Kjeld Hjortnaes and Joseph Sifakis (Eds.).
- [15] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-step Semantics. In Proceedings of the 25th European Symposium on Programming, ESOP 2016 (Lecture Notes in Computer Science), Peter Thiemann (Ed.), Vol. 9632. Springer, New York, NY, USA, 589–615. https://doi.org/10.1007/978-3-662-49498-1\_23
- [16] Benjamin C. Pierce. 2004. Advanced Topics in Types and Programming Languages. The MIT Press, Massachusetts, USA.
- [17] Andrew Pitts and Ian Stark. 1993. Observable Properties of Higher Order Functions that Dynamically Create Local Names, or: What's new?. In Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science (MFCS'93), A.M. Borzyszkowski and S. Sokolowski (Eds.). Springer, Berlin, Heidelberg, Germany, 122–141.
- [18] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2019. The verified CakeML compiler backend. *Journal of Functional Programming* 29 (2019), e2. https://doi.org/10.1017/ S0956796818000229
- [19] Amin Timany. 2018. Contributions in Programming Languages Theory: Logical Relations and Type Theory. Ph.D. Dissertation. Leuven, Belgium. Advisor(s) Bart Jacobs and Frank Piessens. https://lirias.kuleuven.be/1838165