First Year Report and Thesis Proposal

Ramana Kumar (rk436)

June 22, 2012

First Year Report

I work in the area of mechanised reasoning. I learned how to use the HOL4 theorem proving system in 2009 during the Honours year of my undergraduate degree. Working towards the MPhil ACS last year, I dabbled with the Isabelle/HOL, Coq, and Agda systems, and completed a project, the basis of my final essay, using HOL4. During the past year, I joined the mailing lists for all of these tools and got to know them better. My initial research proposal was on translating formal developments and tool libraries across logics and between systems.

Currently, I am working on a project using HOL4 to build and verify a compiler for a higher order functional language, preserving semantics from high level concrete syntax to machine code or a hardware model. This project forms the core of my thesis proposal, and work towards it is described in the final section of my first year report. The rest of my work during the first year falls into two categories: multi-system reasoning, as in my initial proposal, primarily using the OpenTheory proof exchange format, and forays into the world of reasoning using dependent types.

1 OpenTheory

The OpenTheory project[1], started by Joe Hurd, aims to enable formal developments to be shared between the many theorem proving systems that implement higher order logic, such as HOL4, HOL Light, Isabelle/HOL, and ProofPower. The project defines the concept of a *theory package* [20], a composable reusable formalisation, supported by a format, *article files* [21], for encoding proofs, and provides a *standard library* [22] of common concepts as defined in higher order logic on which to base further developments.

To use OpenTheory, a theorem proving system must be able to communicate in the standard format of theory packages and article files. Currently, few systems can. My work this year included improving OpenTheory support in HOL4 and HOL Light. HOL Light was the first system that, thanks to Joe Hurd, could write article files for proofs created with the system. Now both HOL4 and HOL Light can both read and write article files.

An OpenTheory theory package may be used to represent a collection of results about a particular topic, but it can also encode a transient result or even an unproved conjecture, and thus can be used as a communication medium for proof tools. The OpenTheory standard library fixes the background knowledge required for this communication to be meaningful. I discussed this idea on the OpenTheory mailing list, and eventually wrote a "rough diamond" (short paper) with Joe Hurd called "Standalone Tactics using OpenTheory" that was accepted for publication

[1] `http://www.gilith.com/research/opentheory/`

in the proceedings of the Third International Conference on Interactive Theorem Proving (ITP 2012). I will present the paper at the conference in August. The work behind this paper includes standalone tactics "extracted" from existing functionality in HOL4 and HOL Light; I also wrote a small "from scratch" standalone tactic in Haskell.

Of the HOL-based theorem provers, Isabelle/HOL has perhaps the largest community and would be a great source of useful tools and theories. Brian Huffman implemented OpenTheory article reading for Isabelle/HOL. However, article writing remains unimplemented and is theoretically non-trivial because of Isabelle's architecture (in particular, the way it constructs proofs) and because Isabelle/HOL's logic includes a feature (type classes) not present in other HOL systems. I spent a bit of time this year discussing approaches to the problem of writing Isabelle proofs in OpenTheory format with some Isabelle developers, and may continue this discussion in the future (for example at ITP 2012).

2 Dependent Types

I spent some of my time this year using and learning about Agda, a programming language and environment based on dependent type theory. I attended the fifteenth Agda Implementors' Meeting (later editions will be called "Agda Intensive Meeting") in February, talked with the community about potential research directions, and made small contributions to the online documentation.

My focus when using Agda was the idea of writing proof tools with stronger type information enabling easier and better use and reuse. To take a concrete example, HOL4 is written in ML, there is an ML type `term` of HOL terms and an ML type `thm` of HOL theorems. Functions in ML of type `term` $\rightarrow$ `thm` are called "conversions". A conversion's type does not say exactly what it does: a conversion that takes a term and reduces all beta-redexes (that is, returns a theorem saying the original term is equal to another one with no beta-redexes) has the same type as a conversion that does an eta-expansion. Therefore when writing composite conversions, one must rely on informative names and documentation more than static typing information, though only the latter is guaranteed to be up to date and mechanically checked.

Writing conversions with dependent types allows the possibility that the types are more informative, for example `thm` could be augmented with the information that it's a theorem of a specific form, such as an equality with the input term on the left hand side. I attempted to write a dependently-typed conversion in Agda, but got bogged down in details such as how to represent terms and theorems.

As it turns out, the problem of scarce static information about proof tools is being investigated by Stampoulis and Shao in work on VeriML [42, 43] with applications to tactic-based theorem proving in Coq. It would be interesting to see whether their ideas can be applied to systems like HOL4 or OpenTheory where the object logic is not itself dependently-typed. If so, a dependently typed programming language would be a good basis for building high quality standalone tactics for use across HOL-based systems.

3 MiniML Compiler

Background At ITP 2011, Michael Norrish presented his mechanisation of computability theory in HOL4. Subsequent discussion raised the idea of deeply embedding the semantics of a real programming language in higher order logic, to complement Norrish's embedding of the lambda calculus (and eventually to prove the two equivalent). This year, Scott Owens defined the syntax and semantics of a pure subset of the ML programming language, currently called "MiniML",

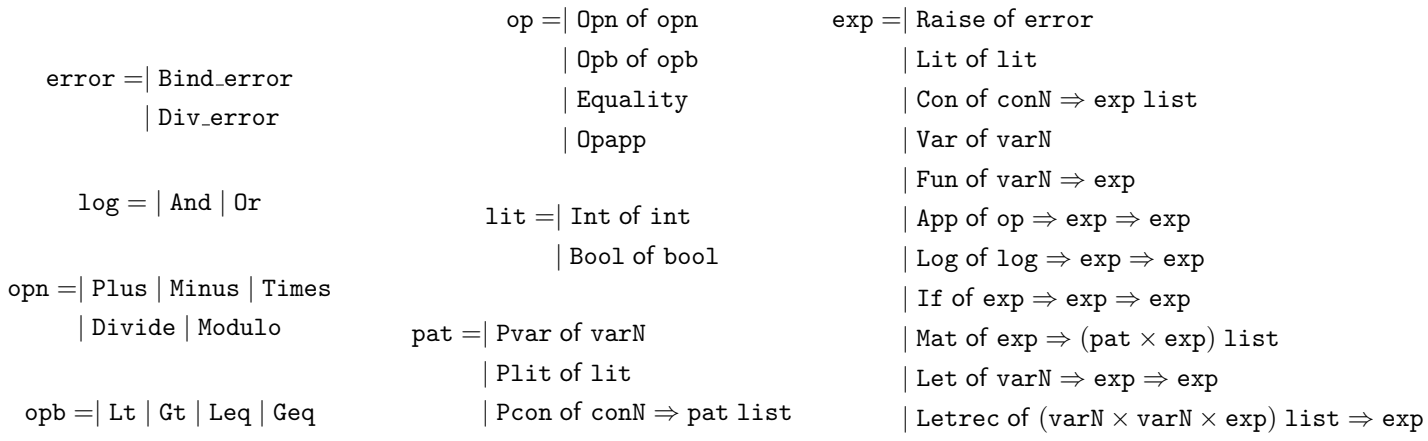

Figure 1: MiniML Syntax

in HOL and proved the type soundness theorem for its type system. The abstract syntax is shown in Figure 1. Magnus Myreen wrote a HOL4 library for automatically translating (certain computable) HOL functions to MiniML programs together with a proof that the semantics of the program matches the function. This work reduces the trust required by the common practice, which I call the *printing approach*, of printing the equations used to define a recursive function in HOL in the syntax of, say, Standard ML, and running the result with the hope that any theorems about the HOL function still apply to the ML version. The more trustworthy *translation approach* is described in a paper by Myreen and Owens to appear in the 17th ACM SIGPLAN International Conference on Functional Programming 2012.

So we can automatically translate computable functions in higher order logic to semantically equivalent MiniML programs. But how can we run the resulting programs, maintaining our semantic assurance? We can take the printing approach and run MiniML programs with an existing compiler for SML or OCaml, thereby relying on the correctness of the printer, which depends on the semantics of SML/OCaml, and relying on the correctness of the compiler. Or we can take another step in the translation direction, from functions in logic towards runnable programs, with a compiler from MiniML programs to machine code. Given semantics for the machine code (e.g., Fox and Myreen [14]), we need only that this compiler is semantics-preserving or that its output can be validated.

I have begun work on a semantics-preserving compiler for MiniML. (The input is well-typed MiniML abstract syntax; there is no parser, type inferencer, or type checker, but all are planned.) The target is a bytecode, like the one used by Myreen et al. [34], for a stack machine, and the plan is to use an approach similar to theirs to compile the bytecode to machine code with proof. The compiler is written in HOL4 (and Lem [36]) and supports everything in MiniML[2], in particular inductive datatypes, pattern matching, and mutually recursive higher order functions.

One can run the compiler (in an untrusted way) using the printing approach. Using this plus an SML interpreter for the bytecode I have run the MiniML compiler on some rudimentary tests, including implementations of quicksort and McCarthy's 91 function.

[2]almost: division and modulo and arbitrary precision integer arithmetic need more work on the bytecode side

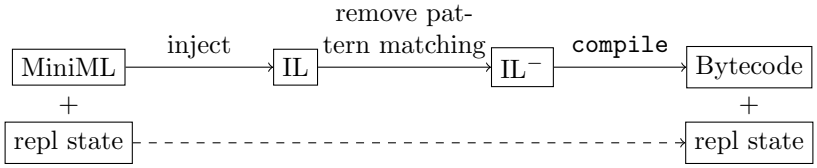

Figure 2: MiniML Compiler High Level Architecture

$$\texttt{call\_ctxt} = \mid \texttt{NonTail} \mid \texttt{Tail of num} \Rightarrow \texttt{num}$$

$$\begin{aligned}\texttt{ctbind} = \mid\ & \texttt{CTLet of num} \mid \texttt{CTArg of num} \\ \mid\ & \texttt{CTEnv of num} \mid \texttt{CTRef of num}\end{aligned}$$

$$\texttt{ctenv} = \texttt{varN} \mapsto \texttt{ctbind}$$

$$\begin{aligned}\texttt{cs} = \langle & \texttt{env} : \texttt{ctenv} \\ ; & \texttt{sz} : \texttt{num} \\ ; & \texttt{code} : \texttt{bc\_inst list} \\ ; & \texttt{tail} : \texttt{call\_ctxt} \\ ; & \texttt{code\_length} : \texttt{num} \\ ; & \texttt{next\_label} : \texttt{num} \\ ; & \texttt{inst\_length} : \texttt{bc\_inst} \to \texttt{num} \\ ; & \texttt{decl} : (\texttt{ctenv} \times \texttt{num})\ \texttt{option}\rangle\end{aligned}$$

Figure 3: Compiler State Record

Implementation The architecture of the compiler is currently as in Figure 2. The intermediate language IL differs from MiniML by allowing multi-argument (anonymous or named) functions, explicit declaration of non-recursive (in addition to recursive) named functions, and untyped projections (in addition to pattern matching). IL$^-$ is the same as IL except without pattern matching. In addition to expressions, MiniML allows top-level declarations for binding names persistently. The repl state contains a map from the names of declared variables to stack locations.

Intermediate languages, each with a suitable semantics, facilitate the definition of source-to-source translations and optimisations. The purpose of IL is currently just to facilitate pattern-matching compilation, a somewhat naive transformation to conditional tests in top-to-bottom left-to-right order, with the slight optimisation of saving shared success and failure branches in zero-argument functions bound outside the tests. This transformation will be a fallback for a more sophisticated unverified compilation to good decision trees [28] (or something similar) coupled with translation validation [41] for assurance.

The translation, `compile`, from IL$^-$ takes each kind of expression in IL$^-$ to a sequence of bytecode instructions. The most complicated cases are, unsurprisingly, closures (that is, creating function values) and function calls. The `compile` phase ensures that a function call in tail position is done properly (that is, as a jump, reusing the old return pointer). The information required to do tail call optimisation must be threaded through the compiler, and affects let-binding since a tail call within a binder needs to discard the bound values before jumping. The `compile` function has type `cs` $\to$ `ILexp` $\to$ `cs`; the contents of a compiler state `cs`, which include the generated bytecode, are shown in Figure 3 and described below.

The `env` component of the compiler state tracks the location (e.g., on the stack) of bound source-language variables and facilitates the compilation of variable references. The `sz` component tracks the number of objects in the current stack frame, also for the purpose of enabling variable references. The `code` component is the emitted bytecode. Putting the code into the state enables compilation to add to the end of existing compiled code, which is useful for a repl.

The `tail` component tracks whether compilation is happening in tail position; if so, both the number of let-bound variables and the number of arguments passed to the enclosing function are stored.

The `code_length` component is the length of the `code` list. The bytecode has no textual labels; rather, each instruction is implicitly labelled by the sum of the lengths of all preceding instructions. The `inst_length` component (which is never modified) measures the length of bytecode instructions for the purpose of this labelling, and `next_label` stores the accumulated length, that is, the label of the next instruction to be emitted. These three components are used to allow jumps to be compiled correctly. Finally, the `decl` component is used when compilation is of a declaration rather than an expression, as a signal to keep certain bound variables on the stack and enabling `env` to be extended correctly.

Proofs The compiler has not been proved correct. Therefore I will not describe it any further because it is likely to change, both to correct errors and to simplify proofs. Indeed, this process of restructuring definitions to ease the proof effort has already begun. From Easter term onwards I have been working on both figuring out the precise statement and trying to prove the simplest version of a "correctness" proof for the MiniML compiler. The rough statement is: if the MiniML semantics evaluates a program to a value, then the bytecode semantics evaluates the result of compiling that program to an equivalent value. This statement is weak because it assumes the program is well-typed and terminating; a more ambitious result would go in the other direction, relating all results of the compiler back to the input language semantics.

The bulk of the proof effort so far has been on properties of the semantics of the intermediate language IL. Similar to Owens's semantics for MiniML, I am using environment semantics for IL: specifically, (big-step) evaluation is defined as a relation between an environment (mapping variables to values), an expression, and a result value.

An example of a desirable property of the semantics is the *free variables only* property: adding or removing bindings in the environment that are not in the free variables of the expression does not affect the result. This property, like any involving environments, is complicated by the fact that the result value may be a closure which has captured the environment present at its creation. Therefore, "not affecting the result" cannot mean that the result is exactly the same but rather appropriately equivalent. Compiler optimisation phases that do more than rearrange environments will require a subtle treatment of what it means not to affect the result.

There is a spectrum of options for addressing the problem of formalising when result values are appropriately equivalent. What varies along the spectrum is the coarseness of the relation: how similar equivalent values must be. Let us consider two points at opposite ends:

1. Use the compiler. There are two approaches under this option:

   (a) Define the appropriate equivalence relation on values for the property being proved. This is "using the compiler" because the equivalence relation is tailored to the transformations the compiler actually performs, including any unforced choices. In the case of free variables only[3], we might say two closures are equivalent if the bodies are the same and equivalent values are found in the restriction of the saved environments to the body's free variables. But for a transformation that removes unused let-bindings, we would define another equivalence relation that allows closures' bodies to differ according to that transformation, or according to a narrow class of transformations.

   (b) Canonicalise all closures so that equivalent closures are actually equal. Ensure that the semantics only produces canonical values. This would work for the free variables

[3]This property is useful for transformations that, for example, introduce new code including let-bindings.

only property because we can canonicalise environments by restricting them to the body's free variables. However, picking canonical representatives of equivalence classes may not always be so easy.

2. Use semantics.

   Define a very coarse "semantic" equivalence relation on values that is likely to relate the results of any transformations appropriately. The coarsest such relation is *contextual equivalence* (e.g., the "operational equivalence" in Plotkin [40]). Usually, contextual equivalence relates expressions: two expressions are contextually equivalent if for all possible surrounding expressions (contexts) that yield an observable value (not a closure), inserting either expression yields the same value. The same idea can be used to relate values by considering contexts with a free variable rather than a hole for an expression. However, contextual equivalence is traditionally difficult to reason about because of the quantification over all contexts and because it is not syntax-directed. Thus, one might try to define a more usable but still semantically motivated relation.

I have experimented with all three approaches in parallel and am currently focussing on using 1a. My feeling is that 1b is simply a less elegant version version of 1a. Approach 2 would be desirable because then there is only one equivalence relation and various properties can be proved "once and for all". But it is difficult to define and use that relation. Defining usable approximations or versions of contextual equivalence is an area of past and active research, see for example [38, 37] or [18].

The current state of the proofs is as follows. I have made partial progress on the free variables only property (I had completed a version of it using approach 1b above, but since then MiniML has been refactored and I have switched to 1a.) I have also made partial progress on the correctness of the "inject" and "remove pattern matching" phases shown in Figure 2. I have yet to attempt a statement of correctness for `compile`, but that is worth doing soon to get a picture of how all the results will fit together.

Tool Development While working on correctness proofs, I have taken many opportunities to add results that look generally useful back to HOL4's standard library and to improve automation by marking more theorems as automatic rewrites. I wrote a small HOL4 library for proving that the universe of an inductive datatype is countable[4] (assuming all the types it is made up from are countable). I have also begun looking into ways to include recursion through finite maps in inductive datatypes, something not currently supported automatically by HOL4 but which I have found useful to do manually (because values can include environments, which are best represented as finite maps to values).

Improvement of the proof tools used in large scale verification work should be done, when necessary, alongside and driven by that work. There is an interplay between the structure of a formal development and the quality of the tools: one tries to play to the strengths of the tools available and but also to write custom tools for the recurring problems in the development. I suspect tool development might become an important part of the MiniML project. In this view I agree with Chlipala [7] that the application of formal methods to large systems like compilers requires good engineering and design encouraging proof automation.

Applications I have taken part in discussions about potential applications of the MiniML project. These include building an embedded device with security properties verified to a very low level.

[4]This was useful when experimenting with "canonicalising" closures and when environments were represented by lists, because it enables values to be totally ordered and hence environments to be sorted.

Thesis Proposal

The problem I propose to work on (described in Section 4) has compiler verification at its core. Work on compiler verification varies along a number of dimensions. Three are in common with work on compilers that are not necessarily verified:

Source What features are present in the input language? Does it have good theoretical properties? Is it amenable to formal methods? Is it popular and/or standardised?

Target Is the output language realistic: how far away is a running program or a complete application?

Quality Is the compiler efficient and usable? Does it handle bad input well? Does it optimise? Is the generated code small and efficient?

Three more dimensions are particular to verification:

Properties What is verified? E.g., memory safety, full functional correctness, resource usage, security, etc.

Strength When do the verification results apply? E.g., to whole programs only (no linked code), or to well-typed programs only? What are the assumptions?

Trustworthiness What is the case for believing the results? How were they proved? Are there unstated assumptions or blind spots due to abstraction?

The sticking points of my proposal are:

- A higher order functional source language,
- A target at least as realistic as machine code,
- Quality comparable to commonly used ML compilers,
- Properties including at least functional correctness, and,
- Trustworthiness to the level of LCF-style mechanical theorem proving.

Of course, going further along these dimensions and doing well on strength would be desirable.

4 Route to Dissertation

I describe the problem I propose to work on, why it's important, why it's new and exciting, and finally I sketch the work to be done over the remaining two years.

4.1 Problem

Some properties of computer systems are amenable to formal specification. The general problem I want to address is how to make a proof that a model of a system meets its specification mean more about the system running in reality. (There will always be a gap, and the philosophical inquiry about why is of great interest to me, but not the primary problem.)

More specifically, how can we build a complete system with useful behaviour and strong assurances about its behaviour underpinned by rigorous semantics? What tools and methods

for abstraction and modelling work? In the end I will produce some system including a verified compiler that is suitable for some application, and from that we will be able to conclude that the particular tools and methods used work there. However, I also hope to record why they worked, what did not work, and any other reusable ideas that might guide similar and further work on high assurance systems.

4.2 Importance

Formal methods applied to computer systems traditionally derive importance from *critical systems*, those where incorrect behavior threatens valuable things such as people's lives, health, or assets. The development of reusable methods for constructing systems with a high assurance of correctness is directly relevant to critical systems, and of course non-critical systems only stand to benefit as well. Of particular interest, given my goal of producing a *complete system*, are security properties, that is, where the correctness of the system depends on information or behaviour being withheld from unauthorised users.

Compilers play a special role in the verified system picture: they are general-purpose (a verified compiler can be used to raise assurance for many applications), and they are traditionally large and complex (hence hard to verify) but treated as semantically transparent (that is, assumed to be correct). Leroy [26] provides good motivation for verifying compilers in his introduction. The payoff for generality is largest when programs in the source language are amenable to verification, because results about them are what the compiler's correctness preserves. Hence it is important to work on a compiler for, essentially, the logic of a theorem prover: the input programs are primed for verification.

Moore [29] issued the development of a complete verified stack as a "grand challenge". The direct benefits of working on this challenge are high assurance systems and ideas about how to build them more easily and better. Such work also has the indirect benefit of directing improvement of the surrounding proof tools. Moore calls this project a *technology driver*.

4.3 Novelty

The following directions for research (especially when taken together) distinguish my proposal from existing work on compiler verification, which I look at in Section 5.

A Complete System It is common to verify one component of a system, perhaps a very sophisticated version of that component, and then assume the results will be suitable for inclusion in a complete system verification. By requiring a working complete system, albeit with simpler components, I hope to address the often neglected problem of making sure that results compose correctly. There is also an opportunity for programming language research, specifically how high and low level language features interact, and what are the minimal additions to a functional language to make it suitable as the sole programming language on an embedded system.

Pattern Matching There is little existing work on semantically sound pattern matching (Kirchner et al. [23] provide an exception). My proposal requires that this problem be addressed, and I plan to verify a simple pattern match compiler. It would also be nice to verify either an efficient pattern match compiler (e.g., by Augustsson [1] or Maranget [28]) or a validator of its output.

Higher Types The semantics of functions as values touches many aspects of the verification of a compiler for a language with that feature. If mutable state is also available, the

semantics become quite intricate, as Pitts [37] describes. Higher order functions also admit idiosyncratic optimisation possibilities, like uncurrying. Much of the existing work on compiler verification is for compilers of first order languages where both the compiler and the proofs can look quite different.

Optimisations Aiming to generate code comparably efficient to existing unverified compilers necessitates implementing and verifying/validating some non-trivial optimisation passes. Efficiency is not typically a primary goal in the existing work on compiler verification; I propose to go further than usual in this direction.

4.4 Plan

Methodological choices in compiler verification, even given my sticking points above, are many and often have a high cost of commitment without obvious guarantees of suitability. Examples include how syntax is represented (first order, higher order, nominal), the coarseness of the equivalence relations between values, the kinds of semantics used (substitution, environment, big or small step), and where to do translation validation rather than verification. These choices are often constrained by my use of higher order logic as implemented in HOL4, which is where MiniML lives. For those where the tradeoffs appear close, I plan to explore all options in parallel continually favouring the one that looks most promising. Therefore, it will pay to repeatedly take stock and look at the big picture; it is easy to get lost in the details of interactive theorem proving. With this in mind, possible milestones for the next two years include:

(Essential)

1. A weak ("forwards") but end-to-end correctness result.
2. Experiments and literature review to determine the optimisations that are critical for reasonable performance.
3. Implementing those optimisations and verifying/validating them. (Includes design of appropriate intermediate languages and other methodology.)
4. Experimentation to determine language extensions required for a complete hardware application, and implementation and verification/validation of those extensions.
5. A small but complete hardware application including I/O, with forwards correctness proved.

(Optional)

- A better pattern matching compiler.
- Self-compilation (bootstrapping). This might simply become possible automatically.
- Strong ("backwards") correctness, using determinism, and perhaps using coinductive big step semantics [27] to deal with divergence.
- More source language features, to match Standard ML (plus perhaps the Basis Library). The main things missing are I/O, mutable state, and handled exceptions.
- Compilation of a simple theorem prover (e.g., a stripped-down HOL Light).
- Compositional correctness and a linked code example.

5 Related Work

Compiler verification is an old problem: Dave [10] provides a bibliography of the subject from 1967 to 2003. More has been done since. My focus below is on work for higher order source languages and/or a low level target (that is, approaching a complete system).

Lambda Tamer Headed by Adam Chlipala, the Lambda Tamer project[5] addresses compiler verification and related issues in Coq, making full use of the dependent types and automation facilities in that envirnoment. It is close to my proposal in focusing on higher order typed source languages. Results include verified compilers from impure functional languages [7] and pure lambda calculus [6] to assembly code.

Source Higher order functional languages.

Target Idealised assembly code.

Quality Some optimisations like common subexpression elimination.

Properties Functional correctness.

Strength Assumes valid terminating input programs.

Trustworthiness Mechanical proof checking with Coq (similar in trustworthiness to HOL4).

CLI Stack The team at Computational Logic, Inc. produced a complete verified stack [4] from a simple operating system and high level language, Gypsy, via an assembly language, Piton, to a custom microprocessor, the FM8502. They used the Boyer-Moore theorem prover; the incarnation then was Nqthm, nowadays it is ACL2. In [29], Moore summarises what the CLI stack achieved and where it falls short.

Source Micro-Gypsy, a first order imperative language.

Target Machine code for FM8502.

Quality Basic.

Properties Functional correctness.

Strength Equivalence of the semantics to running the compiler, restricted to terminating runs.

Trustworthiness Mechanical proof checking with the Boyer-Moore prover, which has a larger trusted code base than LCF-style theorem provers.

CompCert Perhaps the most famous recent example of large scale compiler verification is the CompCert project[6] headed by Xavier Leroy [25, 26, 24].

Source C (a large subset thereof).

Target PowerPC machine code.

Quality Good, including many basic optimisations, dataflow analysis, and register allocation by graph colouring. The generated code has performance comparable to `gcc`'s.

Properties Functional correctness.

Strength Whole terminating programs only.

Trustworthiness Mechanical proof checking with Coq.

[5] `http://ltamer.sourceforge.net`
[6] `http://compcert.inria.fr`

The CerCo (Certified Complexity) project[7] aims to extend the ideas in CompCert to get guarantees about complexity. Thus it differs in the "Properties" dimension in also proving bounds on execution time and other resources.

Jitawa Myreen produced a verified just-in-time compiler [34, 31] from a first order functional language to machine code. As an application, Myreen and Davis [32] presented a verified theorem prover with a verified runtime environment. In related work, Myreen and Gordon describe *function extraction* [33] or "decompilation into logic" whereby machine code can be formally analysed by first extracting a function with equivalent behavior.

Source First order Lisp.

Target x86 machine code.

Quality Basic, including tail call optimisation.

Properties Functional correctness.

Strength Whole terminating programs. Includes verified garbage collection [30].

Trustworthiness Mechanical proof checking with HOL4.

Vlisp A verified implementation of Scheme [16, 35].

Source Scheme (a higher order Lisp).

Target A custom virtual machine.

Quality Basic.

Properties Functional correctness.

Strength Terminating programs, using a denotational semantics approach.

Trustworthiness Rigorous but informal by-hand proofs.

Compositional Correctness Hur et al.'s work on semantics [19, 18, 2] is directly motivated by the issues raised by verifying compilers for higher order languages in such a way that separately verified code can be linked in without invalidating the correctness result.

CPS and Uncurrying There has been some work on verifying certain transformations that may be of use in a compiler for a higher order functional language. Dargaye and Leroy [8] verified CPS transformations in Coq, and also developed a framework for verifying uncurrying transformations transformations [9].

References

[1] Lennart Augustsson. Compiling pattern matching. In *FPCA*, pages 368–381, 1985.

[2] Nick Benton and Chung-Kil Hur. Biorthogonality, step-indexing and compiler correctness. In Graham Hutton and Andrew P. Tolmach, editors, *ICFP*, pages 97–108. ACM, 2009.

[3] Stefan Berghofer and Martin Strecker. Extracting a formally verified, fully executable compiler from a proof assistant. *Electr. Notes Theor. Comput. Sci.*, 82(2):377–394, 2003.

[4] William R. Bevier, Warren A. Hunt Jr., J. Strother Moore, and William D. Young. An approach to systems verification. *J. Autom. Reasoning*, 5(4):411–428, 1989.

[7] `http://cerco.cs.unibo.it`

[5] Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *J. ACM*, 43(1):166–192, 1996.

[6] Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 54–65. ACM, 2007.

[7] Adam Chlipala. A verified compiler for an impure functional language. In Hermenegildo and Palsberg [17], pages 93–106.

[8] Zaynah Dargaye and Xavier Leroy. Mechanized verification of CPS transformations. In Nachum Dershowitz and Andrei Voronkov, editors, *LPAR*, volume 4790 of *Lecture Notes in Computer Science*, pages 211–225. Springer, 2007.

[9] Zaynah Dargaye and Xavier Leroy. A verified framework for higher-order uncurrying optimizations. *Higher-Order and Symbolic Computation*, 22(3):199–231, 2009.

[10] Maulik A. Dave. Compiler verification: a bibliography. *ACM SIGSOFT Software Engineering Notes*, 28(6):2, 2003.

[11] John Field and Michael Hicks, editors. *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. ACM, 2012.

[12] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In Robert Cartwright, editor, *PLDI*, pages 237–247. ACM, 1993.

[13] Arthur D. Flatau. *A Verified Implementation of an Applicative Language with Dynamic Storage Allocation*. PhD thesis, The University of Texas at Austin, 1993.

[14] Anthony C. J. Fox and Magnus O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In Matt Kaufmann and Lawrence C. Paulson, editors, *ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2010.

[15] Thilo Gaul, Andreas Heberle, Wolf Zimmermann, and Wolfgang Goerigk. Construction of verified software systems with program-checking: An application to compiler back-ends. In Amir Pnueli and Paolo Traverso, editors, *Proceedings of RTRV '99: Workshop on Runtime Result Verification*, Trento, Italy, 1999.

[16] Joshua D. Guttman, John D. Ramsdell, and Vipin Swarup. The Vlisp verified Scheme system. *Lisp and Symbolic Computation*, 8(1-2):33–110, 1995.

[17] Manuel V. Hermenegildo and Jens Palsberg, editors. *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. ACM, 2010.

[18] Chung-Kil Hur and Derek Dreyer. A Kripke logical relation between ML and assembly. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 133–146. ACM, 2011.

[19] Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. The marriage of bisimulations and kripke logical relations. In Field and Hicks [11], pages 59–72.

[20] Joe Hurd. Composable packages for higher order logic theories. In M. Aderhold, S. Autexier, and H. Mantel, editors, *Proceedings of the 6th International Verification Workshop (VERIFY 2010)*, July 2010.

[21] Joe Hurd. *OpenTheory Article Format*, August 2010. Available for download at `http://gilith.com/research/opentheory/article.html`.

[22] Joe Hurd. The OpenTheory standard theory library. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *Third International Symposium on NASA Formal Methods (NFM 2011)*, volume 6617 of *Lecture Notes in Computer Science*, pages 177–191. Springer, April 2011.

[23] Claude Kirchner, Pierre-Etienne Moreau, and Antoine Reilles. Formal validation of pattern matching code. In Serge Autexier, Stephan Merz, Leendert W. N. van der Torre, Reinhard Wilhelm, and Pierre Wolper, editors, *Trustworthy Software*, volume 3 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[24] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[25] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[26] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[27] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, 2009.

[28] Luc Maranget. Compiling pattern matching to good decision trees. In Eijiro Sumii, editor, *ML*, pages 35–46. ACM, 2008.

[29] J. Strother Moore. A grand challenge proposal for formal methods: A verified stack. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *Lecture Notes in Computer Science*, pages 161–172. Springer, 2002.

[30] Magnus O. Myreen. Reusable verification of a copying collector. In Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani, editors, *VSTTE*, volume 6217 of *Lecture Notes in Computer Science*, pages 142–156. Springer, 2010.

[31] Magnus O. Myreen. Verified just-in-time compiler on x86. In Hermenegildo and Palsberg [17], pages 107–118.

[32] Magnus O. Myreen and Jared Davis. A verified runtime for a verified theorem prover. In van Eekelen et al. [44], pages 265–280.

[33] Magnus O. Myreen and Michael J. C. Gordon. Function extraction. *Sci. Comput. Program.*, 77(4):505–517, 2012.

[34] Magnus O. Myreen, Konrad Slind, and Michael J. C. Gordon. Extensible proof-producing compilation. In Oege de Moor and Michael I. Schwartzbach, editors, *CC*, volume 5501 of *Lecture Notes in Computer Science*, pages 2–16. Springer, 2009.

[35] Dino Oliva, John D. Ramsdell, and Mitchell Wand. The Vlisp verified PreScheme compiler. *Lisp and Symbolic Computation*, 8(1-2):111–182, 1995.

[36] Scott Owens, Peter Böhm, Francesco Zappa Nardelli, and Peter Sewell. Lem: A lightweight tool for heavyweight semantics. In van Eekelen et al. [44], pages 363–369.

[37] A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 227–273. Cambridge University Press, 1998.

[38] Andrew M. Pitts. Operational semantics and program equivalence. In Gilles Barthe, Peter Dybjer, Luis Pinto, and João Saraiva, editors, *APPSEM*, volume 2395 of *Lecture Notes in Computer Science*, pages 378–412. Springer, 2000.

[39] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.

[40] Gordon D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977.

[41] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In Bernhard Steffen, editor, *TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998.

[42] Antonis Stampoulis and Zhong Shao. VeriML: typed computation of logical terms inside a language with effects. In Paul Hudak and Stephanie Weirich, editors, *ICFP*, pages 333–344. ACM, 2010.

[43] Antonis Stampoulis and Zhong Shao. Static and user-extensible proof checking. In Field and Hicks [11], pages 273–284.

[44] Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors. *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*, volume 6898 of *Lecture Notes in Computer Science*. Springer, 2011.