Towards a Dependable Optimising Compiler for the Reduceron

Progress Report

Jason S. Reich

14th February 2011

Abstract

A careful developer may use formal methods to ensure that their program code is correct to specifications. However, poorly constructed compilers (and their associated machinery) can produce object code that does not have the same meaning as the source program. Therefore, to ensure the correctness of the executable program, each component of the compilation pipeline needs to be verified.

This report discusses my progress on a particular instance of the problem; compiling a small functional language to be executed on a special-purpose FPGA-based soft-processor, *the Reduceron*. This includes the issue of improving program execution performance through *supercompilation*, a compile-time optimisation. Material that may form part of a thesis is highlighted.

The strategy outlined in the qualifying dissertation (Reich, 2010) is adjusted to account for developments and results obtained in the interim period. From this, a research agenda for the next six months is derived and the overall path to a thesis is discussed.

Keywords: *functional programming, compiler design, compile-time optimisation, supercompilation, formal methods, theorem proving, the Reduceron, dependently-typed programming Agda*

Contents

Part I.

Progress Report

1. Introduction

My research is concerned with the production of a dependable optimising compiler for the Reduceron platform.

The Reduceron (Naylor and Runciman, 2008, 2010) is an FPGA graph-reduction machine for executing programs written in a lazy functional language. This language is close to subsets of Haskell 98 (Peyton Jones et al., 2003) and Clean (van Eekelen and Plasmeijer, 2001).

An existing compiler transforms programs written in this high-level language into an encoding suitable for execution by the Reduceron. While the machine is quite sophisticated in its approach, the current compiler only performs basic, low-level optimisations. Other compilers for functional languages, such as the Glasgow Haskell Compiler (Peyton Jones et al., 1993), use several monolithic transformations to improve program performance when executed on traditional imperative architectures. Furthermore, the compiler's correctness has only been verified through testing.

For the optimisation component of the compiler, I have been investigating supercompilation (Reich et al., 2010), a compile-time optimisation for lazy functional programs. Supercompilation (Bolingbroke and Peyton Jones, 2010; Jonsson and Nordlander, 2008; Mitchell and Runciman, 2008; Sørensen, 1996; Turchin, 1979) is a source-to-source program transformation that executes a program, as far as possible, at compile-time. The resulting (residual) program benefits from improved performance as the supercompiler removes intermediate data structures and specialises higher-order functions.

In my qualifying dissertation (Reich, 2010), I discussed approaches to compiler verification from the literature, expanded on a paper detailing an experimental implementation of a supercompiler for the Reduceron and began to investigate mechanical verification tools. It concluded with a proposal outlining my future research goals and milestones.

This progress report details my activities since the submission of the qualifying dissertation and the impact of these activities in relation to the proposal. Departures from the proposal are highlighted and reflected upon. Similar to the qualifying dissertation, a programme for further research is outlined.

1.1. Summary of Departures from the Proposal

Since my qualifying dissertation, three significant developments have occurred.

- The supercompiler described in the recently published paper by Bolingbroke and Peyton Jones (2010) has lessened the need to investigate simpler designs. *See section 2.4.*
- More focus has been placed on the verification of the non-optimising components of a Reduceron compiler. *See section 2.5.*
- The dependently-typed programming language, Agda, (Norell, 2007, 2009) had almost been dismissed for use at the time of the qualifying dissertation. It has since emerged as the tool of choice. *See section 2.3.*

1.2. Thesis

Several observations motivate my research goals:

- There is a close relationship between operational semantics and the supercompilation process.
- The Reduceron has a minimalist design such that there is a deterministic relationship between its semantic reductions and the hardware clock.
- Dependently-typed programming languages enable proofs about programs expressed within the type system, in a concise and readable but expressive fashion.

For the above reasons my working thesis is;

> *It is feasible to produce a concise, mechanically verified proof of correctness for a Reduceron-targeting supercompiler, formally demonstrating both semantic equivalence between source and target languages, and performance improvement with respect to plain compilation.*

This will be tested through the creation of a verified supercompiler implementation, satisfying the correctness properties. This will be an innovative and unique contribution for a number of reasons:

- There are no mechanically verified supercompilers for practical languages.
- Dependently-typed languages have only recently been used to verify compilers. There are many open questions as to the approaches that should be used.
- While supercompilers have been demonstrated empirically to be improve the performance of some programs, this behaviour has not been formally characterised and verified.

1.3. Structure of this Report

This report consists of three parts.

Part I — Progress Report The main body of the report specifically discusses progress towards my thesis and plans to develop the research further.

Part II — Trip Reports This part contains brief trip reports for two academic events that I attended; Meta 2010 and VSTTE 2010.

Part III — A Comparative Study of Mechanical Theorem Provers Also included is the current form of a technical report, in preparation, contrasting a variety of mechanical theorem proving tools. This report is likely to form part of my thesis.

2. Progress

In this section, I will detail my progress and research activities since the submission of my qualifying dissertation.

2.1. Summary of Planned Tasks

In my qualifying dissertation, I categorised my research into five phases, estimating time requirements. Please refer to the qualifying dissertation for a detailed description of each phase.

- Phase A — Verification Tools. *(3 months.)*
- Phase B — A Supercompiler for the Reduceron. *(4 months.)*
- Phase C — Simplifying and Restructuring the Supercompiler. *(2 months.)*
- Phase D — Verifying the Supercompiler. *(12 months.)*
- Phase E — Classifying Behaviour of the Supercompiler and the Reduceron. *(4 months.)*

From these, I derived objectives for the six month period up until the submission of this progress report.

- Complete Phase A, Verification Tools, by selecting an appropriate logic and toolset for the task, gaining competency in their use and producing some representative proofs within their frameworks. *(2 months.)*
- Complete Phase B, A Supercompiler for the Reduceron, and progress on Phase E, Classifying Behaviour of the Supercompiler and the Reduceron. The outcome should be an empirically dependable program optimiser, improving on the performance results presented in the Meta 2010 (Reich et al., 2010) paper. *(2.5 months.)*
- Make progress on Phase C, Simplifying the Supercompiler, and Phase D, Verifying the Supercompiler. Remove redundancies in the supercompiler design and implementing abstractions to aid verification. Begin to prove properties for some simplified supercompilation components. *(1.5 months.)*

2.2. Conference Attendance

I attended two academic events since the submission of the progress report, to develop the knowledge, skills and contacts required to complete my research.

2.2.1. Meta 2010

The *Second International Workshop on Metacomputation in Russia* (Meta 2010) is a gathering of researchers working on in the areas of program analysis and program manipulation based on metacomputation; in particular, supercompilation." [1] I presented a paper on *Supercompilation and the Reduceron* (Reich et al., 2010) and participated in discussions on related topics. A trip report is included in chapter 4.

One of the recent influences on my research, the Bolingbroke and Peyton Jones (2010) style supercompiler, was introduced to me through Simon Peyton Jones's invited talk. While it currently has unsolved issues, such as the creation of residual code that is exponentially larger than the original program, it's close relationship with the language's operational semantics will likely make it an easier target for verification than the design I presented at Meta 2010. (Reich et al., 2010)

Closing the workshop, Andrei V. Klimov of the Keldysh Institute of Applied Mathematics remarked that the mechanised verification of a supercompiler for a usable language was the next major milestone, following the Coq implementation for lambda calculus by Krustev (2010).

2.2.2. VSTTE 2010

The *3rd International Conference on Verified Software: Theories, Tools and Experiments* (VSTTE 2010) aims to "advance the state of the art in the science and technology of software verification through the interaction of theory development, tool evolution, and experimental validation." [2] I presented a poster on the verification of a supercompiler to invite discussion of the topic with those experienced in formal verification.

However, this year's participants tended to be more interested in automated theorem proving tools, lightweight verification and the verification of programs written in imperative languages. The general message from those with the relevant experience was that this seemed to be reasonable problem to be solved using mechanised verification and that making contacts with experienced users is essential for learning to use any tool.

[1] `http://meta2010.pereslavl.ru/`
[2] `http://www.macs.hw.ac.uk/vstte10/Call_for_Papers.html`

VSTTE 2010 also included a verification competition where participants were asked to produce verified implementations of five specifications. While I could not participate at the time due to a technical issue, I later used these problems to further develop my skills in Agda. A trip report is included in chapter 5.

2.3. Phase A — Verification Tools

Phase A is intended to guide the selection of a logic and theorem proving environment to complete the research.

Conferences and technical report Despite my intentions, discussions at Meta 2010 and VSTTE 2010 did not assist me in deciding which mechanical verification tool to use. As a result, I have prepared a technical report comparing several mechanised theorem provers, both in terms of their headline characteristics and my experiences of using them to formalise the McCarthy and Painter (1967) compiler correctness problem.

The report, included in Part III, discusses Agda, Coq, HOL Light and Isabelle/HOL. An existing HOL Light proof of McCarthy-Painter was used in the study and Brian Huffman at Portland State University guided aided the development of an Isabelle/HOL proof. The Agda and Coq proofs could probably benefit from similar refinement by an expert user.

Agda While lacking some of the tactical niceties of the other tools, Agda (Norell, 2007, 2009) has a focus as a programming language, exposes its underlying logic, has syntactic and semantic similarities to Haskell and compiles to Haskell code. It is for these reasons that Agda has been selected as the verification tool.

Furthermore, Agda and the use of dependently-typed languages are relatively recent developments in the field of formal verification. Leroy (2009) used Coq in the development of the Compcert C verified compiler but Agda has not been used in the verification of a practical compiler. This should make it an interesting approach in the context of a thesis.

Summary According to the proposal in the qualifying dissertation, Phase A should have lasted a further two months. It has slightly overrun due to the preparation of the technical report. However, I have familiarised myself with the practicalities of using a variety of theorem provers and selected Agda as the best suited to the project's needs.

2.4. Phase B & C — A Supercompiler for the Reduceron

Phase B was due to produce a supercompiler that operates on F-like programs and yields significant speed-ups on the Reduceron while Phase C was to simplify it for verification purposes.

Publications Bolingbroke and Peyton Jones (2010) have published a new approach to supercompilation. This development has made me reassess how to structure a supercompiler for mechanical verification. The advantage of Bolingbroke's approach is that where my Meta 2010 supercompilation design performs transformation to simulate program execution, Bolingbroke actually evaluates the expression according to operational semantics and then retrieves the residual program from semantic states. This close relationship with the operational semantics should make it easier to confirm that semantic equivalence is maintained.

Similarly, Mitchell (2010) published a new approach to ensuring the termination of the supercompiler process. It is unclear whether this technique will be easier to represent to Agda's termination checker than homeomorphic embedding.

Summary Due to the emergence of this design, I have decided against improving the Meta 2010 supercompiler implementation (Phase B) and simplifying it for verification (Phase C). Instead, I aim to utilise Bolingbroke and Peyton Jones's approach, This is discussed further in subsection 3.1.2 of the plan.

2.5. Phase D — Verification of a Compiler

Phase D was originally envisaged to only verify the optimising component of the F-lite to Reduceron core compiler, the supercompiler. It became apparent that to perform tasks such as the classification of supercompiler behaviour, a full model of F-lite to Reduceron core compilation is required. Therefore, Phase D has been enlarged to encompass the verification of the entire compilation.

Semantics To that end, I have produced Agda implementations of F-lite source-level and Reduceron core operational semantics. These implementations are derived from relational specifications of the semantics, also represented in Agda in a style inspired by Danielsson (2010).

It is hoped that proofs on these relational specification will be more concise than those performed directly on the functional implementation. However, as the functional implementation is extracted from the specification, the proved properties should map.

Bisimulation Variants of bisimulation are under investigation for the correctness condition. It seemed an appropriate choice as it fits well with operational semantics. The issue lies with weak bisimulation being required as it may take an arbitrary number of steps to reach an equivalent state. Transitivity over the correctness condition is required so that the proof can be decomposed. However, weak bisimulation is not implicitly transitive and requires additional assumptions.

Compiler Initially, it seemed that the compiler would be complex enough to require the use of datatype-generics, a technique commonly used in Haskell compilers but little investigated in languages that require totality. I began to investigate representing generics techniques in Agda, such as generalised catamorphisms and Uniplate. (Mitchell and Runciman, 2007) Results, using explicitly sized-types, were promising but a simplified compiler structure was developed that, for now, has alleviates the need for boilerplate removal.

Summary Progress has been made on representing the semantic equivalence property and encoding the semantics and compiler components. This phase is being completed roughly as planned in the qualifying dissertation.

2.6. Phase E — Classifying Behaviour of the Supercompiler and the Reduceron.

In Phase E, I intend to map out the program classes that benefit or suffer from this form of supercompilation.

Motivation It is rare that proofs of correctness for optimising compilers actually demonstrate that the resulting programs are guaranteed to perform better (or, at least no worse) than the original. The deterministic, predictable behaviour of the Reduceron architecture provides an opportunity to determine this property.

Approach If the relationship between F-lite operational semantics and Reduceron execution can be established, then it should be possible to reason about a Bolingbroke style supercompiler's effect on program performance. As the supercompiler is performing reduction at compile-time instead of execution-time, the reduction steps performed by the supercompiler should predictably remove execution clock cycles.

The point where additional computation may be added to the program is where calls to residual functions are introduced. The relationship between program constructs

and residual function applications needs to be determined to complete the model of a supercompiler's effect on program execution.

Summary These relationships will be best determined once the formal proof linking the source language, supercompilation and compilation to Reduceron core is complete. As such, Phase E has been delayed. It should, however, form a distinctive contribution in the thesis.

2.7. Progress Summary

Overall, the path laid out in the qualifying dissertation has been followed. It is the time allocated to each phase that appears to have needed adjustment. Slightly less time was required for simplifying the supercompiler due to the work by Bolingbroke and Peyton Jones (2010). Slightly more time was required to decide on a proof assistant. Overall, progress is being made towards the research goals.

3. Plans

In this section, I will restructure the proposal from the qualifying dissertation and describe my immediate research agenda in light of the results from the last six months.

3.1. Strategy

Two alterations have been made to the strategy described in the qualifying dissertation to account for the results of the interim period.

3.1.1. Phase D — Verifying the Compiler

As discussed in section 2.5, the remit of Phase D has been enlarged to include the entire compiler chain, including optimising component. This is to enable a better understanding of how the supercompiler affects program performance.

The proof should decompose as follows:

- Caseless, let-lifted source programs are simulated by compiled code.
- Caseless source programs are simulated by those where let-bindings are all lifted to the root of each function body, a caseless-to-let-lifted-source transformation.
- Source programs are simulated by source programs following Scott encoding, a source-to-caseless-source transformation.
- Source programs are simulated by supercompiled programs, where supercompilation is a source-to-source transformation.

The supercompilation proof will demonstrate that driving and tying (Reich et al., 2010) preserve simulation; thus, the entire process preserves simulation. An unresolved issue is how to demonstrate to the Agda termination checker that supercompilation must terminate.

3.1.2. Phase F — A Verifiable Supercompiler for the Reduceron

The remaining work of Phase B, A Supercompiler for the Reduceron, and Phase C, Simplifying and Restructuring the Supercompiler, will be merged into a single phase. This new phase will see the development of a supercompiler similar to that described by Bolingbroke and Peyton Jones (2010) as it appears it will be more receptive to formal reasoning.

However, this supercompiler will be built through the principle of correctness by construction, a verification pattern enabled by dependently-typed languages, such as Agda. This pattern will require the specifications to be properly formalised before the implementation. Therefore, Phase D and Phase F now depend on each other.

3.2. Time Allocation

- Phase A — Verification Tools. *(4 months total. 0.5 months remaining.)*
- Phase B — A Supercompiler for the Reduceron, Meta 2010 version. *(4 months total. Completed.)*
- Phase C — Simplifying and Restructuring the Supercompiler. *(Now part of Phase F.)*
- Phase D — Verifying the Compiler. *(10 months. 9 months remaining.)*
- Phase E — Classifying Behaviour of the Supercompiler and the Reduceron. *(4 months total. 3.5 months remaining.)*
- Phase F — A Verifiable Supercompiler for the Reduceron. *(5 months. 4 months remaining.)*

3.3. Upcoming Research Visits

I have upcoming research visits to the University of Nottingham (2nd to 3rd February) and Chalmers University of Technology (21st to 25th February). Both are home to groups of Agda users with considerable experience. The objective of each visit is to improve my knowledge of Agda and invite advice from experienced users of the tool on how to approach this particular verification problem. To facilitate this, I have been invited to give a seminar on my progress so far to both groups.

These visits should contribute to the completion of the technical report for Phase A and the skills required for Phases D and F.

3.4. Immediate Plan

Over the next six months, I intend to;

- Complete Phase A, Verification Tools, using my upcoming visits to the University of Nottingham and Chalmers University of Technology to complete the technical report comparing mechanical theorem provers. *(0.5 months.)*
- Progress on Phase D, Verifying the Compiler, by completing the proof of the non-optimising components of the F-lite to Reduceron core compiler. This proof is intended to be submitted for publication at Trends in Functional Programming 2011. *(3 months.)*
- Progress on Phase F, A Verifiable Supercompiler for the Reduceron, specifying and producing a working prototype of a Bolingbroke and Peyton Jones (2010) style compiler for the Reduceron. Proofs of semantic equivalence and termination will likely be incomplete but largely outlined. *(2.5 months.)*

3.5. Path to the Thesis

I envisage a Ph.D. thesis consisting of;

- An introduction to the Reduceron, its compiler and supercompilation. *Based upon the extended Meta 2010 paper included as part of the qualifying dissertation (Reich, 2010) using results from Phase F.*
- A review of verification methods for proving compilers correct. *An extended form of that submitted in the qualifying dissertation.*
- A comparison of interactive theorem provers that may be used to mechanise a proof of correctness. *Based on the report included in Part III.*
- The formalised proof of semantic preservation for the plain Reduceron compiler, excluding optimisation component. *To be completed as part of Phase D and based on the paper to be submitted to TFP 2011.*
- A formal analysis of the optimal qualities of the supercompiler applied to the Reduceron architecture. *To be completed as part of Phase E.*
- The formalised proof of semantic preservation for the full Reduceron compiler, extending the previous proof. *To be completed as part of Phase D.*

The contributions will be:

- A mechanically verified supercompiler for the Reduceron architecture, confirming both the semantic equivalence property *and* the performance improvement property.

- A more general methodology for solving compiler correctness issues using a dependently-typed programming language.
- A framework for reasoning about the performance characteristics with respect to compiler optimisations.

Part II.

Trip Reports

4. Meta 2010 — *1st to 5th July 2010*

4.1. Introduction

> *"The Second Metacomputation Workshop aims to bring together researchers working in the areas of program analysis and program manipulation based on metacomputation, in particular, supercompilation, distillation, mixed computation, generalized partial computation and partial evaluation."* [1]

The workshop was held in the Russian city of Pereslavl-Zalessky, at the Program Systems Institute of the Russian Academy of Sciences. I attended the Meta 2010 workshop to improve my knowledge of supercompilation, through discussions with the prominent researchers of the topic and to present a paper on *"Supercompilation and the Reduceron."* (Reich et al., 2010)

4.2. Selected Attendees

Gavin Mendel-Gleason and Geoff Hamilton At Dublin City University. Geoff works on a variant of supercompilation which he calls *distillation* and is supervising Gavin who is investigating the verification of a supercompiler for System F.

Peter Jonsson and Johan Norlander At Luleå University of Technology. Johan is supervising Peter's PhD on supercompiler strict languages and Haskell.

Dimitur Krustev At IGE-XAO Balkan Ltd. Has been proving a supercompiler for a small lambda calculus as a hobby.

Andrei Klimov Heads the Sector of Program Analysis and Transformation at the Keldysh Institute of Applied Mathematics, Russian Academy of Sciences. One of Valentin Turchin's students, he has formed a company investigating the supercompilation of major imperative languages, such as Java.

[1] `http://meta2010.pereslavl.ru/`

Ilya Klyuchinikov and Sergei Romanenko At the Keldysh Institute of Applied Mathematics, The Russian Academy of Sciences. Ilya, under the supervision of Sergei, is investigating the implementation of a supercompiler for a small functional language and its use *for* proving theorems.

Simon Peyton Jones At Microsoft Research Cambridge. Supervises Max Bolingbroke at Cambridge University who is investigating the implementation of a supercompiler for the Glasgow Haskell Compiler.

4.3. Points of Discussion

Simon's talk In Simon's invited talk, he outlined a supercompiler implementation, developed with Max, to be published at Haskell Symposium. In their design, driving is performed by evaluating the expression under the operational semantics and extracting equivalent expressions from intermediate states. Termination is abstracted so a number of methods can be experimented with. Memoisation is used to reuse residual expressions from equivalent derivations. My instinct is that its direct use of operational semantics and well-known functional programming idioms might make it a good choice for verification.

Termination with 'couple' Geoff highlighted that `couple` is well-quasi-ordered in itself. Using `couple` as the termination condition would save Neil's generalisation case even arising!

Generalisation Simon is not convinced by the need to generalise residual expressions but my instinct is his lack of generalisation is what leads to the code explosion.

Tie Various discussions with Geoff, Peter, Ilya and Simon on whether to fold on renamings or instances of expressions. My approach currently folds on all instances of expression but I can see now that this would prematurely terminate supercompilation, leaving some terms unoptimised.

Other uses of supercompilation Ilya has been proving monadic laws on monad instances with his supercompiler. Interesting work but seems to require a lot of extra machinery on top of it to ensure equivalent terms are reached.

Proving a supercompiler Andrei remarked that the mechanised verification of a supercompiler for a usable language was the next major milestone, following the Coq implementation for lambda calculus by Dimitur.

Other items

- People seem to be standardising on NoFib imaginary.
- Gavin has been trying to use Coq to prove a supercompiler.
- Dimitur indicated that his was a pretty tough exercise.
- Peter talked about OTT (`http://www.cl.cam.ac.uk/~pes20/ott/`) for extracting semantics for theorems.
- Sergei and Andrei were very interested in the Reduceron as PSI @ RAS in Pereslavl have just taken delivery of first 200 nodes in a FPGA-based supercomputer.

5. VSTTE 2010 — *16th to 19th August 2010*

5.1. Introduction

> *"The Third International Conference on Verified Software: Theories, Tools, and Experiments follows a successful inaugural working conference at Zurich (2005) and a successful conference in Toronto (2008). This conference is part of the Verified Software Initiative (VSI), a fifteen-year, cooperative, international project directed at the scientific challenges of large-scale software verification. VSI also includes UKCRC's Grand Challenge 6, i.e. Dependable S ystems Evolution. VSTTE is open to anyone who is interested in participating actively in the VSI effort."* [1]

I attended VSTTE 2010, held at Heriot-Watt University in Edinburgh, to present a poster on the supercompilation process and my thoughts on verifying it. The aim of this visit was to meet people actively using mechanical theorem provers and getting their impressions of using it on this problem instance.

5.2. Talks

Conference focus

Most talks in the main conference and satellite workshops were focused on the use of automated theorem provers and model checking. Despite being generally interesting topics in the field of formal verification, few related to the questions to which I was seeking answers.

The L4.verified project

Gerwin Klein gave an invited talk on the L4.verified microkernel project. Their approach uses Isabelle/HOL (Nipkow et al., 2010) to model the specifications of their kernel,

[1] `http://www.macs.hw.ac.uk/vstte10/Call_for_Papers.html`

prove the correctness of a design and low-level C implementation. The design is automatically extracted during the course of development into a Haskell prototype to test the design actually works.

It is very interesting to see Isabelle/HOL and its code extractor being used in projects of this scale. However, Klein does make brief mention of moving to Coq in the future.

5.3. Poster Reception

A few participants visited my poster. Many felt it was an interesting and achievable goal. However, due to the research interests of the participants, there were no specific discussions on tool capabilities. The main advice was to ensure that I had contact with someone with experience of the tool I wanted to use.

5.4. Verification Competition

This year, a competition was organised for the participants. Teams of up to three people could use any tool they wished to verify implementations of five specifications. A two hour window was given to perform the tasks and the results were collected and analysed for trends.

Most participants used automated tools such as Boogie, Dafney and KeY but there were a few Isabelle and ProofPower based entries.

I was intending to enter using either Agda, Coq or Isabelle/HOL as my tool but unfortunately broke my netbook on the first day of the conference. C'est la vie. I did later use the competition problems to improve my skills in Agda.

Part III.

A Comparative Study of Mechanical Theorem Provers

6. Motivation and Scope

This chapter discusses some the arguments in favour of mechanising theorem proofs. Several theorem proving tools are introduced and their features compared. A single compiler verification instance introduced to be used as common, practical basis for comparison.

6.1. Verification

Software properties can be verified through formal reasoning and proofs, much in the same way a logician formally infers new theorems based on known assumptions. These proofs can be constructed 'by hand,' relying on the skill and care of the developer. However, the size of modern programs has increased the likelihood and severity of human error. This has led to the development of *proof assistants*, tools that ensure the soundness of inference steps and perform some of the more menial aspects of the proof process.

Two classes of theorem proving tools have emerged: mechanised (or interactive) and automated. Mechanised proof assistants require a human to 'drive' the proof process but will perform the necessary rewriting and check the proof is sound. Automated theorem provers attempt to prove program properties algorithmically. However, they generally do this for logics of limited expressivity to ensure decidability or semi-decidability.

In this report, I investigate a number of the popular mechanised proof assistants used in the verification of functional programs. The tools are compared both on their design philosophy and on their practical use through the development of a correctness proof of a well-known result.

6.2. Tools Under Comparison

I have chosen four tools that have appeared frequently in the literature describing the verification of properties on programs written in functional languages. They only represent a small sample of the tools available, but it seemed appropriate to prioritise those which have been used for similar tasks.

HOL Light HOL Light (Harrison, 2010) is an interactive theorem prover and a member of the LCF/HOL family. It is implemented in Objective Caml. The aim is to "[give] the system a simple and uncluttered feel" through a simpler logical core and [by utilising] little legacy code," (Harrison, 2010) compared with its sibling systems.

The HOL Light tutorial document (Harrison, 2007) is extensive and written very clearly. The library itself is fully documented but not always as articulate. I have, for example, been finding it difficult to ascertain the difference between `REWRITE_TAC` and `SIMP_TAC` beyond that `SIMP_TAC` is "more powerful ... [and] will exploit contextual information."

Isabelle/HOL Isabelle is "a generic proof assistant" (Nipkow et al., 2010) written in ML. The Isabelle system provides the Isar language for directing the proof assistant. Isar aims "for proof text naturally understandable for both humans and computers." (Nipkow et al., 2010) However, a number of logics can be embedded within the Isar framework. The most popular logic instance for the system is Isabelle/HOL, another member of the HOL family.

Isabelle/HOL is extensively used for software verification and has quite an active community. There is a lot of documentation, tutorial and reference material exist but is difficult to know which source is best for the information required.

An interesting technology available to Isabelle/HOL users is Haskabelle, "a tool to translate programs written in Haskell into Isabelle specifications." (Haftmann, 2010) The tool works on a subset of Haskell 98 programs and is targeted at proving partial correctness. Isabelle also includes a code extraction feature, by which Haskell and SML programs can be generated from Isabelle proofs.

Coq An alternative logical framework to HOL is the Calculus of Inductive Constructions. (Coquand and Huet, 1988) Propositions are represented as types (which are members the sort 'Prop'). If a term of the appropriate type can be constructed, the proposition is true with the term as its proof. Coq is a theorem prover based on this deductive system and is implemented in O'Caml.

The Coq community has a similar level of documentation, tutorial and reference material to the Isabelle/HOL community. The `#coq` channel on Freenode is fairly active and quite helpful to new users.

Like Isabelle/HOL, Coq can also extract executable code from theorems. A number of target functional languages are supported, currently Objective Caml, Haskell and Scheme. However, there is no Coq equivalent to Haskabelle.

Agda Agda (Norell, 2007, 2009) is dependently-typed programming language, based on Martin-Löf's (1971) type theory. It differs from the other three mechanised proof

assistants in several ways. It is written in Haskell rather than O'Caml. This, possibly, has led to the Agda language being similar more similar in style to Haskell, in contrast with the ML-styled HOL Light and Isabelle.

Another key difference is that it focuses on being a dependently-typed programming language, in which you happen to be able to express proofs, as opposed to Coq, a theorem prover in which you can do dependently-typed programming.

6.3. A Verification Problem

The four tools each have their own focusses and philosophies. To compare them, a single verification instance has been selected to allow direct comparison of their features in the context of a practical problem.

In their seminal paper, McCarthy and Painter (1967) produce a hand proof of correctness for a compiler that translates a simple source arithmetic language to instructions for a register machine.

The source language only contains natural numbers, variables bound to natural numbers and the binary addition operator. The target language consists of four instructions:

- `LI n` — Load immediate value into the accumulator.
- `LOAD r` — Load the value of register r into the accumulator.
- `STO r` — Store the value of the accumulator in register r.
- `ADD r` — Add the value of register r to the accumulator.

McCarthy and Painter encode the semantics of the two languages into a *first-order predicate logic*. States for the source semantics (and the target semantics) are defined as predicates over variables (or registers) containing natural number values.

A compiler is also defined in terms of first-order predicate logic and, as an argument, takes a function that maps variables bound in the source state to registers initialised in the target state.

A relation of *partial equality* between states is defined. For states σ_1 and σ_2 and a set, A, of variable names, the relation $\sigma_1 =_A \sigma_2$ holds if $\forall x \in A \cdot \sigma_1(x) = \sigma_2(x)$. Using this condition, McCarthy and Painter construct a theorem for compiler correctness, which is proved using induction over the source language and the laws that they had previously defined. A Haskell encoding of this proof is included in Appendix A.

The authors point out that it is trivial to extend the proof to handle a source language that contains multiplication. However, constructs such as sequential composition, conditionals and jump statements would require *"a complete revision of the formalism."* (McCarthy and Painter, 1967)

By modern standards, the proof appears overly verbose. It exposes a large amount of the theory that is taken for granted in a modern logic over computer languages. Although the source language may not contain the features required for it to be considered 'useful' in a software engineering context, the McCarthy and Painter (1967) example posed a vital question and lay the foundations for a field.

6.4. Summary

In this section, mechanical theorem proving has been motivated and four tools have been introduced: Agda, Coq, HOL Light and Isabelle/HOL. These shall be compared through the formalisation of the McCarthy and Painter (1967) compiler in each.

7. HOL Light

In this chapter, the practicalities of using HOL Light are discussed, especially with regard to proving the McCarthy and Painter (1967) problem.

Proof of McCarthy-Painter A mechanised proof of the problem outlined in section 6.3 is actually included in the standard HOL Light distribution. Written by John Harrison, the developer of HOL Light, it is based on an earlier proof produced for the HOL-4 system by Robert Bauer and Ray Toal. It is reproduced, with Harrison's original comments in Appendix B and will be used to illustrate the characteristics of theorem proving with HOL Light.

Interface In keeping with the lightweight philosophy of the system, there are few comforts for the user. Proofs are performed in the O'Caml interactive interpreter. The system is initialised by interpreting the entire HOL Light library. This process takes a number of minutes. There is no mechanism by which one can remove introduced definitions. Therefore, initialisation has to be repeated whenever a valid, but incorrect, HOL term is defined or a library file has been edited.

Proof scripts Proof scripts are loaded on top of the HOL Light library. HOL Light utilises two languages. The host, O'Caml, is used to instruct the theorem prover and another language, embedded within it, represents the proof terms.

The style of a HOL Light proof is to define proved lemmas and laws and bind them to O'Caml variables. These are then, in turn, used as arguments to tactics which prove other lemmas and theorems. These tactics are expressed as O'Caml functions.

For example, in lines 36 to 39, a HOL Light term data structure is defined, resulting in laws about its inductive and recursive properties. The `exp_INDUCT` laws are used in line 195 to prove the correctness theorem. Similarly, line 84 defines a function for updating memory locations and binds its laws to `update_def`.

Additional lemmas Additional lemmas, that I had not considered in my Haskell encoding (Appendix A), are required. The distributivity `S'` over append and the effects of operation on memory cannot be assumed without further proof.

Theorem Lines 174 to 179 define the correctness theorem that is then proved used in the strategy in lines 193 to 198. This strategy first uses induction over the source expression, then applies rewrite rules over various forms.

The author admits that the application of `GSYM SKOLEM_THM` and `EXISTS_REFL` is a "hack." Without the author's expert knowledge of the system, I am unsure that I could have produced a similar, working solution.

Documentation This is a common problem as it is quite opaque what the tactics and rules are actually doing. The library itself is documented but not always as articulate. For example, it is difficult to ascertain the difference between `REWRITE_TAC` and `SIMP_TAC` beyond that `SIMP_TAC` is "more powerful ... [and] will exploit contextual information."

Code extraction Unfortunately, there is no method to extract a working implementation from the theorem prover. Instead, an equivalent program in a working programming language would need to be developed, possibly breaking the properties proved upon it.

8. Isabelle/HOL

The development of an Isabelle/HOL formalisation of McCarthy and Painter (1967) is outlined as a literate program. Brian Huffman at Portland State University guided the creation of this proof and has my thanks.

Interface The main method for interacting with Isabelle is through the generic Proof General interface. Proof General provides contextual highlighting of proofs, stepping through the loading (and unloading) of definitions and the visualisation current goal states.

Proof scripts In contrast with HOL Light, in Isabelle (Nipkow et al., 2010) there is no distinction as such between the term language and tactic language. To begin, we define the datatypes representing source and target syntax (`exp` and `list inst` respectively), functions representing semantics (`E` and `S`) and `C`, the compilation function.

```
theory McCarthy
imports HOL List
begin

types string = "char list"

(* Source arithmetic expressions *)
datatype exp = Lit nat
             | Var string
             | Plus exp exp

(* Source semantics *)
primrec E :: "exp => (string => nat) => nat" where
    "E(Lit n)      s = n"
  | "E(Var v)      s = s v"
  | "E(Plus e1 e2) s = E e1 s + E e2 s"

(* Target machine memory addresses *)
datatype cell = Acc
              | Reg nat

(* Target machine instructions *)
datatype inst = Li nat
              | Load nat
```

```
              | Sto nat
              | Add nat

(* Target memory storage operation *)
definition update :: "cell => nat => (cell => nat) => (cell => nat)" where
  "update x z s y = (if y = x then z else s y)"

(* Target instruction semantics *)
primrec S :: "inst => (cell => nat) => (cell => nat)" where
    "S (Li n)   s = update Acc n s"
  | "S (Load r) s = update Acc (s (Reg r)) s"
  | "S (Sto r)  s = update (Reg r) (s Acc) s"
  | "S (Add r)  s = update Acc (s (Reg r) + s Acc) s"

(* Target semantics for lists of instructions *)
primrec S' :: "inst list => (cell => nat) => (cell => nat)" where
    "S' []          s = s"
  | "S' (inst#rest) s = S' rest (S inst s)"

(* Source to Target compilation *)
primrec C :: "exp => (string => nat) => nat => inst list" where
    "C (Lit n)      m r = [Li n]"
  | "C (Var v)      m r = [Load (m v)]"
  | "C (Plus e1 e2) m r = C e1 m r
                        @ [Sto r]
                        @ C e2 m (r + 1)
                        @ [Add r]"
```

Lemmas Like the HOL Light proof, lemmas are required to demonstrate that `S'` distributes over `@` append and immediately reading an `updated` variable gives the value but immediately reading a lower variable bypasses the `update`.

The first lemma, `s'_append` is proved by introducing all the universally quantified variables, applying induction over the left argument of the append function and simplifying the results.

```
lemma s'_append: "∀p1 p2 s. S' (p1 @ p2) s = S' p2 (S' p1 s)"
apply (rule allI)
apply (induct_tac p1)
apply auto
done
```

The `update_` lemmas are proved using Isabelle's automatic simplifier, ensuring the use of the definition of `update`.

```
lemma update_different: "∀x y z s. ~(x = y) --> (update x z s y = s y)"
apply (auto simp add: update_def)
done

lemma update_same: "∀x z s. update x z s x = z"
apply (auto simp add: update_def)
done

lemma update_below: "∀q r z s. q < r⟶update (Reg r) z s (Reg q) = s (Reg q)"
apply (auto simp add: update_def)
done
```

Proof development and outline Developing the proof of the main theorem, more 'manual' tactics were used to explore the problem and determine the lemmas (above) that would be required.

The proof introduces variables for the universal quantifications and performs induction over source expressions. Isabelle's `simp_all` tactic was able to discharge the first two (base) cases with a little guidance.

For the third, inductive, case, the `impI` rule is used to introduce the antecedents. Then `drule_tac ... in spec` and `back` are used to instantiate variables manually in the inductive assumptions and antecedents. Isabelle'ssimplifier was then able to finish the proof. As part of this process, the required form of the `update_below` lemma became apparent.

```
(*
  Assumptions;
  * For all variables, the register it is mapped to is less than r.
  * For all variables, the value of the variable in source state "s" is
    equal to that of the mapped register in target state "s'".

  Theorem;
  * The evaluation of the compiled form of "e", in mapping "m", starting
    at register "r" is equal to that of the evaluation of "e".
  and
  * All registers below "r" are unchanged after the evaluation of "e".
*)
theorem Correctness:
  "∀e m s s' r.
     (∀v. m v < r)⟶
     (∀v. s v = s' (Reg (m v)))⟶
         (S' (C e m r) s' Acc = E e s) ∧
         (∀x < r. (S' (C e m r) s' (Reg x) = s' (Reg x)))"
apply (rule allI)
apply (induct_tac e)
apply (simp_all add: update_def)
```

```
apply (simp_all add: update_same update_different s'_append)
apply (intro allI impI)
apply (drule_tac x="m" in spec) back
apply (drule_tac x="s" in spec) back
apply (drule_tac x="(update (Reg r)
    (E exp1 (λa. s' (Reg (m a)))) (S' (C exp1 m r) s'))" in spec)
apply (drule_tac x="Suc r" in spec)
apply (drule mp)
apply (intro allI impI conjI)
apply (drule_tac x="v" in spec)
apply (simp only:)
apply (drule mp)
apply (intro allI impI conjI)
apply (drule_tac x="m" in spec)
apply (drule_tac x="s" in spec)
apply (drule_tac x="s'" in spec)
apply (drule_tac x="r" in spec)
apply (drule mp)
apply (simp)
apply (drule mp)
apply (simp)
apply (drule_tac x="v" in spec)
apply (drule_tac x="v" in spec)
apply (simp add: update_different)
apply (simp add: update_same update_different update_below)
done
```

Refinement This proof was then simplified such that Isabelle's simplifiers could handle the majority of it automatically. An interesting feature of the Isabelle tools was that it was able to tell me that a proof of the `Nat.less_SucI` lemma already existed when I attempted to define it myself.

```
theorem CorrectnessAgain:
  "∀e m s s' r.
     (∀v. m v < r) ⟶
     (∀v. s v = s' (Reg (m v))) ⟶
         (S' (C e m r) s' Acc = E e s) ∧
         (∀x < r. (S' (C e m r) s' (Reg x) = s' (Reg x)))"
apply (rule allI)
apply (induct_tac e)
apply (simp_all add: update_def)
apply (simp add: update_same update_different update_below Nat.less_SucI s'_append)
done
```

Documentation There is excellent documentation for Isabelle/HOL's standard laws and lemmas. These takes the form of typeset literate proof scripts. However, the large

number of Isabelle tutorials, covering a wide variety of incompatible versions, make it difficult to get to grips with the core commands and idioms.

Code extraction The Isabelle to Haskell code extractor is very easy to use and appears to produce code similar to that which would have been written by a human programmer. The commands in the documentation were slightly incorrect, presumably due to describing a previous version of Isabelle.

9. Coq

In this chapter, a literate script of McCarthy and Painter (1967) is presented in Coq. It explains the proof and my experience of the tool in producing it.

Interfaces Coq provides a number of user interfaces including the Proof General used by Isabelle. The included GUI, CoqIDE, is highly unstable at the present time. Proof General, on the other hand, provides nearly all of the functionality exposed by CoqIDE and rarely has issues.

Proof scripts Like Isabelle, the term and tactic languages are one and the same. Coq's language is quite dissimilar to popular functional languages such as ML and Haskell.

The syntax and semantics of the source and target languages are defined as inductive datatypes and fixpoint functions respectively. Unlike Haskell and Isabelle, Coq's notation requires an explicit pattern matching operation `match`.

```
Require Import Arith Bool BoolEq List Omega.

(* Source Language *)

Inductive source : Set :=
  | Con : nat -> source
  | Var : nat -> source
  | Add : source -> source -> source.

Fixpoint eval e s := match e with
  | Con n   => n
  | Var v   => s v
  | Add x y => eval x s + eval y s
end.

(* Target Language *)
Inductive target : Set :=
  | Li    : nat -> target
  | Load  : nat -> target
  | Store : nat -> target
  | Sum   : nat -> target.

Inductive cell : Set :=
  | Acc : cell
```

```
| Reg : nat -> cell.
```

Constructivism As Coq is a constructive logic, one cannot match on propositional `False` as it is the empty type, something that cannot be constructed. Therefore, booleans are required to test decidable properties. This results in some conversion between propositional and boolean forms.

```
Definition cell_beq (x : cell) (y : cell) : bool := match x,y with
  | Acc, Acc     => true
  | Acc, Reg n   => false
  | Reg n, Acc   => false
  | Reg n, Reg m => beq_nat n m
end.

Lemma cell_beq_refl : forall x, cell_beq x x = true.
intros. induction x. auto. induction n. auto. auto.
Qed.

Lemma injective_reg : forall q r : nat, Reg q = Reg r -> q = r.
intros. simplify_eq H. auto.
Qed.

Lemma injective_reg2 : forall q r : nat, q = r -> Reg q = Reg r.
auto.
Qed.

Lemma eqS : forall x y : nat, x = y -> S x = S y.
auto.
Qed.

Lemma cell_beq_antirefl : forall x y,  x <> y -> cell_beq x y = false.
double induction x y.
intro. elim H. auto.
intros. auto.
intros. auto.
double induction n n0.
intro. elim H. auto.
intros. auto.
intros. auto.
intros. simpl. apply H0.
injection. elim H1.
apply injective_reg2.
apply eqS.
apply injective_reg.
auto.
Qed.

Definition update (x : cell) (y : nat) (f : cell -> nat) (z : cell) :=
  if cell_beq z x then y else f z.
```

```
Definition exec e s := match e with
  | Li n    => update Acc n s
  | Load r  => update Acc (s (Reg r)) s
  | Store r => update (Reg r) (s Acc) s
  | Sum r   => update Acc (s Acc + s (Reg r)) s
end.

Fixpoint exec' es s := match es with
  | cons e t => exec' t (exec e s)
  | nil      => s
end.

(* Compiler *)
Fixpoint compile (e : source) (m : nat -> nat) (r : nat) : list target
  := match e with
   | Con n   => Li n :: nil
   | Var v   => Load (m v) :: nil
   | Add x y => compile x m r
             ++ Store r
             :: compile y m (S r)
             ++ Sum r
             :: nil
end.
```

Lemmas Lemmas similar to those in the HOL Light and Isabelle/HOL proofs are required. `exec'_append` demonstrates that `exec'` distributes over `append`.

```
Lemma exec'_append : forall p1 p2 s, exec' (p1 ++ p2) s = exec' p2 (exec' p1 s).
intro p1. induction p1. auto. simpl. intros p2 s. rewrite IHp1. reflexivity.
Qed.
```

The `update_` lemmas prove that immediately reading an `updated` variable gives the value but immediately reading a different, or specifically, lower variable bypasses the `update`.

```
Lemma update_same : forall x z s, update x z s x = z.
intros. unfold update. rewrite cell_beq_refl. auto.
Qed.

Lemma update_different : forall x y z s, ~(y = x) -> update x z s y = s y.
intros. unfold update. rewrite cell_beq_antirefl. auto. auto.
Qed.
```

An additional result is required to prove the `update_below` lemma. The lemma named `ltne` proves that if a number is strictly less than another, they cannot be the same.

```
Lemma ltne : forall q r : nat, q < r -> q <> r.
intros.
assert ({q = r} + {q <> r}).
apply eq_nat_dec.
destruct H0.
intuition.
auto.
Qed.

Lemma update_below : forall q r z s, q < r -> update (Reg r) z s (Reg q) = s (Reg q).
intros. unfold update. rewrite cell_beq_antirefl. auto. intro RqRr.
apply injective_reg in RqRr. generalize RqRr. apply ltne. auto.
Qed.
```

Theorem A similar correctness condition to that in the HOL Light and Isabelle/HOL proofs is used. However, I was forced to manually split the conjunctive in the consequent to achieve the desired result.

```
Theorem correctness :
  forall e m s,
    (forall s' r,
      (forall v, m v < r) ->
      (forall v, s v = s' (Reg (m v))) ->
        exec' (compile e m r) s' Acc = eval e s)
    /\
    (forall s' r,
      (forall v, m v < r) ->
      (forall v, s v = s' (Reg (m v))) ->
        forall x, x < r -> (exec' (compile e m r) s' (Reg x) = s' (Reg x))).
intros. induction e.
auto.
split.
intros. simpl. rewrite update_same. auto. auto.
destruct IHe1 as [IHeq1 IHlo1]. destruct IHe2 as [IHeq2 IHlo2]. split.
intros s' r HmvLr Habscon.
simpl. repeat (rewrite exec'_append; simpl). rewrite update_same.
rewrite IHlo2. rewrite update_same. rewrite IHeq1. rewrite IHeq2. ring.
intros. apply lt_S. auto.
intros. rewrite update_below. rewrite IHlo1. auto.
auto. auto. auto. apply HmvLr. auto. auto.
intros. apply lt_S. auto.
intros. rewrite update_below. rewrite IHlo1. auto.
auto. auto. auto. auto. auto.
intros s' r HmvLr Habscon x HxLr.
```

```
simpl. repeat (rewrite exec'_append; simpl). rewrite update_different.
rewrite IHlo2. rewrite update_below. rewrite IHlo1. auto.
auto. auto. auto. auto.
intros. apply lt_S. auto.
intros. rewrite update_below. rewrite IHlo1. auto.
auto. auto. auto. auto. auto. discriminate.
Qed.
```

Refinement The Coq proof was reformulated with the techniques learned from developing the proof in Agda. It is included in Appendix C.

Documentation Plenty of tutorial and course material exists for learning to use Coq. The documentation of the standard library is very extensive and hides enough detail to prevent the reader being overwhelmed.

Code extraction The Coq to Haskell code extractor is very easy to use and appears to produce code similar to that which would have been written by a human programmer.

10. Agda

Proof of McCarthy-Painter This is an encoding of the seminal compiler correctness problem from McCarthy and Painter (1967) using the accumulating compile function trick from Hutton (2007).

Interface Agda is interfaced through a custom emacs mode which provides many useful features such as unicode character entry, on-the-fly type checking and highlighting of compilation errors.

An interesting feature is the ability to introduce *"holes"* into function and type definitions. These *holes* represent incomplete definitions and can only be filled by expressions satisfying the relevant type constraints. In many cases, Agda can complete the expressions automatically.

These *holes*, when used in the context of *types as propositions* represent unresolved proof obligations.

Standard library Agda focuses on being a programming language first and a theorem prover second. Therefore, many of its idioms are derived from the programming world. For example standard library of data structures and lemmas is highly modularised.

```
module McCarthy where

open import Data.Empty
  using (⊥-elim)
open import Data.Fin
  using (Fin; zero; suc; toℕ)
open import Data.Integer
  using (ℤ)
  renaming (_+_ to _ℤ+_)
open import Data.List
  using (List; []; _::_)
open import Data.Nat
  using (ℕ; _+_; zero; suc; _≟_; _<_; decTotalOrder
```

```
      ; z≤n; s≤s)
open import Data.Nat.Properties
  using (≤-step; ≤-steps; 1 +n≰n)
open import Data.Product
  using (_×_; _,_; ∃)
open import Data.Vec
  using (Vec; lookup)
open import Relation.Nullary
  using (yes; no)
open import Relation.Binary.PropositionalEquality
  using (_≡_; refl; module ≡-Reasoning; cong; trans)
open ≡-Reasoning
  using (begin_; _■; _≡⟨_⟩_)
open import Relation.Binary
  using (module DecTotalOrder)
open DecTotalOrder decTotalOrder
  using () renaming (refl to ≤-refl)
```

Proof scripts Programs look very similar to those written in Haskell. Data structures are written in the generalised algebraic data structure style. One very visible feature of Agda is the availability of unicode characters as identifiers.

```
  -- Source syntax and semantics
data Expr (novs : ℕ) : Set where
  Lit    : (n : ℤ)           → Expr novs
  Var    : (v : Fin novs)    → Expr novs
  _:+_   : (x y : Expr novs) → Expr novs
Source : ℕ → Set
Source novs = Vec ℤ novs
eval : ∀ {novs} → Expr novs → Source novs → ℤ
eval (Lit n)   s = n
eval (Var v)   s = lookup v s
eval (x :+ y) s = eval x s ℤ+ eval y s
  -- Target language and semantics
data Inst : Set where
  -- Load immediate value n into accumulator
  LI     : (n : ℤ) → Inst
```

```
  -- Load from register r into accumulator
  LOAD : (r : ℕ) → Inst
  -- Store accumulator value to register r
  STOR : (r : ℕ) → Inst
  -- Add the value of register r to the accumulator
  ADD  : (r : ℕ) → Inst
Insts = List Inst


Target : Set
Target = (ℤ × (ℕ → ℤ))


update : (ℕ → ℤ) → ℕ → ℤ → ℕ → ℤ
update f k v k' with k ≟ k'
update f k v .k | yes refl = v
update f k v k' | no k≠k' = f k'


exec : Insts → Target → Target
exec [] s = s
exec (LI n    :: is) (a, rs) = exec is (n, rs)
exec (LOAD r :: is) (a, rs) = exec is (rs r, rs)
exec (STOR r :: is) (a, rs) = exec is (a, update rs r a)
exec (ADD r  :: is) (a, rs) = exec is (a ℤ+ rs r, rs)
```

The compiler is written in a continuation passing style, as described in Hutton (2007, section 13.7). It assumes that variables are mapped to the first `novs` registers.

The argument `r` represents the next register (offset by `novs`) that can be safely used as a temporary store. The argument `is` contains instructions to be appended on to the end of the expression's compiled form.

```
compile : ∀ {novs} → ℕ → Expr novs → Insts → Insts
compile          r (Lit n)  is = LI n :: is
compile          r (Var v)  is = LOAD (toℕ v) :: is
compile {novs} r (x :+ y) is = compile r y (STOR (r + novs)
                                :: compile (suc r) x (ADD (r + novs) :: is))
```

Proof style The proof style in Agda is to write terms, representing proofs, that inhabit types, representing propositions. This has two benefits. It comes very naturally to someone from a statically-typed functional programming background and it is very close to the underlying logic, Martin-Löf's (1971) type theory.

The proof steps are natural structures and there is no basic vocabulary to learn, in contrast with the other three tools.

Lemmas Three lemmas are required. The lemma identified `Lemma-Readwrite` states that if a register is written to and then immediately read back, the resulting value is that which has just been stored.

$\perp - elim$ is a function that uses a contradiction, such as $k \equiv k'$ and $k \not\equiv k'$, to inhabit any type.

```
Lemma-ReadWrite : ∀ {rs} k {v} → update rs k v k ≡ v
Lemma-ReadWrite k with k ≟ k
... | yes refl    = refl
... | no  k≢k'    = ⊥-elim (k≢k' refl)
```

Another lemma, `Lemma-Read<Write`, states that if the register being accessed, `k'`, is less than the register most recently updated, `k`, then the update may be bypassed.

```
Lemma-Read<Write : ∀ {rs} k {v} j → j < k → update rs k v j ≡ rs j
Lemma-Read<Write k j  j<k with k ≟ j
Lemma-Read<Write k .k j<k | yes refl  = ⊥-elim ((1 +n≰n) (j<k))
Lemma-Read<Write k j  j<k | no  k≢j   = refl
```

The final lemma states that if a number `m` exists in the set of integers bounded by `n`, then the integer form of `m` must be less than `n`.

```
Fin⇒< : ∀ {n} (m : Fin n) → toℕ m < n
Fin⇒< zero     = s≤s z≤n
Fin⇒< (suc m)  = s≤s (Fin⇒< m)
```

Theorem The correctness theorem states that;

1. *for all* numbers of source variables, `novs`, source states `s`, next available registers `r + novs`, source expressions `e`, target instructions `is` and target states `(acc, rs)`,
2. *if, for all* source variable reference `v`, the value in the equivalent register is the same that in the source state,
3. *then* there *exists* a new register state `rs'` such that;
   a) *for all* source variable reference `v`, the value in the equivalent register is the same that in the source state,
   b) *for all* registers `q` such that they are less than the next available register `r + novs`, the value in new target state is that same as the old one,
   c) *and* expression `e` is compiled into `is` at register offset `r + novs` and executed in starting state `(acc , rs)` *is equivalent to* the execution of the additional instructions `is` with the value of evaluating the expression in the accumulator.

```
Correctness : ∀ novs s r (e : Expr novs) is acc rs                    -- item 1
            → (∀ v → rs (toℕ v) ≡ lookup v s)                         -- item 2
            → ∃ λ rs' → (∀ v → rs' (toℕ v) ≡ lookup v s)              -- item 3a
                      × (∀ q → q < (r + novs) → rs' q ≡ rs q)         -- item 3b
                      × (exec (compile r e is) (acc, rs)              -- item 3c
                         ≡ exec is (eval e s, rs')))
```

This is proved by structural induction over the expressions. The `Lit` case is trivial as only the accumulator is updated and the executions normalise to the same result.

```
Correctness novs s r (Lit n) is acc rs Arss
  = (rs, Arss, (λ _ _ → refl), refl)
```

The `Var` case only updates the accumulator but requires the use of the initial assumption, item 2 above, and `Arss` in the code. This is introduced using the congruence principle of equality.

```
Correctness novs s r (Var v) is acc rs Arss
  = (rs, Arss, (λ _ _ → refl), cong (λ n → exec is (n, rs)) (Arss v))
```

Finally, the inductive case over addition.

Inductive assumptions The inductive assumptions are generated for the compilation of the `y` expression into the tail of the instructions and the compilation of the `x` expression with a higher available register and the updated (eval y s , `rsy`) state.

Register and laws The result of the inductive assumption applied to `x` gives the new register state required for `3` and the required law for `3a`. The law required for `3b` can be constructed using the transitivity principle of equality and the inductive assumptions of `x` and `y` connected with `Lemma-Read<Write`.

Correctness The 3c condition is now proved using a more verbose reasoning syntax, starting with the left-hand side of the 3c condition.

I. Applying an inductive assumption over y and allowing the execution to continue through the `STOR` instruction.

II. Applying an inductive assumption over x and allowing the execution to continue through the `ADD` instruction.

III. The `rsx (r + novs)` term is replaced by its equivalent through it being less than the next available register used when compiling `r` and through the `Lemma-ReadWrite`.

```
Correctness novs s r (x :+ y) is acc rs Arss
  -- Note: Inductive assumptions
  with Correctness novs s r y
         (STOR (r + novs) :: compile (suc r) x (ADD (r + novs) :: is))
         acc rs Arss
... | (rsy, Irssy, Iltry, Icorrecty)
  with Correctness novs s (suc r) x (ADD (r + novs) :: is)
    (eval y s) (update rsy (r + novs) (eval y s))
    (λ v → trans
           (Lemma-Read<Write (r + novs) (to𝕟 v) (⩽-steps r (Fin⇒< v)))
           (Irssy v))
...   | (rsx, Irssx, Iltrx, Icorrectx)
  -- Note: Register and laws
   = (rsx, Irssx
  , (λ q sq⩽r → trans
    (Iltrx q (⩽-step sq⩽r))
    (trans
         (Lemma-Read<Write (r + novs) q sq⩽r)
```

```
        (lltry q sq⩽r)))
-- Note: Correctness
,(begin
  exec
    (compile r y
      (STOR (r + novs) :: compile (suc r) x (ADD (r + novs) :: is)))
    (acc, rs)
-- Note: Correctness I
≡⟨ lcorrecty ⟩
  exec
    (STOR (r + novs) :: compile (suc r) x (ADD (r + novs) :: is))
    (eval y s, rsy)
-- Note: Correctness II
≡⟨ lcorrectx ⟩
  exec is (eval x s ℤ+ rsx (r + novs), rsx)
-- Note: Correctness III
≡⟨ cong (λ y → exec is (eval x s ℤ+ y, rsx))
        (trans (lltrx (r + novs) ⩽-refl) (Lemma-ReadWrite (r + novs))) ⟩
  exec is (eval x s ℤ+ eval y s, rsx)
■))
```

Documentation Like Coq, plenty of tutorial and course material exists for learning to use Agda. The documentation of the standard library is very extensive and the language's concise style helps to make it very understandable.

Code extraction The Agda to Haskell code extractor always produces code which is very likely to maintain the laws proved on it. However, it is almost unreadable by humans due the mechanical, unoptimised process used to extract it. The code produced also hinders many of GHC's compiler optimisations.

11. Summary and Conclusions

In this technical report, I was attempting to get practical experience with four theorem provers through the implementation of a well-known compiler verification problem. This chapter summarises the findings and draws conclusions.

11.1. Interface

HOL Light HOL Light interfaces with the user purely through the O'Caml interactive interpreter. The interpreter can be wrapped with interfaces that supply command history but the system is still limited by the need reinitialise the entire system if any alteration is made.

Isabelle/HOL and Coq The shared Proof General interface is well-developed and features the incremental loading and unloading of proof objects. This allows for a very rapid but robust development process.

The shared interface also enables easier movement between the systems, by relating each of the logics' idioms with a standard vernacular.

Agda Agda's custom emacs interface takes a different approach to Proof General The general 'programming language first' philosophy results in an environment closer to an advanced IDE (i.e. Eclipse) rather than a proof assistant.

Holes are a very natural concept to those who are regularly users of statically-typed languages, as it provides an immediate interface to the type-checker.

11.2. Proof style and development

HOL Light The embedded nature of HOL Light means that the syntax is quite unnatural. The application of tactics in a purely functional style is different to how one would expect to instruct a proof assistant but not unreasonable. However, certain language idioms require expert knowledge.

Isabelle/HOL and Coq Isabelle and Coq use a more imperative style. The assistant is instructed with sequenced tactics altering the implicit proof state. Personally, I find that Coq's commands are more descriptive of their behaviour.

Their respective logics, High Order Logic and the Calculus of Constructions, differ in that the former is a classical logic where the latter is constructive. The main impact of this is that, when using Coq, one cannot assume the law of the excluded middle.

Agda In Agda, there is a return to the purely functional style used by HOL Light. However, this, used in combination with constructive logic expressed in the type system, explicitly makes lemmas and theorems *proof transformers*. This style of problem solving is very familiar to functional programmers, who are used to writing functions on non-dependent data types.

This also makes the logic very exposed to user, reducing the issue of lemma names and proof idioms. One can immediately see what any given proof transformer is doing by looking at its type definition.

However, the lack of tactics does result in large amounts of boilerplate code. The other systems allow easier generalisation and proof reuse.

11.3. Documentation

HOL Light The documentation of the library is complete but the descriptions of the tactics and laws are not entirely obvious. This is not helped by the misdirection caused by its embedded style.

Isabelle/HOL, Coq and Agda All of these tools have excellent documentation and plenty of course and tutorial material. Agda does benefit from having a very exposed logic, allowing easier reading of how library facilities work.

11.4. Code extraction

HOL Light There is no automated method for extracting code from HOL Light theorems.

Isabelle/HOL and Coq The code generated by Isabelle/HOL and Coq is very similar. However, there is little documentation on the soundness and completeness of the translation. It is unclear how much of the languages can be translated and whether it is producing equivalent code.

Agda Agda appears to produce direct syntactic translations of Agda objects in Haskell, increasing confidence in its soundness. However, the resulting code for, even a small library, is almost completely unreadable to a human user and impedes many of a Haskell compiler's optimisations.

11.5. Conclusions

HOL Light does not seem particularly well suited to my needs. Despite aiming for a lightweight approach, there is still quite a steep learning curve in learning its style and library. Combined with HOL Light's poor user interface and lack of code extraction, it just does not seem to suit this project.

Isabelle/HOL is long used and well developed. However, it still hides many of expert experience to get the full benefits of the system. This still applies, but to a lesser extent, with Coq.

Agda, while lacking the tactical niceties of the other proof tools, still manages to produce concise and readable proofs through its highly exposed core logic. The code extractor is not ideal but the high confidence in its soundness and completeness is a good starting point for improvement by the Agda implementers.

The learning curve for Agda also appears to be much shallower then the other three tools, once again due to the exposed logic. The production of the Agda proof (chapter 10) helped me produce the improved proof for Coq in Appendix C.

The combination of the good documentation, a concise, understandable proof style and working code extractor makes Agda an excellent choice for verification projects.

A. Haskell Encoding of the McCarthy-Painter Proof

The source of this section is a Literate Haskell program describing the syntax and semantics of two languages. A compiler between the two languages is formulated and proved using equational reasoning. It is adapted from that encoding described by McCarthy and Painter (1967).

A.1. Abstract Syntax

Basic data types In both the source and target languages, values are represented by the built-in Haskell Int data-type. A data structure is defined to describe target machine memory addresses where Registers are identified with Ints. Haskell Strings are used to identify variables in the source language.

```
type Value = Int
type Register = Int
data Address
   = Ac
   | Reg Register
  deriving (Show, Eq)
type Name = String
```

Source AST The source language consists of numerical constants, labelled variables and addition between two other source language constructs.

```
data Source
   = Const Value
   | Var Name
   | Add Source Source
  deriving Show
```

Target AST The target language consists of a sequence of instructions. Li loads an immediate value into the accumulator, Load loads the value stored in some memory location into the accumulator, Sto stores the value in the accumulator to some memory location and Sum adds a stored value to the accumulator.

```
data TargetInst
  = Li Value
  | Load Register
  | Sto Register
  | Sum Register
  deriving Show
type Target = [TargetInst]
```

A.2. State Representation

Abstract and Concrete State Abstract states are mappings of variable Names to numeric Values. Concrete states are mappings of memory Addresses to numeric Values. The unbound function represents undefined memory.

```
type Abstract = (Name → Value)
type Concrete = (Address → Value)
unbound x = error ("No mapping for name/address '" ++ x ++ "'")
```

A mapping between Registers in the Concrete states back to variable Names in the Abstract states and.

```
type Mapping = (Register → Name)
```

The write function overwrites an existing Concrete state with a new Address mapping.

```
write :: Address → Value → Concrete → Concrete
write x n s = λy → if y ≡ x then n else s y
```

A.3. Semantics

Source semantics The semantics of the Source language are defined in a denotational style.

```
source :: Source → Abstract → Value
source (Const n) _ = n
source (Var v)   s = s v
source (Add x y) s = source x s + source y s
```

Target semantics The semantics of the Target language are defined in a denotational style.

```
target :: Target → Concrete → Concrete
target is s = foldl (flip step) s is

step :: TargetInst → Concrete → Concrete
step (Li n)    s = write Ac n s
step (Load r) s = write Ac (s (Reg r)) s
step (Sto r)   s = write (Reg r) (s Ac) s
step (Sum r)  s = write Ac (s Ac + s (Reg r)) s
```

A.4. Compiler

The compiler has an auxiliary input which tracks lowest unused register for storing temporary values. It increments this value when compiling the second argument of an Addition.

```
compile :: Mapping → Int → Source → Target
compile m t (Const n) = [Li n]
compile m t (Var v) = [Load (m v)]
compile m t (Add x y) = compile m t x
  ++ [Sto t]
  ++ compile m (t + 1) y
  ++ [Sum t]
```

A.5. Proof

Correctness theorem The following property will be proved.

```
(∀ r | m r < t) ∧ (∀ v | s v ≡ s' (Reg (m v)))
 ⇒ source x s ≡ target (compile m t x) s' Ac
```

Proof The anticedant conjuctions are taken as assumptions. They shall be referred to as the *unused register assumption* and the *equivalent states assumption*.

Apply induction to the input Source expression x in the consequent. For the Const case, reduction of the resulting expression gives the desired result.

```
source (Const n) s
  ≡ target (compile m t (Const n)) s' Ac
      {-Unfold compile -}
  ≡ target [Li n] s' Ac
      {-Unfold target and foldl -}
  ≡ step (Li n) s' Ac
      {-Unfold step -}
  ≡ write Ac n s' Ac
      {-Unfold write -}
  ≡ if Ac ≡ Ac then n else s'
      {-Unfold if -}
  ≡ n
```

Similarly the Var case, reduction of the resulting expression and the application of the second assumption gives the desired result.

```
source (Var v) s
  ≡ target (compile m t (Var v)) s' Ac
      {-Unfold compile -}
  ≡ target [Load (m v)] s' Ac
      {-Unfold target and foldl -}
  ≡ step (Load (m v)) s' Ac
      {-Unfold step -}
  ≡ write Ac (s' (m v)) s' Ac
      {-Unfold write -}
  ≡ if Ac ≡ Ac then s' (m v) else s' Ac
      {-Unfold if -}
  ≡ s' (m v)
      {-Equivalent states assumption -}
  ≡ s v
```

For the Add case, one can assume that the inductive hypothesis holds for expressions x and y.

```
source (Add x y) s
  ≡ target (compile m t (Add x y)) s' Ac
        {-Unfold compile -}
  ≡ target (compile m t x ++ [Sto t]
    ++ compile m (t + 1) y ++ [Sum t])
    s' Ac
        {-Distribute target over (++) and unfold step -}
  ≡ let
```

```
    s1 = target (compile m t x) s'
    s2 = write (Reg t) (s1 Ac) s1
    s3 = target (compile m (t + 1) y) s2
    in write Ac (s3 Ac + s3 (Reg t)) Ac
        {-Unfold write and if -}
≡ let
    s1 = target (compile m t x) s'
    s2 = write (Reg t) (s1 Ac) s1
    s3 = target (compile m (t + 1) y) s2
    in s3 Ac + s3 (Reg t)
        {-s2 and s3 equivalent for Reg t using assumptions -}
        {-s2 is equivalent to s' for compiled code -}
≡ let
    s1 = target (compile t x) s'
    s2 = write (Reg t) (s1 Ac) s1
    32 = target (compile m (t + 1) y) s'
    in s3 Ac + s2 (Reg t)
        {-Unfold write -}
≡ target (compile m (t + 1) y) s' Ac
  + target (compile m t x) s' Ac
        {-Inductive hypothesis -}
≡ source x s + source y s
```

Finally, as Haskell is a lazy language, one must ensure that equivalence is preserved for the undefined case.

```
source ⊥ s
    ≡ target (compile m t ⊥) s' Ac
        {-Unfold 'compile' -}
    ≡ target ⊥ s' Ac
        {-Unfold 'target' and 'foldl' -}
    ≡ ⊥
```

B. HOL Light Encoding of the McCarthy-Painter Proof

HOL Light proof by John Harrison, 21st April 2004.

```
(*****************************************************************************
 *
 *  mp.ml
 *
 *  An HOL mechanization of the compiler correctness proof of McCarthy and
 *  Painter from 1967.
 *
 *  From a HOL-4 original by Robert Bauer and Ray Toal
 *
 *  HOL Light proof by John Harrison, 21st April 2004
 *
 *****************************************************************************)

(* ------------------------------------------------------------------------- *)
(* Define a type of strings, not already there in HOL Light.                 *)
(* We don't use any particular properties of the type in the proof below.    *)
(* ------------------------------------------------------------------------- *)

let string_INDUCT,string_RECURSION =
  define_type "string = String (int list)";;

(* ------------------------------------------------------------------------- *)
(* The definitions from Robert's file.                                       *)
(* ------------------------------------------------------------------------- *)

(*
 *  The source language
 *  -------------------
 *
 *  Syntax:
 *
 *  The language contains only expressions of three kinds: (1) simple
 *  numeric literals, (2) simple variables, and (3) plus expressions.
 *)

let exp_INDUCT,exp_RECURSION =
  define_type "exp = Lit num
                   | Var string
                   | Plus exp exp";;

(*
```

```
42   *  Semantics:
     *
     *  Expressions evaluated in a state produce a result.  There are no
     *  side effects.  A state is simply a mapping from variables to
     *  values.  The semantic function is called E.
47   *)

     let E_DEF = new_recursive_definition exp_RECURSION
               `(E (Lit n) s = n)
           /\  (E (Var v) s = s v)
52         /\  (E (Plus e1 e2) s = E e1 s + E e2 s)`;;

     (*
      *  The object language
      *  -------------------
57    *
      *  Syntax:
      *
      *  The target machine has a single accumulator (Acc) and an infinite
      *  set of numbered registers (Reg 0, Reg 1, Reg 2, and so on).  The
62    *  accumulator and registers together are called cells.  There are four
      *  instructions: LI (load immediate into accumulator), LOAD (load the
      *  contents of a numbered register into the accumulator), STO (store
      *  the accumulator value into a numbered register) and ADD (add the
      *  contents of a numbered register into the accumulator).
67    *)

     let cell_INDUCT,cell_RECURSION =
       define_type "cell = Acc
                         | Reg num";;
72
     let inst_INDUCT,inst_RECURSION =
       define_type "inst = LI num
                         | LOAD num
                         | STO num
77                       | ADD num";;

     (*
      *  update x z s is the state that is just like s except that x now
      *  maps to z.  This definition applies to any kind of state.
82    *)

     let update_def =
         new_definition `update x z s y = if (y = x) then z else s y`;;

87   (*
      *  Semantics:
      *
      *  First, the semantics of the execution of a single instruction.
      *  The semantic function is called S.  Executing an instruction in
92    *  a machine state produces a new machine state.  Here a machine
      *  state is a mapping from cells to values.
      *)

     let S_DEF = new_recursive_definition inst_RECURSION
97             `(S (LI n) s = update Acc n s)
           /\  (S (LOAD r) s = update Acc (s (Reg r)) s)
           /\  (S (STO r) s = update (Reg r) (s Acc) s)
           /\  (S (ADD r) s = update Acc (s (Reg r) + s Acc) s)`;;
```

```
(*
 * Next we give the semantics of a list of instructions with the
 * semantic function S'.  The execution of an intruction list
 * in an initial state is given by executing the first instruction
 * in the list in the initial state, which produce a new state s1,
 * and taking the execution of the rest of the list in s1.
 *)

let S'_DEF = new_recursive_definition list_RECURSION
          `(S' [] s = s)
       /\ (S' (CONS inst rest) s = S' rest (S inst s))`;;

(*
 * The compiler
 * ------------
 *
 * Each source language expression is compiled into a list of
 * instructions.  The compilation is done using a symbol table
 * which maps source language indentifiers into target machine
 * register numbers, and a parameter r which tells the next
 * available free register.
 *)

let C_DEF = new_recursive_definition exp_RECURSION
           `(C (Lit n) map r = [LI n])
        /\ (C (Var v) map r = [LOAD (map v)])
        /\ (C (Plus e1 e2) map r =
                APPEND
                    (APPEND (C e1 map r) [STO r])
                    (APPEND (C e2 map (r + 1)) [ADD r]))`;;

(* ------------------------------------------------------------------------- *)
(* My key lemmas; UPDATE_DIFFERENT and S'_APPEND are the same as Robert's.   *)
(* ------------------------------------------------------------------------- *)

let cellth = CONJ (distinctness "cell") (injectivity "cell");;

let S'_APPEND = prove
 (`∀p1 p2 s. S' (APPEND p1 p2) s = S' p2 (S' p1 s)`,
  LIST_INDUCT_TAC THEN ASM_SIMP_TAC[S'_DEF; APPEND]);;

let UPDATE_DIFFERENT = prove
 (`∀x y z s. ~(x = y) ==> (update x z s y = s y)`,
  SIMP_TAC[update_def]);;

let UPDATE_SAME = prove
 (`∀x z s. update x z s x = z`,
  SIMP_TAC[update_def]);;

(*
 * The Correctness Condition
 * -------------------------
 *
 * The correctness condition is this:
 *
 * For every expression e, symbol table map, source state s,
 *      target state s', register number r:
 *
```

```
    *  If all source variables map to registers LESS THAN r,
    *  and if the value of every variable v in s is exactly
162 *  the same as the value in s' of the register to which
    *  v is mapped by map, THEN
    *
    *  When e is compiled with map and first free register r,
    *  and then executed in the state s', in the resulting
167 *  machine state S'(C e map r):
    *
    *  the accumulator will contain E e s and every register
    *  with number x less than r will have the same value as
    *  it does in s'.
172 *)

    let correctness_condition =
        '∀e map s s' r.
            (∀v. map v < r) ==>
177         (∀v. s v = s' (Reg (map v))) ==>
                (S' (C e map r) s' Acc = E e s) /\
                (∀x. (x < r) ==> (S' (C e map r) s' (Reg x) = s' (Reg x)))';;

    (*
182 *  The Proof
    *  ---------
    *
    *  The proof can be done by induction and careful application of SIMP_TAC[]
    *  using the lemmas isolated above.
187 *
    *  The only "hack" is to throw in GSYM SKOLEM_THM and EXISTS_REFL to dispose
    *  of state existence subgoals of the form '?s. ∀v. s v = t[v]', which
    *  otherwise would not be proven automatically by the simplifier.
    *)
192

    let CORRECTNESS_THEOREM = prove
     (correctness_condition,
      MATCH_MP_TAC exp_INDUCT THEN
      REWRITE_TAC[E_DEF; S_DEF; S'_DEF; update_def; C_DEF; S'_APPEND] THEN
197   SIMP_TAC[ARITH_RULE '(x < y ==> x < y + 1 /\ ~(x = y)) /\ x < x + 1'; cellth;
               UPDATE_SAME; UPDATE_DIFFERENT; GSYM SKOLEM_THM; EXISTS_REFL]);;
```

C. Reformulated Coq Encoding of the McCarthy-Painter Proof

```
Require Import Arith Bool BoolEq List Omega.

(* Source Language *)
Inductive expr : Set :=
  | Con : nat -> expr
  | Var : nat -> expr
  | Add : expr -> expr -> expr.

Fixpoint eval e s := match e with
  | Con n   => n
  | Var v   => s v
  | Add x y => eval x s + eval y s
end.

(* Target Language *)
Inductive inst : Set :=
  | Li    : nat -> inst
  | Load  : nat -> inst
  | Store : nat -> inst
  | Sum   : nat -> inst.

Definition update (rs : nat -> nat) (k v k' : nat) := match beq_nat k' k with
  | true  => v
  | false => rs k'
end.

Fixpoint step (i : inst) (s' : nat * (nat -> nat)) := match i with
  | Li    n => (n, snd s')
  | Load  r => (snd s' r, snd s')
  | Store r => (fst s', update (snd s') r (fst s'))
  | Sum   r => (fst s' + snd s' r, snd s')
end.

Fixpoint exec is s' := match is with
  | nil     => s'
  | i :: is => exec is (step i s')
end.

(* Compiler *)
Fixpoint compile (m : nat -> nat) (r : nat) (e : expr) (is : list inst) := match e with
  | Con n   => Li n :: is
  | Var v   => Load (m v) :: is
  | Add x y => compile m r y (Store r :: compile m (S r) x (Sum r :: is))
```

```
end.

(* Lemmas *)
Lemma update_same :
  forall rs key value,
    update rs key value key = value.
intros. unfold update. rewrite <- beq_nat_refl. auto.
Qed.

Lemma lt_is_ne :
  forall q r : nat, q < r -> q <> r.
intros. assert ({q = r} + {q <> r}).
apply eq_nat_dec. destruct H0. omega. auto.
Qed.

Lemma n_nlt_n :
  forall n, (n < n) -> False.
intros. omega.
Qed.

Lemma update_less :
  forall rs hi value lo,
    lo < hi -> update rs hi value lo = rs lo.
intros. unfold update. case_eq (beq_nat lo hi).
intros. assert (lo = hi). apply beq_nat_eq. auto.
  rewrite H1 in H. apply n_nlt_n in H. elim H.
auto.
Qed.

(* Correctness *)
Theorem correctness :
  forall m s e r is acc rs,
    (forall v, m v < r) -> (forall v, rs (m v) = s v) ->
    exists rs', (forall v, rs' (m v) = s v)
              /\ (forall q, q < r -> rs' q = rs q)
              /\ exec (compile m r e is) (acc, rs) = exec is (eval e s, rs').
intros until s. induction e.
intros. exists rs.  repeat (split; auto).
intros r is acc rs Hltr Heqs. exists rs.
  repeat (split; auto). simpl. rewrite Heqs. auto.
intros r is acc rs Hltr Heqs.
destruct IHe2 with (r := r) (acc := acc) (rs := rs)
  (is := (Store r :: compile m (S r) e1 (Sum r :: is))).
auto. auto. destruct H as [Hyltr H].  destruct H as [Hyeqs Hycor].
destruct IHe1 with (r := S r) (acc := eval e2 s)
  (rs := (update x r (eval e2 s))) (is := (Sum r :: is)).
intros. apply lt_S. auto.
intros. rewrite update_less. auto. auto.
  destruct H as [Hxltr H]. destruct H as [Hxeqs Hxcor].
exists x0. split. auto. split. intros. rewrite Hxeqs. auto.
  rewrite update_less. rewrite Hyeqs.
auto. auto. auto. auto.
simpl. rewrite Hycor. simpl. rewrite Hxcor. simpl.
  rewrite Hxeqs. rewrite update_same. auto. auto.
Qed.
```

Bibliography

M. Bolingbroke and S. Peyton Jones. Supercompilation by evaluation. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 135–146. ACM Press, 2010.

T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, 1988.

N.A. Danielsson. Operational Semantics Using the Partiality Monad. Contributed talk at Dependently Typed Programming, 2010.

F. Haftmann. From higher-order logic to Haskell: there and back again. *ACM/SIGPLAN Workshop Partial Evaluation and Semantics-Based Program Manipulation*, pages 155–158, 2010.

J. Harrison. HOL Light tutorial (for version 2.20). URL: `http://www.cl.cam.ac.uk/~jrh13/hol-light/tutorial_220.pdf`, December 2007.

J. Harrison. The HOL Light theorem prover. URL: `http://www.cl.cam.ac.uk/~jrh13/hol-light/`, June 2010.

G. Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

P. A. Jonsson and J. Nordlander. Positive Supercompilation for a higher order call-by-value language: Extended Proofs. Technical report, Luleå University of Technology, 2008.

D. Krustev. A simple supercompiler formally verified in Coq. In *The Second International Workshop on Metacomputation in Russia*, 2010.

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

P. Martin-Löf. A theory of types. Technical report, University of Stockholm, 1971.

J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. *Mathematical Aspects of Computer Science*, 19:33–41, AMS, 1967.

N Mitchell. Rethinking supercompilation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 309–320. ACM, 2010.

N. Mitchell and C. Runciman. Uniform boilerplate and list processing. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 49–60. ACM, 2007.

N. Mitchell and C. Runciman. A supercompiler for core Haskell. In *IFL 2007*, volume 5083 of *LNCS*, pages 147–164. Springer, May 2008.

M. Naylor and C. Runciman. The Reduceron: Widening the von Neumann bottleneck for graph reduction using an FPGA. In *Implementation and Application of Functional Languages (IFL 2007, Revised Selected Papers)*, volume 5083 of *LNCS*, pages 129–146. Springer, 2008.

M. Naylor and C. Runciman. The Reduceron Reconfigured. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 75–86. ACM, 2010.

T. Nipkow, L. Paulson, M Wenzel, G. Klein, F. Haftmann, T. Weber, and J. Hlzl. A bluffer's glance at Isabelle. URL: `http://www.cl.cam.ac.uk/research/hvg/Isabelle/overview.html`, June 2010.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology and Göteborg University, 2007.

U. Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer Berlin / Heidelberg, 2009.

S.L. Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1), January 2003.

S.L. Peyton Jones, C.V. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference*, pages 249–257, 1993.

J. S. Reich. Towards a Dependable Supercompiler for the Reduceron. Qualifying dissertation, University of York, August 2010.

J. S. Reich, M. Naylor, and C. Runciman. Supercompilation and the Reduceron. In *The Second International Workshop on Metacomputation in Russia*, 2010.

M. H. Sørensen. Turchin's supercompiler revisited: An operational theory of positive information propagation. Master's thesis, Københavns Universitet, Datalogisk Institut, 1996.

V. F. Turchin. A supercompiler system based on the language Refal. *ACM SIGPLAN Notices*, 14(2):46–54, ACM Press, 1979.

M. van Eekelen and R. Plasmeijer. Concurrent Clean language report (version 2.0). Technical report, University of Nijmegen, 2001.