Brack: A Verified Compiler for Scheme via CakeML

Pascal Lasnier
University of Cambridge
Cambridge, UK

Jeremy Yallop
University of Cambridge
Cambridge, UK

Magnus Myreen
Chalmers University of Technology
Gothenburg, Sweden

Introduction. This talk introduces Brack[1], a new verified compiler for a subset of Scheme including `call/cc`, developed in the HOL4 interactive theorem prover [13]. Complete verification was achieved by building a compiler from Scheme to ML and using the verified compiler CakeML [10] for the remainder of compilation to machine code. Compiling via CakeML is the approach taken by the recent Haskell-like language compiler PureCake [9], and we see it as a promising approach to verified compilation for other new projects.

Features. Brack features arithmetic, selection, mutable variables, lambda expressions, `letrec`s, lists, and first-class continuations with `call/cc`, making Brack the first verified compiler to support first-class continuations. With these features, it is possible to implement recursive algorithms such as factorial and the Fibonacci sequence, list operations such as map and fold, and backtracking-based non-deterministic algorithms implemented using `call/cc` [8, Chap. 22]. Brack is written in ~4000 non-whitespace lines of HOL4 proof scripts.

CakeML and HOL4. CakeML [10] is a complete verified compiler for ML. Verified compilers are written in proof assistants such as HOL4 [13] and feature semantic preservation proofs to verify that the observable trace of program execution is the same before and after it is compiled. Such proofs are based on (1) a formal semantics of the source language, (2) a formal semantics of the target language, and (3) a definition of the compilation algorithm. HOL4 users tend to write (1)–(3) as functional programs in the logic of HOL4 in order to maximise use of rewriting in proofs.

Even with significant use of rewriting in proofs, semantic preservation proofs are difficult to develop, so verified compilers often feature multiple compiler passes between several intermediate languages (ILs), with separate semantic preservation proofs for each pass which compose for the whole compiler. Brack exploits CakeML's existing verification for full compilation to machine code, and adds a single pass from Scheme to CakeML which performs a CPS transform.

Continuation-passing style. Like many Scheme compilers, from Rabbit [15] onwards, Brack uses a continuation-passing style (CPS) transform in its compilation. CPS is a natural program representation for one of Scheme's most distinctive features: first-class continuations, through the control operator `call/cc` [7]. This natural representation enables the compiler verification proof for `call/cc`, making Brack the first verified compiler to support first-class continuations.

Semantic preservation. The verification of Brack requires a semantic preservation proof from Scheme to CakeML that composes with the CakeML compiler's semantic preservation theorem. To achieve this proof, Brack takes advantage of the fact that operational small-step semantics are a *defunctionalisation* of the continuation-passing style, a notion explored by Reynolds, Danvy, and others [1–4, 12]. Particularly, Ager et al. demonstrate that a CEK-machine may be *refunctionalised* into a CPS interpreter, based on the reduction steps corresponding to particular continuations defined by the abstract machine [1].

The semantics of Brack are based on the formal small-step operational semantics described in the Sixth Revised Report for Scheme (R^6RS) [14]. Implementing the operational semantics as a CESK-machine [5, 6] allows us to apply Ager et al.'s refunctionalisation technique, thus producing a CPS interpreter. Because HOL4 terms, which the interpreter is written in, correspond to pure ML, the interpreter may be further transformed into a CPS compiler to pure ML. We then make additional optimisations, including direct compilation of mutable Scheme variables into ML `ref` variables.

Proof of semantic preservation is a simulation argument, derivative of the correctness proof in Plotkin's work on the CPS transform [11]. The nature of the CPS transform as a refunctionalisation of the small-step semantics, however, reduces much of the simulation proof to a proof of correctness of refunctionalisation [1]. As a result, the proof of semantic preservation for Brack went smoothly, and in fact first-class continuations were surprisingly simple to verify, despite their apparently complex behaviour.

Lessons. The feasibility of proving semantic preservation for Brack, by deriving a CPS transform from its small-step semantics and using a simulation argument, is a testament to the value of the CPS transform for representing control *for the purposes of verification.* Given the smooth compiler verification experience our approach enabled, we recommend that projects developing verified compilers for languages with similar control features adopt a similar approach.

Acknowledgements. We thank the anonymous reviewers and Michael Lee for helpful comments. Swedish Research Council grant 2021-05165 funded Myreen.

[1] https://github.com/CakeML/cakeml/tree/master/compiler/scheme

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming* (Uppsala, Sweden) *(PPDP '03)*. Association for Computing Machinery, New York, NY, USA, 8–19. https://doi.org/10.1145/888251.888254

[2] Olivier Danvy. 2004. On Evaluation Contexts, Continuations, and the Rest of Computation. (02 2004).

[3] Olivier Danvy. 2008. Defunctionalized interpreters for programming languages. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, BC, Canada) *(ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 131–142. https://doi.org/10.1145/1411204.1411206

[4] Olivier Danvy and Lasse R. Nielsen. 2001. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Florence, Italy) *(PPDP '01)*. Association for Computing Machinery, New York, NY, USA, 162–174. https://doi.org/10.1145/773184.773202

[5] Mattias Felleisen and D. P. Friedman. 1987. A calculus for assignments in higher-order languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Munich, West Germany) *(POPL '87)*. Association for Computing Machinery, New York, NY, USA, 314. https://doi.org/10.1145/41625.41654

[6] Matthias Felleisen and Daniel P. Friedman. 1987. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts*. https://api.semanticscholar.org/CorpusID:57760323

[7] Matthias Felleisen, Daniel P Friedman, Eugene E Kohlbecker, and Bruce F Duba. 1986. Reasoning with continuations. In *LICS*, Vol. 86. 131–141.

[8] P. Graham. 1994. *On Lisp: Advanced Techniques for Common Lisp*. Prentice Hall.

[9] Hrutvik Kanabar, Samuel Vivien, Oskar Abrahamsson, Magnus O. Myreen, Michael Norrish, Johannes Åman Pohjola, and Riccardo Zanetti. 2023. PureCake: A Verified Compiler for a Lazy Functional Language. *Proc. ACM Program. Lang.* 7, PLDI, Article 145 (June 2023), 25 pages. https://doi.org/10.1145/3591259

[10] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 179–191. https://doi.org/10.1145/2535838.2535841

[11] G.D. Plotkin. 1975. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science* 1, 2 (1975), 125–159. https://doi.org/10.1016/0304-3975(75)90017-1

[12] John C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2* (Boston, Massachusetts, USA) *(ACM '72)*. Association for Computing Machinery, New York, NY, USA, 717–740. https://doi.org/10.1145/800194.805852

[13] Konrad Slind and Michael Norrish. 2008. A Brief Overview of HOL4. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics* (Montreal, P.Q., Canada) *(TPHOLs '08)*. Springer-Verlag, Berlin, Heidelberg, 28–32. https://doi.org/10.1007/978-3-540-71067-7_6

[14] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robby Findler, and Jacob Matthews. 2009. Revised6 Report on the Algorithmic Language Scheme. *Journal of Functional Programming* 19, S1 (2009), 1–301. https://doi.org/10.1017/S0956796809990074

[15] Guy L. Steele. 1978. *Rabbit: A Compiler for Scheme*. Technical Report. USA.