

# The CakeML Project's Quest for Ever Stronger Correctness Theorems

Magnus O. Myreen 

Chalmers University of Technology, Gothenburg, Sweden

---

## Abstract

The CakeML project has developed a proof-producing code generation mechanism for the HOL4 theorem prover, a verified compiler for ML and, using these, a number of verified application programs that are proved correct down to the machine code that runs them (in some cases, even down to the underlying hardware). The purpose of this extended abstract is to tell the story of the project and to point curious readers to publications where they can read more about specific contributions.

**2012 ACM Subject Classification** Theory of computation → Higher order logic

**Keywords and phrases** Program verification, interactive theorem proving

**Digital Object Identifier** 10.4230/LIPIcs.ITP.2021.1

**Category** Invited Paper

**Supplementary Material** <https://cakeml.org>

**Funding** This work was partially supported by the Swedish Foundation for Strategic Research, and by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

**Acknowledgements** I thank the ITP programme committee for inviting me to give this talk. I am grateful for comments on drafts of this text from Michael Norrish, Johannes Åman Pohjola, Andreas Löw, Yong Kiam Tan, Oskar Abrahamsson and Konrad Slind. I want to thank the team of CakeML contributors for making the CakeML project such a pleasure to work on. Finally, I would like to thank the late Mike Gordon whose enthusiasm for the CakeML project and belief in its contributions are still felt and appreciated to this day.

## 1 Motivation and Beginning

Around the year 2012, the CakeML project started as a reaction to the practice of using the following steps to produce verified applications using interactive theorem provers (ITPs).

1. Write functions in the logic of an ITP;
2. prove correctness properties of the functions as theorems in the ITP; and
3. ask the ITP to generate SML/OCaml/Haskell code based on the functions in the logic.

Scott Owens and my reaction was the following: the code generators in Step 3 ought to show *with theorems proved in the ITP* that the generated code has the same behaviour as the supplied functions, and the theorems should ideally be expressed in terms of a formal semantics for the programming language in question (SML, OCaml or Haskell). However, the code generators<sup>1</sup> in use at the time did not provide users with such theorems.

The first publication of the CakeML project [32] showed that it is possible to build a proof-producing code generator that automatically proves a theorem stating the relationship between the given function's behaviour and the behaviour of the generated ML code with respect to an operational semantics of a subset of SML.

---

<sup>1</sup> In Coq terminology, this code generation is called code extraction.



© Magnus O. Myreen;

licensed under Creative Commons License CC-BY 4.0

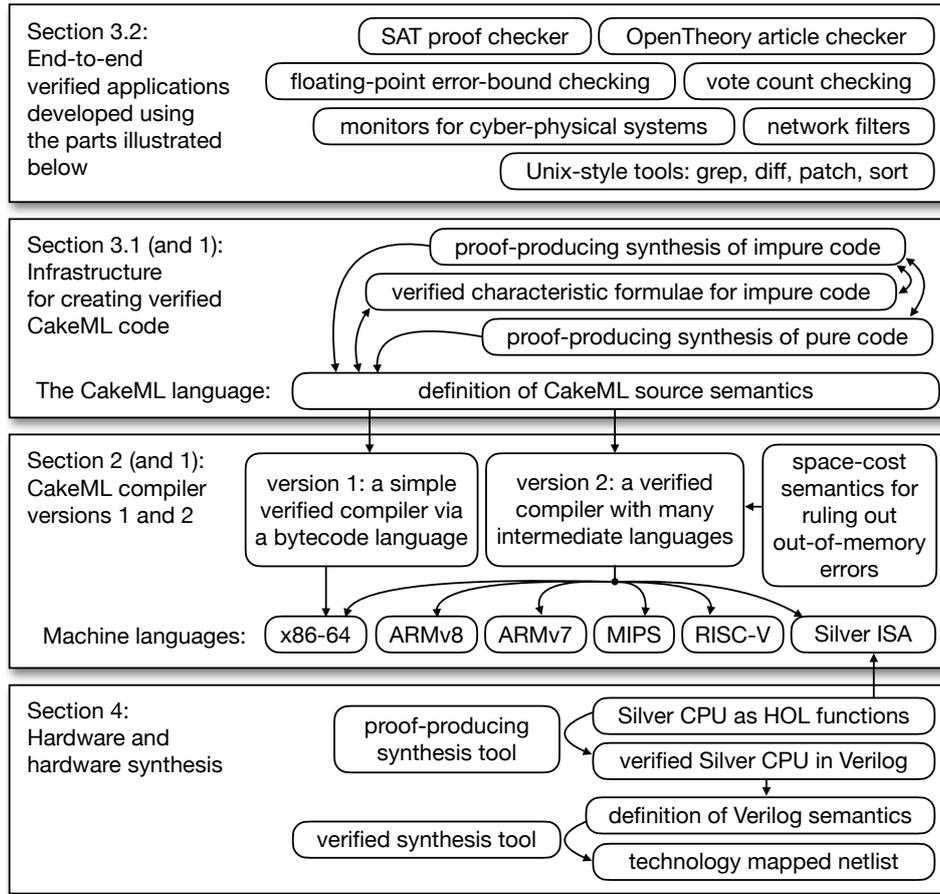
12th International Conference on Interactive Theorem Proving (ITP 2021).

Editors: Liron Cohen and Cezary Kaliszyk; Article No. 1; pp. 1:1–1:10

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** A diagram of the parts of the CakeML project described in this paper.

The introduction to this first publication made it clear that there really ought to be two parts to code generation: A) proof-producing translation of functions in the logic to functions deeply embedded in a programming language, and B) verified compilation for programs written in that programming language down to executable machine code. At the time when Scott Owens and I wrote this, we did not envision that we would deliver on B. However, that changed when Ramana Kumar got involved.

In the build up to the next publication, the project got its name “CakeML” which was initially short for Cambridge and Kent ML, following the tradition of naming SML implementations based on location names. At the time, Ramana and I were at Cambridge and Scott was at Kent. However, as the project grew, we have decided that the name CakeML is to be regarded as just a name rather than an abbreviation.

A close collaboration between Ramana Kumar, Scott Owens, Michael Norrish and myself delivered on B and resulted in the second paper *CakeML: A Verified Implementation of ML* [22], which explains how we created a verified read-eval-print loop implementing the CakeML operational semantics. The second paper has become the canonical reference for the CakeML project as a whole, even though the CakeML compiler has changed significantly since this first version, as will be described in the next section.

This paper gives a tour of the CakeML project and its ambition to achieve end-to-end verified code. Figure 1 illustrates the different parts of the project: the CakeML compiler is at the centre of the figure (Section 2); infrastructure and verified applications are above the compiler (Section 3); below the compiler, we have work that dives into the hardware layers (Section 4). Future work is mentioned in Section 5, but is not part of Figure 1.

## 2 Verified Compiler and Runtime

The CakeML project has put much focus on its verified compiler, which has evolved from a simple compiler to one with eight intermediate languages and several compilation passes within each intermediate language [41].

### 2.1 In-Logic Execution and Transportation of Correctness Results

The purpose of the CakeML compiler is to provide a way to generate machine code from given source programs in a way that allows users to transport any properties proved of the source code down to the generated machine code. This transportation feature is, of course, a desire in most compiler verification projects, but the CakeML project has taken this desired feature perhaps more literally than most other compiler verification projects. A property is fully transported when the final theorem is about the generated machine code and does not mention the compiler. In-logic execution of the compiler is necessary for this.

The CakeML compiler is written in such a way that it can be fully executed inside of an ITP. Such in-logic executions result in theorems of the form `compile input = machine_code`, where `input` is a concrete source-level program and `machine_code` is a concrete list of bytes. With concrete compilation results as theorems in the logic, and a standard result that `compile` only produces machine code that has identical behaviour (up to out-of-memory errors), one can transport properties  $P$  proved of the source-level program `input` down to the `machine_code`. Crucially, the in-logic compilation allows us to state the resulting theorem in terms of  $P$  and the semantics of `machine_code` *without any mention* of the complicated `compile` function.

These ideas are explained in Kumar et al. [21], which argues for the importance of ITP code generation mechanisms that prove the correctness down to the level of machine code.

### 2.2 Compiler Bootstrapping inside an ITP

The CakeML project is perhaps most well known for the fact that it is the first project to bootstrap a verified compiler entirely inside an ITP. Bootstrapping entails running a compiler on itself to generate machine code implementing itself. This seemingly circular concept can seem confusing at first. However, it follows quite directly from the concepts we have seen above: given a compiler function `compile` defined as a normal function in an ITP, we can apply the proof-producing code generator [32] to generate a behaviourally equivalent source program `compile_prog` and, to this source program, we can apply the in-logic compilation method described in the previous section. The result is a verified implementation of the `compile` function as concrete machine code. Separately from CakeML, the idea of bootstrapping a compiler inside an ITP has been described in a simple minimal setting [30].

### 2.3 The First Version: Compilation in a Read-Eval-Print Loop

The CakeML compiler was, from the start, an end-to-end compiler consisting of lexing, parsing, type inference and code generation. Each part was proved correct: the parser implementation was proved sound and complete with respect to a context-free grammar

for Standard ML syntax; our type inferencer was proved sound (and later complete [42] when Yong Kiam Tan joined the effort) w.r.t. the type system; we proved that no typed program will ever hit a runtime error in our operational semantics; and finally we also proved that any program that avoids runtime errors in the source semantics is compiled to identically behaving machine code, up to out-of-memory errors that can happen in the generated machine code. This caveat about out-of-memory errors is necessary because the CakeML source language does not pose any limits on the size of lists, arrays or integers, but the compilation target, x86-64 machine code operates in a finite state space.

The first version of the compiler was bootstrapped and used as a component in a verified x86-64 implementation of a read-eval-print loop for the CakeML language [22]. In this first version, the code generator was simple: it compiled to a stack-based bytecode language, which mapped quite directly to short snippets of x86-64 machine code. The implementation did include a bignum library [31] and a verified garbage collector [28]. The verification of the type inferencer built on prior work on unification [23].

The most challenging part of the first version was perhaps the dynamic compilation aspect: the compiler is repeatedly executed at runtime by the read-eval-print loop, which affected the compiler proofs [18]. We made use of proof methods developed in the context of verification of a just-in-time compiler [29].

## 2.4 A New Optimising Compiler Backend

Even before the first compiler version was ready, a plan was shaping up to do better both in terms of proof methodology and compiler implementation. We also decided to focus on more conventional static ahead-of-time compilation rather than dynamic compilation.

The proof methodology was improved by switching from relational big-step semantics to functional big-step semantics [35]. Semantics written in functional big-step style take the form of clocked interpreter functions that neatly fit with proof-by-rewriting in higher-order logic. This style of semantics allowed us to conveniently prove preservation of terminating and non-terminating behaviour using only one simulation proof per compiler phase. Most simulation proofs are in the direction of compilation, i.e. forward.

The compiler implementation was redone almost entirely. The new design had several intermediate languages that were designed with optimisations in mind. The new implementation avoided compiling via a stack-based bytecode, and instead used graph-colouring based register allocation. We also took care to compile curried functions into efficient code [36]. At this point Yong Kiam Tan and Anthony Fox had become active in the project. Yong Kiam Tan made significant contributions to the lower-level languages, including register allocation, compilation of calling conventions and the assembler. Anthony Fox was crucial in making the new compiler target five machine languages [11]: 64-bit x86-64, ARMv8, MIPS and RISC-V, and 32-bit ARMv7; later a sixth (the Silver ISA) was added, as will be described in Section 4.

The new compiler implementation has offered plenty of opportunities for student projects and experimental extensions. Students have, for example, contributed new optimisations [3], infrastructure for visualisation of the compiler’s transformations [16] and even a generational copying garbage collector [9]. One of the experimental extensions has explored giving CakeML a flexible fast-math-compatible floating-point semantics [5].

The most up-to-date description of the CakeML compiler is Tan et al. [41]. That paper includes benchmarks comparing the performance of the CakeML generated code with code from other ML compilers. The new CakeML compiler generates code with performance numbers that are in the same ballpark as the performance numbers for code compiled with Poly/ML, SML/NJ and native-code compiled OCaml.

## 2.5 A Space Cost Semantics

The CakeML compiler’s correctness statement has always contained an irritating get-out clause: the CakeML compiler’s correctness statement allows the generated machine code to exit early due to out-of-memory errors. As mentioned earlier, out-of-memory errors cannot be ruled out in general since the source semantics has no bounds on the size of data and numbers, but the target machine code runs in settings where memory is finite.

To address this wart in the compiler correctness theorem, we have defined a space cost semantics [15] that can be used to prove tight memory bounds for concrete CakeML programs. The cost semantics has been proved sound w.r.t. CakeML’s compiler and enables transfer of liveness properties proved for the source code down to liveness properties of the generated machine code.

## 3 Verified Applications

In parallel with the development of the compiler, we improved our techniques for generating CakeML source code and post hoc verification of manually written CakeML code.

### 3.1 Characteristic Formulae and Improved Code Generation

In the beginning, we could only produce verified CakeML code using the proof-producing synthesis tool mentioned in Section 1. This tool could initially only target the pure part of the CakeML language. However, the CakeML language includes SML-style impure features such as exceptions and mutable state in the form of references, arrays, and byte arrays. We did take some early steps towards support for impure CakeML code for a journal version [33] of the original synthesis paper, but the solution was ad hoc and unsatisfactory.

Separately from CakeML, Arthur Charguéraud developed a separation logic style reasoning framework called Characteristic Formulae for ML (CFML) for reasoning about OCaml code in Coq [8], and in 2016, I had the fortune of having Armaël Guéneau, who had worked with CFML, visit me for an internship. During his five-month internship, Guéneau produced a CFML variant in HOL for CakeML, and developed and proved it sound w.r.t. CakeML operational semantics. This CakeML adaption [13] of CFML covered every aspect of the CakeML language: (mutually recursive) functions as values, mutable state, exceptions and even reasoning about I/O through CakeML foreign function interface.

Guéneau’s original work on Characteristic Formulae (CF) for CakeML has been extended and used for several applications. A file-system model was built on top of it in order to allow us to reason about file accessing programs [10]. For this work, Johannes Åman Pohjola improved CakeML’s foreign-function interface. Later, Son Ho contributed an important forward simulation tactic for separation logic CF proofs (unpublished). Most recently, the CF framework was extended to handle correctness proofs of non-terminating programs [4]; this extension makes it possible to prove liveness properties for diverging (productive and non-productive) CakeML programs. A simplified version of this Hoare logic for non-termination has been proved sound and complete.

Once CF was well developed, we revisited our desire for proof-producing code generation that targets CakeML’s impure features. We took inspiration from how effectful computations are modelled using monads in Haskell and developed techniques for creating impure CakeML code from monadic HOL functions [2]. Our method builds on the separation logic setup provided by CF for CakeML. The fact that CF for CakeML and our code synthesis tool share infrastructure meant that we were able to provide links by which results proved in

one system could be transferred for use in the other. The speed of the CakeML compiler binary was significantly improved by this access to the impure features of CakeML. In a separate improvement to the code generator, some reduction in code bloat was provided by new techniques for code generated for case expressions [43].

In parallel to the work described above which took place in HOL4, Lars Hupel and Tobias Nipkow developed a different CakeML code generator [17] in Isabelle/HOL. This code generator differed from the one in HOL4 in that the Isabelle/HOL version was (for most part) a once-and-for-all verified tool, while the code generator in HOL4 was (for most part) a proof-producing tool. The bridge between Isabelle/HOL and HOL4 was provided by the Lem tool [27]. Most parts of the CakeML source language semantics are specified in the Lem language and, from this prover-independent specification, the Lem tool generates definitions in HOL4 and Isabelle/HOL.

### 3.2 Example Applications

The tools described above have been used to create end-to-end verified applications:

- We have verified a number of Unix-like tools such as grep, sort, cat, diff, patch, and a word frequency counter. These examples were mostly developed to showcase features of our file system models and CF setup.
- From early on [34], there was a drive to develop a verified proof assistant for higher-order logic based on CakeML. Ramana Kumar made significant progress in this direction [18, 20]. In the process, a new definition mechanism for higher-order logic (proposed by Rob Arthan) was proved sound [19]. Later, this line of work was continued when Arve Genggelbach and Johannes Åman Pohjola [37] proved consistency of ad-hoc overloading, as allowed by constant definitions in Isabelle/HOL.
- There has been work on verified checkers of different kinds: a checker for OpenTheory article files [1], a checker for proof certificates produced by SAT solvers [40], a checker for vote counting [12], and a checker for floating-point error bounds [6].
- Formally specified security components, such as filters, monitors, and attestation schemes, are being synthesized to CakeML running on seL4 [38, 39]; the intent is to thereby improve the security of legacy embedded systems.
- A verified CakeML application was also developed to be a verified monitor for cyber-physical systems [7].

## 4 Extending down into Hardware

The CakeML project has also dived into the hardware levels of computing systems. Our adventure in the hardware world was motivated by two questions:

1. Can our end-to-end correctness theorems be extended beyond the level of machine code and into the hardware layers? (And if so, down to what layer?)
2. Can the techniques that we have developed for proof-producing code generation and compiler verification be adapted to work for the creation of verified hardware?

Andreas Löw has explored these questions in his research. For question 2, he formalised a subset of Verilog in HOL4 and built a proof-producing HOL-function-to-Verilog generator [25] using ideas from the proof-producing CakeML code generator.

For question 1, Andreas Löw defined a small CPU, called Silver, as HOL functions in the subset understood by his Verilog code generator, and proved correctness of the CPU w.r.t. a specification of the custom instruction set architecture (ISA) that it supports. The ISA that

the Silver CPU implements was added as a target to the CakeML compiler by Anthony Fox. Finally, a number of CakeML developers helped verify a minimal implementation of a file system written in Silver machine code. Together these components allowed us to create a *verified stack* where end-to-end correctness theorems extend from high-level specifications all the way down to the Verilog implementation of the CPU that executes the machine code that the CakeML compiler produces, with all layers in between verified [26]. In short, the result was a *Silver platform for CakeML programs to stand on*.

The exploration of questions 1 and 2 continues: Andreas Lööw has built a verified Verilog-to-netlist compiler [24] that can compile (a subset of) the Verilog produced by his first tool. The netlists are technology mapped netlists for FPGAs.

## 5 Conclusions and Future Directions

The CakeML project started off as a simple reaction to the state of code generators/extractors and has since snowballed into an ambitious compiler verification project. The core aim has remained the same: to provide users with tools that allow transfer of properties proved about functions in logic down to concrete executable code (machine code or even hardware).

### 5.1 Reflection

What made it possible for the CakeML project to develop so far? There are, of course, many factors at play and no definite answers, but here are some points I believe are important:

- *Struck a chord*. I believe the end-to-end correctness theorems that the project has as its core aim inspires people to join, and functional programming (and reasoning about functional programs) is of interest to many with relevant backgrounds.
- *Luck*. We were fortunate early on: the initial group of people worked well together which was vital for getting the project off the ground.
- *No initial funding and no named leader*. The CakeML project started and ran for several years without any CakeML-specific funding, which meant that there was no named project leader and people were contributing to the project due to their own interest rather than because they were paid to work on this project.

### 5.2 What Next?

When something has been built well, I believe it has potential to be a platform to build on. One aspect that I am keen to explore is the use of CakeML and its compiler as part of other projects, e.g. projects that develop verification tools or compilers for languages other than ML. In this direction, there is on-going work within the CakeML project to develop a compiler for a clean low-level imperative language using parts of the CakeML compiler; and, similarly, there is another on-going project developing a verified compiler for a Haskell-inspired functional language using the CakeML compiler as a component. There is also on-going work on developing a verified compiler for a choreographic language [14].

---

#### References

- 1 Oskar Abrahamsson. A verified proof checker for higher-order logic. *Journal of Logical and Algebraic Methods in Programming*, 112:100530, 2020. doi:10.1016/j.jlamp.2020.100530.
- 2 Oskar Abrahamsson, Son Ho, Hrutvik Kanabar, Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Yong Kiam Tan. Proof-producing synthesis of CakeML from monadic HOL functions. *Journal of Automated Reasoning (JAR)*, 2020.

- 3 Oskar Abrahamsson and Magnus O. Myreen. Automatically introducing tail recursion in CakeML. In Meng Wang and Scott Owens, editors, *Trends in Functional Programming (TFP)*, volume 10788 of *LNCS*, pages 118–134. Springer, 2017. doi:10.1007/978-3-319-89719-6\_7.
- 4 Johannes Aman Pohjola, Henrik Rostedt, and Magnus O. Myreen. Characteristic formulae for liveness properties of non-terminating CakeML programs. In *Interactive Theorem Proving (ITP)*. LIPICS, 2019.
- 5 Heiko Becker, Eva Darulova, Magnus O. Myreen, and Zachary Tatlock. Icing: Supporting fast-math style optimizations in a verified compiler. In *Computer Aided Verification (CAV)*. Springer, 2019. doi:10.1007/978-3-030-25543-5\_10.
- 6 Heiko Becker, Nikita Zyuzin, Raphael Monat, Eva Darulova, Magnus O. Myreen, and Anthony Fox. A verified certificate checker for finite-precision error bounds in Coq and HOL4. In *Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2018.
- 7 Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. VeriPhy: verified controller executables from verified cyber-physical system models. In Jeffrey S. Foster and Dan Grossman, editors, *Programming Language Design and Implementation (PLDI)*, pages 617–630. ACM, 2018. doi:10.1145/3192366.3192406.
- 8 Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In *International Conference on Functional Programming (ICFP)*. ACM, 2011. doi:10.1145/2034773.2034828.
- 9 Adam Sandberg Ericsson, Magnus O. Myreen, and Johannes Åman Pohjola. A verified generational garbage collector for CakeML. *Journal of Automated Reasoning (JAR)*, 63, 2019. doi:10.1007/s10817-018-9487-z.
- 10 Hugo Férée, Johannes Åman Pohjola, Ramana Kumar, Scott Owens, Magnus O. Myreen, and Son Ho. Program verification in the presence of I/O – semantics, verified library routines, and verified applications. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, 2018. doi:10.1007/978-3-030-03592-1\_6.
- 11 Anthony C. J. Fox, Magnus O. Myreen, Yong Kiam Tan, and Ramana Kumar. Verified compilation of CakeML to multiple machine-code targets. In Yves Bertot and Viktor Vafeiadis, editors, *Certified Programs and Proofs (CPP)*, pages 125–137. ACM, 2017. doi:10.1145/3018610.3018621.
- 12 Milad Ketab Ghale, Dirk Pattinson, and Ramana Kumar. Verified certificate checking for counting votes. In Ruzica Piskac and Philipp Rümmer, editors, *Verified Software. Theories, Tools, and Experiments (VSTTE)*, volume 11294 of *LNCS*. Springer, 2018. doi:10.1007/978-3-030-03592-1\_5.
- 13 Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified characteristic formulae for CakeML. In Hongseok Yang, editor, *European Symposium on Programming (ESOP)*, volume 10201 of *LNCS*. Springer, 2017. doi:10.1007/978-3-662-54434-1\_22.
- 14 Alejandro Gómez-Londoño. Choreographies and cost semantics for reliable communicating systems, 2020. Licentiate thesis, Chalmers University of Technology.
- 15 Alejandro Gómez-Londoño, Johannes Åman Pohjola, Hira Taqdees Syeda, Magnus O. Myreen, and Yong Kiam Tan. Do you have space for dessert? A verified space cost semantics for CakeML programs. *Proc. ACM Program. Lang. (OOPSLA)*, 4:204:1–204:29, 2020. doi:10.1145/3428272.
- 16 Rikard Hjort, Jakob Holmgren, and Christian Persson. The CakeML compiler explorer - tracking intermediate representations in a verified compiler. In Meng Wang and Scott Owens, editors, *Trends in Functional Programming (TFP)*, volume 10788 of *LNCS*, pages 135–148. Springer, 2017. doi:10.1007/978-3-319-89719-6\_8.
- 17 Lars Hupel and Tobias Nipkow. A verified compiler from Isabelle/HOL to CakeML. In Amal Ahmed, editor, *European Symposium on Programming (ESOP)*, volume 10801 of *LNCS*, pages 999–1026. Springer, 2018. doi:10.1007/978-3-319-89884-1\_35.
- 18 Ramana Kumar. *Self-compilation and self-verification*. PhD thesis, University of Cambridge, 2016.

- 19 Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. HOL with definitions: Semantics, soundness, and a verified implementation. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving (ITP)*, volume 8558 of *LNCS*, pages 308–324. Springer, 2014. doi:10.1007/978-3-319-08970-6\_20.
- 20 Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. Self-formalisation of higher-order logic – semantics, soundness, and a verified implementation. *Journal of Automated Reasoning (JAR)*, 56(3):221–259, 2016. doi:10.1007/s10817-015-9357-x.
- 21 Ramana Kumar, Eric Mullen, Zachary Tatlock, and Magnus O. Myreen. Software verification with ITPs should use binary code extraction to reduce the TCB (short paper). In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving (ITP)*, volume 10895 of *LNCS*, pages 362–369. Springer, 2018. doi:10.1007/978-3-319-94821-8\_21.
- 22 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *Principles of Programming Languages (POPL)*. ACM, 2014. doi:10.1145/2535838.2535841.
- 23 Ramana Kumar and Michael Norrish. (nominal) unification by recursive descent with triangular substitutions. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *LNCS*, pages 51–66. Springer, 2010. doi:10.1007/978-3-642-14052-5\_6.
- 24 Andreas Löw. Lutsig: a verified Verilog compiler for verified circuit development. In Catalin Hritcu and Andrei Popescu, editors, *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 46–60. ACM, 2021. doi:10.1145/3437992.3439916.
- 25 Andreas Löw and Magnus O. Myreen. A proof-producing translator for Verilog development in HOL. In Stefania Gnesi, Nico Plat, Nancy A. Day, and Matteo Rossi, editors, *Proceedings of the 7th International Workshop on Formal Methods in Software Engineering, FormaliSE@ICSE 2019, Montreal, QC, Canada, May 27, 2019*, pages 99–108. IEEE / ACM, 2019. doi:10.1109/FormaliSE.2019.00020.
- 26 Andreas Löw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. Verified compilation on a verified processor. In *Programming Language Design and Implementation (PLDI)*. ACM, 2019. doi:10.1145/3314221.3314622.
- 27 Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *International Conference on Functional Programming (ICFP)*. ACM, 2014. doi:10.1145/2628136.2628143.
- 28 Magnus O. Myreen. *Formal verification of machine-code programs*. PhD thesis, University of Cambridge, 2009.
- 29 Magnus O. Myreen. Verified just-in-time compiler on x86. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Principles of Programming Languages (POPL)*. ACM, 2010.
- 30 Magnus O. Myreen. A minimalistic verified bootstrapped compiler (proof pearl). In Catalin Hritcu and Andrei Popescu, editors, *Conference on Certified Programs and Proofs (CPP)*, pages 32–45. ACM, 2021. doi:10.1145/3437992.3439915.
- 31 Magnus O. Myreen and Gregorio Curello. Proof pearl: A verified bignum implementation in x86-64 machine code. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs (CPP)*, volume 8307 of *LNCS*, pages 66–81. Springer, 2013. doi:10.1007/978-3-319-03545-1\_5.
- 32 Magnus O. Myreen and Scott Owens. Proof-producing synthesis of ML from higher-order logic. In *International Conference on Functional Programming (ICFP)*, pages 115–126. ACM Press, 2012. doi:10.1145/2364527.2364545.
- 33 Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming (JFP)*, 24(2-3):284–315, 2014. doi:10.1017/S0956796813000282.

- 34 Magnus O. Myreen, Scott Owens, and Ramana Kumar. Steps towards verified implementations of HOL light. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving (ITP)*, volume 7998 of *LNCS*, pages 490–495. Springer, 2013. doi:10.1007/978-3-642-39634-2\_38.
- 35 Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. In Peter Thiemann, editor, *European Symposium on Programming (ESOP)*, LNCS. Springer, 2016. doi:10.1007/978-3-662-49498-1\_23.
- 36 Scott Owens, Michael Norrish, Ramana Kumar, Magnus O. Myreen, and Yong Kiam Tan. Verifying efficient function calls in CakeML. *Proc. ACM Program. Lang.*, 1(ICFP), September 2017. doi:10.1145/3110262.
- 37 Johannes Aman Pohjola and Arve Gengelbach. A mechanised semantics for HOL with ad-hoc overloading. In Elvira Albert and Laura Kovacs, editors, *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 73 of *EPiC Series in Computing*, pages 498–515. EasyChair, 2020.
- 38 Konrad Slind. Specifying message formats with contiguity types. In *Interactive Theorem Proving (ITP)*, Leibniz International Proceedings in Informatics. Schloss Dagstuhl Leibniz-Zentrum für Informatik, Dagstuhl Publishing, 2021.
- 39 Konrad Slind, David S. Hardin, Johannes Åman Pohjola, and Michael Sproul. Synthesis of verified architectural components for autonomy hosted on a verified microkernel. In *Hawaii International Conference on System Sciences*, January 2020.
- 40 Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. cake\_lpr: Verified propagation redundancy checking in CakeML. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2021. doi:10.1007/978-3-030-72013-1\_12.
- 41 Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29, 2019. doi:10.1017/S0956796818000229.
- 42 Yong Kiam Tan, Scott Owens, and Ramana Kumar. A verified type system for CakeML. In *Implementation and Application of Functional Programming Languages (IFL)*. ACM Press, 2015. doi:10.1145/2897336.2897344.
- 43 Thomas Tuerk, Magnus O. Myreen, and Ramana Kumar. Pattern matches in HOL: A new representation and improved code generation. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving (ITP)*, volume 9236 of *LNCS*, pages 453–468. Springer, 2015. doi:10.1007/978-3-319-22102-1\_30.