Towards a Dependable Supercompiler for the Reduceron

Qualifying Dissertation

Jason S. Reich

29th June 2010
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 introduces a particular instance of the problem: compiling a small functional language to be executed on a special-purpose FPGA-based soft-processor, the Reduceron. In particular, improving program execution performance through supercompilation, a compile-time optimisation.

Literature on the topic of compiler verification is surveyed, focusing on theorem proving techniques and the application of algebraic abstractions. Initial results of a prototype supercompiler for the Reduceron are presented and experiences of early investigations into theorem proving tools are documented.

A proposal is formed, based on the field survey and these preliminary results. The proposal outlines the research goals and aims, and presents a strategy for achieving them.

Keywords: functional programming, compiler design, compile-time optimisation, supercompilation, formal methods, theorem proving, the Reduceron
# Contents

I. Introduction ................................................. 6
   1. Introduction ............................................. 7
      1.1. The Reduceron ...................................... 7
      1.2. Supercompilation ................................... 8
      1.3. Structure of this Report ............................ 9

II. Field Survey and Review .................................. 10
   2. Compiler Verification through Formal Reasoning ........ 11
      2.1. Adding Up ........................................... 11
      2.2. Checking Your Work ................................ 12
      2.3. Learning Algebra ................................... 13
      2.4. A Different Angle ................................... 14
      2.5. Equivalent Graphs .................................. 15
      2.6. Finding Mistakes ................................... 16
      2.7. Theory into Practice ............................... 17
      2.8. ‘Real’ Languages .................................... 18
      2.9. Supercompiler Correctness ......................... 18
      2.10. Review ............................................. 19

III. Preliminary Findings ...................................... 21
   3. A Supercompiler for the Reduceron ..................... 22
      3.1. Introduction ........................................ 22
      3.2. Our Source Language ................................ 22
      3.3. The Reduceron Architecture ......................... 23
      3.4. Primitive Redex Speculation ....................... 25
      3.5. Benchmark Programs ................................ 26
      3.6. Our Supercompiler Prototype ....................... 28
         3.6.1. Start .......................................... 28
         3.6.2. Termination .................................... 29
         3.6.3. Drive ........................................... 30
3.6.4. Generalisation ........................................... 31
3.6.5. Tie .................................................. 34
3.6.6. Clean Up ............................................ 35
3.7. A Synergistic Effect of Supercompilation .................. 35
3.8. A Potentially Obstructing Effect of Supercompilation ..... 36
3.9. Primitive Redex Speculation Candidate Lifting ............ 36
3.10. Performance Results ................................... 38
3.11. Conclusions and Future Work ............................ 42

4. Mechanical Theorem Proving ................................ 44
4.1. Introduction ............................................ 44
4.2. HOL Light .............................................. 44
4.3. Isabelle/HOL ............................................ 45
4.3.1. Haskabelle ........................................... 46
4.3.2. Code Extraction ..................................... 46
4.4. Coq ..................................................... 46
4.4.1. Code Extraction ..................................... 47
4.5. Summary ............................................... 47

IV. Research Proposal ........................................... 48
5. Research Proposal .......................................... 49
5.1. Overall Goals .......................................... 49
5.2. Specific Aims ........................................... 49
5.3. Strategy ................................................ 50
5.3.1. Phase A — Verification Tools ......................... 50
5.3.2. Phase B — A Supercompiler for the Reduceron .... 50
5.3.3. Phase C — Simplifying and Restructuring the Supercompiler 50
5.3.4. Phase D — Verifying the Supercompiler .............. 51
5.3.5. Phase E — Classifying Behaviour of the Supercompiler and the Reduceron 51
5.4. Current Progress ...................................... 52
5.5. Immediate plan ......................................... 52

Bibliography ................................................... 54
A. Haskell Encoding of the McCarthy-Painter Proof .......... 58
B. HOL Light Encoding of the McCarthy-Painter Proof ...... 61
C. Partial Isabelle/HOL Encoding of the McCarthy-Painter Proof 65
D. Partial Coq Encoding of the McCarthy-Painter Proof ...... 67
Part I.

Introduction
1. Introduction

My research is investigating the dependibility of supercompilation, a compile-time optimisation for programs written in lazy functional languages.

Software is only as correct as the weakest component used in its production, verification and execution. The source code may be shown to be correct with respect to the source semantics and the executing machine may be generated from the target language semantics. However, unless the compiler and its optimisations are verified and correct, the wrong target code may be being executed.

Modern compilers consist of a pipeline of a number of interfaced stages. These stages can include parsing the source program text, checking well-formedness and typing constraints, several stages of translation to a variety intermediate and target languages, optimisation and pretty-printing the object code.

Supercompilation (Jonsson and Nordlander [2008], Mitchell and Runciman [2008], Sørensen [1996], Turchin [1979]) one such optimisation. It 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.

An advantage of a source-to-source optimisation is that existing development software, such as preprocessors and compilers, are all still applicable once the optimisation phase is added. In particular, an existing verified compiler will still produce correct object code for the supercompiled program. However, correctness of the supercompiler itself must be demonstrated.

1.1. The Reduceron

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

The machine is a quite sophisticated in its approach, but 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 transformation to improve program performance when executed on traditional imperative architectures.
1. Introduction

The Reduceron is not a traditional imperative architecture. Taking a fresh approach to the optimisation phase of the compiler is likely to yield benefits. These would come through tailoring the transformations to the particular execution model.

1.2. Supercompilation

Supercompilation involves the following component processes:

- **Drive**: Inlines function bodies and simplifies the result.
- **Check Termination**: Tests expressions to stop driving.
- **Generalise**: Restructure an expression so that it is similar to a previously derived expression.
- **Tie**: Replace an expression with an equivalent function application to either a previously derived or a fresh function definition.
- **Clean Up**: Inline simple function bodies, fold constants and remove dead code.

A fuller description of a particular supercompiler implementation is presented in chapter 3.

For the supercompiler to be correct, each of these processes must maintain semantic equivalence for any valid expression. Furthermore, the driving phase must terminate for any valid expression and the supercompilation process as a whole must terminate for any given program.

Hand proofs of correctness have been completed for specific supercompilers (Jonsson and Nordlander, 2008; Sørensen, 1996) but a mechanised proof of supercompiler correctness (with an extracted implementation) would give stronger guarantees of dependability. Furthermore, mechanising the proof will likely give insights into how best to structure similar optimisations for verification.

A previous project (Reich, 2009) investigated bringing a supercompilation optimiser to the Reduceron platform. However, this project suffered from an overcomplicated design that caused many development difficulties and long compilation times. Most importantly, it difficult to ensure that semantic equivalence would be maintained and that the optimisation was guaranteed to terminate for any given program.

A simplified design would likely improve our understanding and confidence in the optimisation. However, it could also lead to software that is more responsive to formal verification. Producing a proof that the supercompiler is guaranteed to produce semantically equivalent programs and terminate any input would be a starting point for producing a dependable Reduceron stack.
Chapter 2 — Compiler Verification through Formal Reasoning This chapter surveys compiler verification techniques from a history perspective. In particular, it focuses on the use of theorem proving, algebraic techniques to structure the proofs and the use of mechanical systems to aid the verification process.

Chapter 3 — A Supercompiler for the Reduceron This chapter consists of an extended version of the paper accepted by the Second International Workshop on Meta-computation in Russia. (Reich et al., 2010)

It discusses the particular benefits that supercompilation can provide for programs running on the Reduceron, an area where program performance can be worsened and a possible technique for correcting the issue. A high-level description of the prototype supercompiler algorithm is include.

Chapter 4 — Mechanical Theorem Proving An experience report is presented, describing some early impressions of three proof assistants; HOL Light, Isabelle/HOL and Coq. The report contains brief descriptions of these tools capabilities and contrasts approaches to theorem proving.

Chapter 5 — Research Proposal A proposal for a research program is outlined. Immediate and long-term research goals are detailed and a strategy for achieve these goals is formed.
Part II.

Field Survey and Review
2. Compiler Verification through Formal Reasoning

This chapter surveys some of the literature in the field of compiler verification. We discuss some of the topic’s seminal pieces, highlight some of the techniques used to structure compiler verification and consider some of the tools used to aid the verification process.

2.1. Adding Up

The earliest example of compiler verification is reported by McCarthy and Painter (1967). In their seminal piece, they produce a hand proof for the correctness of 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 consist of four instructions:

- \textbf{LI n} — Load immediate value into the accumulator.
- \textbf{LOAD r} — Load the value of register \( r \) into the accumulator.
- \textbf{STO r} — Store the value of the accumulator to register \( r \).
- \textbf{ADD r} — Add the value of register \( r \) to the accumulator.

McCarthy and Painter encode the semantics of the two languages into a \textit{first-order predicate logic}. States for the source semantics (and the target semantics) are defined as predicates over variables (or registers) 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 \textit{partial equality} between states is defined. For states \( \sigma_1 \) and \( \sigma_2 \) and where \( A \) is a set 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 the 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 “\textit{a complete revision of the formalism}.” (McCarthy and Painter, 1967)
2. Compiler Verification through Formal Reasoning

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.

2.2. Checking Your Work

By 1972, there was work investigating the use of proof assistants and mechanised logics to prove correctness properties of compilers. Using LCF (Milner, 1972), an implementation of Scott’s Logic of Computable Functions, Milner and Weyhrauch prove the example from McCarthy and Painter (1967) and begin to demonstrate a machine checkable proof for an "ALGOL-like language with assignments, conditionals, whiles and compound statements." (Milner and Weyhrauch, 1972)

Milner and Weyhrauch (1972) cite the work of other researchers who have either mechanised proofs for languages of about the complexity of McCarthy and Painter (1967) or hand proofs of correctness for ALGOL and LISP compilers. Their contribution involves combining the two goals of machine-checkable proof and the correctness of a more substantial compiler. Milner and Weyhrauch (1972) also use a different formulation of the constructs involved, based on the work of Lockwood Morris (1973).

Instead of the first-order predicate logic used by McCarthy and Painter, Milner and Weyhrauch used a definition of the semantics based in "the theory of typed lambda calculus, augmented by a powerful induction rule".

This means that the “[semantic] meaning of the program is a partial computable function from states to states,” (Milner and Weyhrauch, 1972) the denotational semantics for a language. Furthermore, states themselves are represented as functions from names to values. Many of the low-level laws that must be explicitly stated for programs declared in first-order predicate calculus are implicit in typed lambda calculus, leading to smaller, more manageable proofs.

The proof can be summarised by Figure 2.1. S and T are the source and target languages.

\[
\begin{array}{ccc}
S & \xrightarrow{\text{comp}} & T \\
\downarrow MS & & \downarrow MT \\
\downarrow SIMUL & & \downarrow T
\end{array}
\]

Figure 2.1.: Commutative diagram from Milner and Weyhrauch (1972)
2.3. Learning Algebra

$MS$ and $MT$ are the semantics of those languages expressed as partial functions from states to states. $S$ and $T$ are relevant state functions, mapping names to values. The compiler function, $comp$, takes source programs to target code. The proof is that a retrieval function, $SIMUL$, can be constructed to complete the diagram.

The paper demonstrates a complete, machine-checked proof of the McCarthy and Painter compiler using a lambda calculus encoding. The further goal of machine-checked proof for a compiler for an ALGOL-like language is left incomplete. It is stated that they have "no significant doubt that the remainder of the proof can be done on the machine."

(Milner and Weyhrauch, 1972)

Although they remain uncommitted about the algebraic approach to their proof, Milner and Weyhrauch (1972) are of the opinion that "for machine-checked compiler proofs some way of structuring the proof is desirable."

2.3. Learning Algebra

Milner and Weyhrauch acknowledge that their algebraic approach is inspired by discussions with Lockwood Morris. He develops the details of this algebraic method in the subsequently published paper, ‘Advice on structuring compilers and proving them correct.’ (Lockwood Morris, 1973)

The “essence” of the advice presented in the paper consists of the commutative diagram, Figure 2.1. Although different notation is used, the diagram conveys that same information.

However, Lockwood Morris explicitly discusses the algebraic interpretation of the diagram’s components. The languages, $S$ and $T$, and meanings, $S'$ and $T'$, are described as “heterogeneous (universal) algebras,” much like algebraic data types in modern languages. In particular, $S$ and $T$ are initial algebras, such that there is a unique homomorphism to any other algebra.

The semantics $MS$ and $MT$ are mappings of respective languages to meanings. The functions $comp$ and $SIMUL$ are mappings of between the languages and the meanings, respectively. These, under the algebraic interpretation, become homomorphisms.

For an example, we shall discuss a homomorphism between two datatypes. Consider algebraic datatypes $A$ and $B$. These each contain an arbitrary number of constructors which may take any number of arguments of its own type. For example, $C_0$ is a constructor in the $A$ datatype that takes $m0$ arguments of type $A$. A homomorphism, $\phi$ from $A$ to $B$ would (for all $x$ and some of $y$) satisfy Equation 2.1.

$$\phi(C_x(z_0, z_1, ..., z_mx)) = D_y(\phi(z_0), \phi(z_1), \phi(z_mx))$$ (2.1)

The very definitions of $MS$, $MT$ and $comp$ in his examples show that they are homomorphisms. Due to the initiality of the $S$ and $T$ languages, the $comp$, $MS$ and $MT$
functions are unique homomorphisms. All that remains is to show that encode is a homomorphism, where encode is the inverse of SIMUL. Once this is confirmed, by the unique extension lemma, Equation 2.2 must be true, as there is only one homomorphism from an initial algebra to any other.

\[ \text{encode} \circ \text{MS} = \text{MT} \circ \text{comp} \] (2.2)

The example compiler in Lockwood Morris (1973) compiles a small, ALGOL-like language to a flowchart representation of a stack machine. However, the proof is left incomplete. Thatcher et al. (1980) reasons that the difficulty is largely the lack of algebraic structure on the “right-hand side” of the commutative diagram.

Their paper, ‘More on advice on structuring compilers and proving them correct’, extends the Lockwood Morris (1973) work by enlarging the source language further and introducing a categorical representation of flowcharts over a stack machine as the target language. Most importantly, Thatcher et al. (1980) complete the proof, where Lockwood Morris (1973) could not, by the same unique extension principle.

Thatcher et al. (1980) commend the “extremely powerful methodology” as “no structural induction is required for the definition of the arrows or the proof.” However, they admit that the proofs that encode is a homomorphism are “considerably longer and more cumbersome than [...] expected.” (Thatcher et al., 1980)

One wonders whether this is a problem with their representations, notation or an issue with the general algebraic technique. Furthermore, does this imply that the proofs will become intractable as large languages are introduced?

2.4. A Different Angle

Meijer (1994) takes a different view of compiler correctness. Instead of building a compiler and proving it correct, why not “improve a correct compiler.”

The denotational semantics of the source language, MS in Figure 2.1, can be expressed as a functional program that takes a source program and produces a function mapping source states to values. Can we decompose this function to produce the other functions (comp, MT and SIMUL) in the commutative diagrams?

Meijer (1994) demonstrates such a decomposition for a first-order imperative source language being compiled to a three-address code register machine. He accomplishes this through a divide and conquer approach, appealing to the same algebraic concepts of Lockwood Morris (1973) and Thatcher et al. (1980).

The commutativity diagram is split as shown in Figure 2.2. Notable differences from previous work include the explicit realisation of various homomorphisms as catamorphisms, signified by the ‘banana brackets.’
Catamorphisms have been referred to in the previous literature as unique homomorphisms but were functions constructed to fit the homomorphic structure. Meijer (1994) instead produces functions that can be expressed using a standard catamorphic operator, ensuring their catamorphic (and homomorphic) properties. In this sense, we can think of them as a generalisation of the functional fold operator. (Meijer et al., 1991) Meijer (1994) formally transforms the catamorphic form of the source semantics into the other homomorphisms in the diagram. Through this, he succeeds in producing a compiler that is correct with respect to the rest of the diagram.

This technique seems particularly suited to first-order languages. “[Things] get notoriously hard when the domains themselves become recursive, especially when function spaces are involved.” (Meijer 1994) So, this technique may not be particularly suited to produce dependable compilers for a higher-order functional language.

2.5. Equivalent Graphs

Lazy functional languages can benefit from concise definitions through high-level abstractions. These are executed by graph-reduction machines, of which the G-machine (Augustsson 1984) is the most widely studies. In his thesis, Lester (1988) proves that compiling a lazy function language to a G-machine implementation is correct. Lester (1988) defines the denotational semantics and the operational semantics of the lazy functional language. The operational semantics are modelled as G-machine
2. **Compiler Verification through Formal Reasoning**

reductions.

In terms of the previous commutative diagrams \([\text{Figure 2.1}]\), the denotational semantics still represent \(MS\) morphism on the diagram. However, the operational semantics represent the composition of \(MT\) and \(\text{comp}\), with the target meanings, \(\hat{T}\), at a higher level of abstraction to the state machine specifications used by the previous literature.

Interestingly, there is a return to the decoding of target meanings into source meanings, rather than the inverse that has been used by Lockwood Morris (1973), Thatcher et al. (1980) and Meijer (1994). The reasons for why a decoding (concrete to abstract state) relation is used, rather than an encoding (abstract to concrete state) relation is not documented. This may be because it is more natural to consider the retrieval of a less-defined state from the more-defined.

Lester (1988) begins with a proof for a small lazy functional language that only accounts for programs that produce a result (terminate) under the source semantics. The proof makes use of fixpoint induction over states and structural induction of the abstract syntax representations of the source language. Similar techniques are then used to extend the proof to include built-in functions, data structures, optimisations and other analyses.

2.6. **Finding Mistakes**

Mintchev (1995), as part of his thesis, develops a mechanised proof of a form similar to that of Lester (1988), his supervisor. Proofs are developed in Isabelle/LCF and then again in a theorem prover that is developed as part of the thesis. Mintchev’s theorem prover operates on a subset of the Haskell language, dubbed \(\text{Core}\).

In the development of an Isabelle/LCF proof, a error was found in the original hand proof by Lester (1988). This highlights the benefits of using proof assistants for checking theorem validity. However, the recursive nature of the source language is highlighted as a source of difficulty for the logic, “taking up 85% of over 2000 lines of definitions, axioms, theorems and tactics.” (Mintchev 1995)

Mintchev (1995) uses his own theorem prover to develop a proof for a variation of the problem used in Lester (1988). The operational semantics are for a spineless \(G\)-machine (Burn et al., 1988) as “it avoids unnecessary updates after each reduction step.” (Mintchev 1995)

Another differentiating area is the use of monads, now a common functional programming abstraction, to describe the “heap free operational semantics.” Mintchev (1995) states that it was originally used as useful abstraction for defining the abstract machine but it “paid off in the proof of correctness.” This is because these functional constructs can be easily refactored as imperative statements.

The thesis does not discuss facilities for checking the soundness of the proofs generated using the bespoke tool. It may be assumed that this is due to its small number of
2.7. Theory into Practice

One of the perceived advantages of the Mintchev (1995) theorem prover was that it operated directly on program code, maintaining the direct link between theorems and the executable object. Berghofer and Strecker (2004) investigate using Isabelle/HOL to verify a compiler for a simplified, Java-like language, μJava. In their approach, they use Isabelle’s code extraction facility to produce an ML program based on the correctness theories.

In addition to verifying the semantic equivalence of compiled programs, they also prove a number of other compiler functions correct such as type-checking and pretty printing, as is shown in Figure 2.3. The result that the program is well-typed becomes a necessary precondition for the correctness of code translation and the guarantee of pretty-printing is evidently essential.

Little documentation is provided for how they structure the program to aid the verification process. Instead, they focus on the correctness conditions that are verified and the preconditions that these are dependent on.

One point of interest is that they abstract the target machine to the level of specific state structures, “the heap, a local operand stack ... and local variable assignment” despite the “structure of states on the JVM [, Java Virtual Machine,] level [being] more complex.” Berghofer and Strecker (2004) do question the dependability of the Isabelle code extraction facility as it is “a complex piece of code and thus prone to errors.” It is unclear what guarantees have been developed on the correctness of Isabelle code extraction, since 2004.

Figure 2.3.: The processing steps from Berghofer and Strecker (2004). The greyed section represents the verified component.
2. Compiler Verification through Formal Reasoning

2.8. ‘Real’ Languages

Leroy (2009) represents a monumental achievement in the field of compiler verification; the result of four years of effort. The artefact produced is a mechanically verified compiler implementation which translates a C-like language, Cminor, to PowerPC assembly code.

The system has been developed and proved correct using the Coq proof assistant. Similar to Berghofer and Strecker (2004), a code extraction tool processes the theorems to produce an executable Caml program. Unlike the Isabelle/HOL extraction facility (circa 2004), the Coq code extractor appears to be held in higher esteem. Letouzey (2008) Even still, Leroy (2009) indicates that one of their research aims is to investigate the “feasibility of formally verifying Coq’s extraction mechanism and a compiler from Mini-ML to Cminor.”

The correctness condition used by Leroy (2009) is; for all source programs, if compilation completes successfully and the source program evaluates to a non-error value, then the compiled program should evaluate to the same value. This condition is valid for the compiler that fails for all input. Leroy (2009) states that this issue is solved through non-formal testing methods.

An essential theorem is that if any two compilers are verified, then their composition is also verified. This enables the compilation process to be split into several independent proof efforts, which can easily be composed to form a proof for an entire compiler.

One highlighted area of weakness is “do the formal semantics of Cminor and PPC, along with the underlying memory model, capture the intended behaviours?” (Leroy 2009) As languages of increasing complexity are verified, it may be more difficult to maintain confidence in the initial specifications. Possible solutions include: (a) ensuring that the language specification remains small enough that it can be checked successfully, (b) using a variety of semantic representations simultaneously and checking for inconsistency (c) or generating the lower-level virtual or hardware machines from the semantic specifications.

The only other concern that Leroy (2009) raises is that the Coq logic or implementation might not be sound. This is highly unlikely as the kernel of Coq has been kept intentionally small. Furthermore, “proofs mechanically checked ... are orders of magnitude more trustworthy than even carefully hand-checked mathematical proofs.” (Leroy 2009)

2.9. Supercompiler Correctness

Some components of supercompilation correctness are founded in other mathematical proofs. For example, the proof that driving must terminate under homeomorphic embedding is derived from Kruskall’s Tree Theorem. (Kruskal 1960)

A number of hand-proofs of correctness have been produced for entire supercompiler designs. According to Klyuchnikov (2010), prior to his own verification of the HSOC
supercompiler, Sørensen et al. (1996) and Jonsson and Nordlander (2008) had published proofs of correctness for their respective supercompilers.

However, none of these proofs have been mechanised and, correspondingly, no automatically generated implementation has been produced. It is uncertain how easily this proofs could be machined checked as, in many places, informal defined ‘lemmas’ have been used.

Krustev (2010) is soon to publish about “a simple supercompiler formally verified in Coq.” The abstract highlights to particular features that made the verification possible. “First, a very limited object language; second, decomposing the supercompilation process into many sub-transformations, whose correctness can be checked independently.”

2.10. Review

Since McCarthy and Painter (1967), the formal verification of compilers has been an active topic due to its impact on the dependability of other compiled software. Leroy (2009) states that “a great many on-paper proofs for program analyses and compiler transformations have been published,” to the extent that only a small proportion were discussed in the journal paper.

Similarly, this survey only accounts for a sample of the available literature on the topic of formal reasoning about compilers. The aim has been to focus on research that uses algebraic abstractions to structure proofs or that uses mechanised tools to produce dependable compiler software.

Proofs assistants not only help to ensure that a proof is sound. They are essential in managing the complexity of larger language specifications and more advanced compiler designs. Krustev (2010) Milner and Weyhrauch, 1972 Mintchev (1995) Furthermore, many modern proof assistants provide mechanisms for extracting executable programs from theorems, increasing confidence that the resulting software is a valid implementation of the formal specification. (Berghofer and Strecker 2004 Leroy, 2009)

Induction over source language or states is a common proof method. (Lester, 1988 McCarthy and Painter, 1967) Alternatively, mathematical abstractions, such as initial algebras and unique homomorphisms, are used to reduce the proof by appealing to existing theorems. (Lockwood Morris 1973 Thatcher et al., 1980)

Meijer (1994) suggested using these abstractions to transform the source semantics into the necessary compiler. The proof would be apparent from the formal transformation process. This technique appears not to have been widely adopted due to the difficulty in applying it to higher-order languages.

Some hand-proofs have been completed for certain supercompiler designs. (Jonsson and Nordlander 2008 Klyuchnikov, 2010 Sørensen et al., 1996) It is yet to be seen that the
2. Compiler Verification through Formal Reasoning

proof of a supercompiler can be mechanised. This would facilitate the extraction of a dependable supercompiler implementation.
Part III.

Preliminary Findings
3. A Supercompiler for the Reduceron

This chapter contains an extended version of the paper accepted by the Second International Workshop on Metacomputation in Russia. (Reich et al., 2010)

3.1. Introduction

Functional programming is a distinctive paradigm that has scope for exploiting non-standard technologies at every stage of computation. Supercompilation and the Reduceron are two such technologies.

Supercompilation (Sørensen et al., 1996; Turchin, 1979) is a metaprogramming technique that, at compile-time, evaluates (drives) programs until an unknown is required and then proceeds by case analysis (residuates). Among other benefits, it can remove intermediate data structures and specialise higher order functions, with corresponding performance gains at execution time.

The Reduceron is an FPGA-based soft processor for executing lazy functional programs by graph reduction (Naylor and Runciman, 2008, 2010). The special-purpose processor can perform in parallel many of the steps required for each reduction, whereas conventional architectures need to perform these steps serially.

Does a combination of these technologies lead to further improvements in performance? Are these techniques conflicting, compatible or even mutually beneficial? In this paper, we discuss how the two may interact and present preliminary findings from a prototype supercompiler for the Reduceron source language.

3.2. Our Source Language

Our source language (Naylor, M., 2008) is close to subsets of both Haskell 98 (Peyton Jones, S.L. et al., 2003) and Clean (van Eckelen and Plasmeijer, 2001). It supports algebraic data types, uniform pattern matching by construction, local variable bindings, and various primitive integer operations.

Abstract syntax for our source language is given in Figure 3.1. In addition to the annotated symbols, \( x \) and \( y \) range over expressions. Overlining and pluralisation indicate sequences of productions. For example \( \overline{vs} \) represents a sequence of variable names. All programs declare an integer expression bound to \texttt{main}.
3.3. The Reduceron Architecture

\[ \text{prog} := f \overline{vs} = x \]

\[ \text{exp} := v \ (\text{variables}) \]
\[ \quad | c \ (\text{constructors}) \]
\[ \quad | f \ (\text{functions}) \]
\[ \quad | n \ (\text{integers}) \]
\[ \quad | x \overline{x} \ (\text{applications}) \]
\[ \quad | \text{case } x \text{ of } \overline{c \overline{vs}} \rightarrow x \]
\[ \quad | \text{let } \overline{v} = x \text{ in } y \]

Figure 3.1.: Abstract syntax for our source language.

Listing 3.1 shows an encoding of a program in our source language. This program (somewhat inefficiently) doubles each element in a range, calculates the sum and prints the result.

The basic compiler (before the introduction of supercompilation) first reduces all pattern matching to combinations of one-level case distinctions. A case-elimination phase then translates algebraic data constructors and case expressions to functions and function applications respectively, using a variation of the Scott encoding. For example, after case elimination our range function is as displayed in Listing 3.2. In every function body, a bottom-up traversal inlines saturated applications of non-primitive functions.

Finally, the compiler generates a compact encoding of each function body. The encoding phase must take into account the design parameters of the Reduceron. Encoded forms of function bodies are constrained by limits on the size of the top-level spine, the number and size of nested applications, and the number of case-table arguments in an application. Encoded bodies are split if necessary, with the introduction of auxiliary combinators.

It should be stressed that the compiler is not generating circuitry for the FPGA. Rather, it is generating a representation of the program, suitable for execution on a template instantiation machine (Peyton Jones, 1987). In this case, the template instantiation machine, the Reduceron, has been realised on an FPGA.

3.3. The Reduceron Architecture

The Reduceron features broad memory channels to ‘widen the von Neumann bottleneck.’ (Naylor and Runciman, 2008). Many of the operations required to perform each graph reduction step are simultaneously performed in a single clock cycle.
3. A Supercompiler for the Reduceron

Listing 3.1: An example program, in our source language.

```haskell
{foldl f z xs = case xs of
    Nil -> z;
    Cons y ys -> foldl f (f z y) ys;
};

map f xs = case xs of
    Nil -> Nil;
    Cons y ys -> Cons (f y) (map f ys);

plus x y = (+) x y;
sum = foldl plus 0;
double x = (+) x x;
sumDouble xs = sum (map double xs);

range x y = case (<=) x y of
    True -> Cons x (range ((+) x 1) y);
    False -> Nil;

main = emitInt (sumDouble (range 0 10000)) 0;
}

```

Listing 3.2: `range` function after case elimination.

```haskell
range x y = (<=) x y [range#1,range#2] x y;
range#1 alts x y = Nil;
range#2 alts x y = Cons x (range ((+) x 1) y);
```

Instantiation of a function body takes $\lceil n/2 \rceil$ clock ticks, where $n$ is the number of applications in the body. Establishing the environment for function applications, updating the heap-graph to prevent repeated evaluations and applying primitive functions each take just one clock cycle. Dynamically maintained sharing information allows the Reduceron to avoid a high proportion of redundant updates where no sharing can occur. Constructor reductions (selection of the appropriate case alternative function from a case table) take place in zero clock cycles.

Listing 3.3 shows the evaluation of `range 0 10` to head normal form. Each reduction step is annotated with the operation that is being performed and how many cycles are required. This example takes four clock cycles, under the scheme outlined so far.
3.4. Primitive Redex Speculation

If primitive applications in a function body have fully evaluated arguments at instantiation time, the Reduceron can evaluate them speculatively during instantiation. Primitive redexes need not be constructed in memory, nor fetched again when needed. Even if the result of a primitive redex is not needed, reducing it is no more costly than constructing it.

Once again consider the reduction of the expression `range 0 10`, now with primitive redex speculation (PRS) enabled (Listing 3.4). One clock cycle is avoided for the comparison. Further clock cycles will be saved if the tail of the result is needed, as the addition to form the lower bound of the range has also been speculatively evaluated.

The beneficial effect of PRS is quite marked. For the example program in Listing 3.1, the Reduceron takes 230,029 clock cycles to execute without PRS. With PRS enabled the Reduceron only takes 150,024 clock cycles, a 35% reduction.

However, the number of PRS reductions in each instantiation is limited by a Reduceron design parameter. Currently, this limit is two.

---

**Listing 3.3: Reduction of `range 0 10` without PRS.**

```
range 0 10
= { Instantiate function body (1 cycle) }
  (<=) 0 10 [range#1,range#2] 0 10
= { Primitive application (1 cycle) }
  True [range#1,range#2] 0 10
= { Constructor reduction (0 cycle) }
  range#2 [range#1,range#2] 0 10
= { Instantiate function body (2 cycles) }
  Cons 0 (range ((+) 0 1) 10)
```

**Listing 3.4: Reduction of `range 0 10` with PRS.**

```
range 0 10
= { Instantiate function body (1 cycle) }
  True [range#1,range#2] 0 10
= { Constructor reduction (0 cycle) }
  range#2 [range#1,range#2] 0 10
= { Instantiate function body (2 cycles) }
  Cons 0 (range 1 10)
```
3. A Supercompiler for the Reduceron

3.5. Benchmark Programs

A selection of nineteen programs are used to test and benchmark the Reduceron platform. These programs range from small, toy examples that demonstrate specific effects, such as the sum-series, to significant computations like Knuthbendix.

These programs are described below in terms of their purpose, characteristics and code size. Line counts are for sources including all required auxiliary functions.

**Adjoxo**  An adjudicator for the game noughts and crosses, a.k.a. tic-tac-toe. The input is a game position, and the output is one of the three values — Win, Draw or Loss — indicating the outcome with best play for each of the players whose turn it might be. The method is the usual minimax recursive evaluation of completed game trees. (106 lines)

**Braun**  A Braun tree is a balanced binary tree offering an efficient yet simple implementation of flexible arrays. The program tests the property that converting a list to a Braun tree and back again is equivalent to the identity function. (51 lines)

**Cichelli**  Finds a perfect hash function for Haskell keywords. It uses a backtracking search to find an assignment of natural-number values to each letter that starts or ends a keyword such that hash values for keywords, computed as start-value + end-value + length, are unique and occupy a small integer range without gaps. (200 lines)

**Clausify**  Puts propositional formulae in clausal form using a multi-stage transformation of formula-trees. Almost a purely symbolic application, with hardly any arithmetic. (131 lines)

**Fib**  Computes the Nth number in the fibonacci sequence using a simple but naive doubly-recursive function definition. A purely arithmetic program involving no data structures at all. (10 lines)

**Knuthbendix**  The Knuth-Bendix completion method tries to derive a convergent term-rewriting system for a given equational theory and symbol-weighting scheme. It is a typical symbolic computing application from computer algebra. The example input used in the program gives group-theoretic axioms from which ten rewriting rules are derived. (533 lines)

**MSS**  Computes the maximum segment sum of a list of integers. Works by dividing the input list into all sub-lists, computing the sum of each, and returning the maximum. (47 lines)

**Mate**  Solves chess end-game problems of the form “P to move and mate in N”. The method is brute-force search in an explicit AND-OR game tree developing the given position to depth 2N – 1. Boards are represented by a square-piece association list for each player, where squares are coded as rank-file numeric pairs, so there is a fair amount of primitive arithmetic and comparison. (393 lines)
3.5. Benchmark Programs

**OrdList** Checks the property that insertion of a number into an ordered list of numbers results in a list that is still ordered. Numbers are represented as Peano numerals, so this is a purely symbolic program. *(46 lines)*

**Parts** Computes a celebrated number-theoretic function, the number of partitions of \( n \), where a partition is a bag of positive integers that sum to \( n \). There is a sophisticated closed formula for this number, but the method here is to list and count partitions explicitly. *(54 lines)*

**PermSort** Enumerates the permutations of a list of numbers, and returns the first ordered permutation. *(39 lines)*

**Queens** Solves a programming problem made famous by Wirth: place \( N \) queens on an \( N \times N \) chess board so that no two queens occupy a common rank, file or diagonal. The solution involves backtracking, list processing and an inner recursive loop that tests the safety of each candidate position for a new queen by primitive arithmetic comparisons with the coded positions of queens already in place. *(47 lines)*

**Queens2** A purely symbolic solution to the \( N \)-queens problem. Represents the board as a list of lists. Places a queen on one row at a time, maintaining a grid of threatened squares, and backtracks if a queen cannot be placed. *(62 lines)*

**Sudoku** A Sudoku solver due to Richard Bird [Bird 2006](#). Fills the blank cells on a Sudoku board with valid digits, pruning many possible choices that cannot possibly lead to a solution. *(209 lines)*

**Taut** A tautology checking program based on an example from Hutton’s book. The method is a brute-force evaluation for all possible boolean assignments to variables. *(95 lines)*

**While** A structural operational semantics of Nielson and Nielson’s While language [Nielson and Nielson 1992](#) applied to a program that computes the number of divisors of given integer. *(96 lines)*

**sumDouble** Computes \( \sum_{i=0}^{10000} 2i \) by generating the list of the numbers between 0 and 10,000, doubling each element and then computing the sum using the higher-order function, \( \text{foldl} \). Contains intermediate data structures and primitive operations. The program in [Listing 3.1](#) *(20 lines)*

**sumSquares** Computes \( \sum_{i=0}^{100} i^2 \). This is done in a similar fashion to the previous example. The square function consists of replicating its input \( n \), \( n \) times and summing the result using \( \text{foldl} \). Contains intermediate data structures, primitive operations and nested loops. *(23 lines)*

**sumSumEnum** Computes \( \sum_{i=0}^{100} \sum_{j=0}^{i} j \) by generating the list of the numbers between 0 and 100, mapping a function, \( \text{sumEnum} \), over the list and summing the resulting list. The \( \text{sumEnum} \) function sums the numbers between 0 and its input. Contains intermediate data structures, primitive operations and nested loops. *(22 lines)*

---

27
3. A Supercompiler for the Reduceron

3.6. Our Supercompiler Prototype

The basic Reduceron compiler currently performs very little optimisation. We are developing a supercompiler targeted at the Reduceron platform. Our supercompilation strategy is summarised in Figure 3.2. The starting point for the current prototype was a previous positive supercompiler for a core functional language by [Mitchell and Runciman (2008)].

Mitchell’s design inserted a supercompilation phase between core generation and compilation by the optimising Glasgow Haskell Compiler (GHC). In all but one of the published benchmarks, Mitchell’s supercompiler demonstrated at least equal and often significantly improved performance when compared with GHC alone. It is our hope that a supercompiler targeting the Reduceron can produce similar speedups.

3.6.1. Start

The supercompiler takes, as its input, the source program as an abstract syntax tree and the name of a *starting function*. This starting function is the point at which supercom-
### 3.6. Our Supercompiler Prototype

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free variable — $v$</td>
<td></td>
</tr>
<tr>
<td>Constructor — $c$</td>
<td></td>
</tr>
<tr>
<td>Integer — $n$</td>
<td></td>
</tr>
<tr>
<td>Application to free — $v$ $xs$</td>
<td></td>
</tr>
<tr>
<td>Primitive application — $f^p$ $xs$</td>
<td></td>
</tr>
<tr>
<td>Case over free variable — $\text{case } v \text{ of } ...$</td>
<td></td>
</tr>
<tr>
<td>Case over application to free — $\text{case } v$ $xs$ $\text{ of } ...$</td>
<td></td>
</tr>
<tr>
<td>Case over primitive — $\text{case } f^p$ $xs$ $\text{ of } ...$</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.3.: Expressions that cause simple termination

Supercompilation will begin. In the case of our benchmarks, we supply the name of an auxiliary function that *does not include the input data*, instead of `main`, to enforce separation and prevent bias in the results.

A residual function is produced from the function body, replacing the original with an application to the fresh residual definition. It is the fresh definition that is then processed. This is the same algorithm as the Tie Down procedure used later on in the pipeline (subsection 3.6.5).

#### 3.6.2. Termination

The algorithm terminates on a chain of derivations for one of three reasons:

1. Further supercompilation of the function would require the evaluation of an unknown or a primitive;
2. The current expression is similar to one that has been previously derived or;
3. The current expression simply contains no further inlinable applications.

### Simple Termination

This termination condition is triggered by expressions whose roots can no longer benefit from supercompilation. These expressions are characterised in Figure 3.3 and are detected using pattern matching.

### Homeomorphic Termination

Due to the nature of inlining function bodies in a recursive programs, a condition is required to ensure that loops are not repeatedly supercompiled. The homeomorphic termination condition detects structural similarities between the current expression and...
those that have been previously derived. Upon termination by this condition, driving stops and the expression is generalised with respect to the previous derivation.

The comparison is made using the homeomorphic embedding relation, $\preceq$. An expression, $x$, is homeomorphically embedded in expression $y$ if $y$ can be made into $x$ by replacing some subexpressions of $y$ with their own subexpressions. A formalised definition can be found in Figure 3.4. One should note that local variable names are ignored to ensure that renaming does not affect the relation.

**“No Inlines” Termination**

Simply put, stop driving when the supercompiler can no longer inline any function applications. See the subsection 3.6.3 for details of what constitutes a valid inlining. The simple termination condition, in combination with the simplification rules (section 3.6.3), accounts for a superset of these circumstances.

**3.6.3. Drive**

The drive phase of the algorithm inlines function bodies and simplifies the result. Driving continues until a termination criterion has been met.

**Inlining (Unfolding)**

We inline the first saturated non-primitive application that does not immediately trigger one of the termination criteria. If all inlines cause termination, the first function is inlined. Only applications at the root level and its immediate subexpressions are considered. The applications are considered in order of breadth-first abstract syntax tree traversal.
3.6. Our Supercompiler Prototype

Simplification

The expression is simplified after inlining to remove unnecessary code and propagate relevant information through the expression. The expression is simplified by applying a set of confluent transformations to the abstract syntax tree for as long as possible.

Many of the simplifications used during the drive phase (Figure 3.5) are well-known and long standing transformations (Peyton Jones and Santos, 1998) used in other optimising compilers. Others (Figure 3.6) have been either adapted for supercompilation (Mitchell, 2008) or for this particular language.

The simplifications are permitted to duplicate program code as long as computation is not duplicated. For example, the letInCase rule replicates let bindings (that are not referenced by the subject of the case) down case alternatives. This is permitted as only one of the case alternatives will be evaluated and, at most, only one set of the duplicate bindings will be evaluated.

3.6.4. Generalisation

If driving terminates due to a previous derivation, $x$, being homeomorphically embedded in the current expression, $y$, the supercompiler attempts to factor out the differences between the expressions.

Most Specific Generalisation

If the expressions are related by coupling, most specific generalisation (Sørensen et al., 1996) is used. A triple is formed of a fresh variable, $v^F$, a singleton set containing the current expression, $y$, bound to $v^F$ and another singleton set containing the embedded expression, $x$, bound to $v^F$. This triple is rewritten as far as possible using the rule in Figure 3.7.

Consider the resulting triple as $(v, bs_y, bs_x)$. As long as the bodies of the remaining bindings, $bs_y$, do not contain any free variables (due to references to local bindings), a generalised expression of the form

\[
\text{let } bs_y \text{ in } v
\]

is produced.

Lambdaless Form of Mitchell’s Generalisation

If the expressions are related by diving, Mitchell (2008) proposed generalisation by lifting out the shallowest coupled component of the embedding into a local definition.
(x xs) ys \Rightarrow x \ append(xs, ys)

case c x_1 \ldots x_n \ of \{ \ldots ; c \ v_1 \ldots v_n \rightarrow y ; \ldots \} \Rightarrow \let\{ v_1 = x_1 ; \ldots ; v_n = x_n \} \ in \ y

(c case x of \{ p_1 \rightarrow y_1 ; \ldots ; p_n \rightarrow y_n \}) ys \Rightarrow \case x of \{ p_1 \rightarrow y_1 ys ; \ldots ; p_n \rightarrow y_n ys \}

(let \{ v_1 = x_1 ; \ldots ; v_n = x_n \} \ in \ y) ys \Rightarrow \let\{ v_1 = x_1 ; \ldots ; v_n = x_n \} \ in \ y ys

(case (let \overline{b}s \ in \ y) of \overline{a}s) \Rightarrow \let\overline{b}s \ in (case y of \overline{a}s)

case (case x of \{ p_1 \rightarrow y_1 ; \ldots ; p_n \rightarrow y_n \}) of \overline{a}s \Rightarrow \case x of \{ p_1 \rightarrow \case y_1 of \overline{a}s \ldots p_n \rightarrow \case y_n of \overline{a}s \}

case v of \{ \ldots ; c \ overline{v}s \rightarrow y ; \ldots \} \Rightarrow \case v of \{ \ldots ; c \ overline{v}s \rightarrow y[v / c \ overline{v}s] ; \ldots \}

Figure 3.5.: Simplification laws for our source language.
\[
\begin{align*}
\text{let } & \{ v_1 = x_1, \ldots, v_i = x_i, \ldots, v_n = x_n \} \quad (\text{letInCase}) \\
\text{in } & \left( \text{case } y \text{ of } \{ \ldots; p_j \rightarrow y_j; \ldots \} \right) \\
\Rightarrow & \quad \text{let } \{ v_1 = x_1, \ldots, v_n = x_n \} \text{ in } \left( \text{case } y \text{ of } \{ \ldots; p_j \rightarrow \text{let } \{ v_i = v_j \} \text{ in } y_j; \ldots \} \right) \\
\text{where } & v_i \notin \text{free}(y) \cup \bigcup_j \text{free}(x_j). \\
\end{align*}
\]

\[
\begin{align*}
\text{let } & \{ v_1 = x_1, \ldots, v_i = x_i, \ldots, v_n = x_n \} \text{ in } y \quad (\text{inlineLet}) \\
\Rightarrow & \quad (\text{let } \{ v_1 = x_1, \ldots, v_n = x_n \} \text{ in } y)[v_i/x_i] \\
\text{where } & (v_i \notin \bigcup_j \text{free}(x_j) \text{ and only occurs once in free}(y)) \\
& \text{or } x_i \text{ is a variable or } x_i \text{ is an unsaturated function application.} \\
\end{align*}
\]

\[
\begin{align*}
\text{let } & \{ \ldots; v_i = c \; x_{i1} \ldots x_{im}; \ldots \} \text{ in } y \quad (\text{splitLet}) \\
\Rightarrow & \quad (\text{let } \{ ; \ldots \\
& \quad ; \; v_{i1}^F = x_{i1} \\
& \quad ; \ldots \\
& \quad ; \; v_{im}^F = x_{im} \\
& \quad ; \ldots \} \text{ in } y)[v_i/c \; v_{i1}^F \ldots v_{im}^F] \\
\text{where } & v_{i1}^F \ldots v_{im}^F \text{ are fresh variables.} \\
\end{align*}
\]

\[
\begin{align*}
\text{let } & \{ \} \text{ in } x \Rightarrow x \quad (\text{removeLet}) \\
\end{align*}
\]

\[
\begin{align*}
\text{let } & \overline{bs}_1 \text{ in } (\text{let } \overline{bs}_2 \text{ in } x) \quad (\text{letInLet}) \\
\Rightarrow & \quad \text{let } (\overline{bs}_1 \; ++ \; \overline{bs}_2^F) \text{ in } x \\
\text{where } & \overline{bs}_2^F \text{ is } \overline{bs}_2 \text{ where the bindings have been replaced with fresh variables.} \\
\end{align*}
\]

Figure 3.6.: New and augmented simplification transformations our source language.
3. A Supercompiler for the Reduceron

\[
\begin{pmatrix}
t_g \\
\{v^F = \sigma(x_1, \ldots, x_n) \cup \theta_x\} \\
\{v^F = \sigma(y_1, \ldots, y_n) \cup \theta_y\}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
t_g [v^F / \sigma(v_1^F, \ldots, v_n^F)] \\
\{v_1^F = x_1, \ldots, v_n^F = x_n \cup \theta_x\} \\
\{v_1^F = y_1, \ldots, v_n^F = y_n \cup \theta_y\}
\end{pmatrix}
\]

Figure 3.7.: Most specific generalisation rewrite rule, where \(v_1^F, \ldots, v_n^F\) are fresh variables and the \(\sigma(\text{sst})\) detonates an expression spine and its children.

Mitchell’s language included lambda abstractions, but ours does not. Therefore, the transformation needs to be adjusted to handle coupled components in our language.

One option would be to lift the coupled component to the top-level into a fresh definition, much like tying in subsection 3.6.5. However, this might lead to chains of functions of increasing arity, preventing termination of the supercompiler. Therefore, in the current prototype, this generalisation is disabled pending further investigation.

3.6.5. Tie

Regardless of the conditions that caused termination, any non-atomic child of the current expression is replaced with an equivalent application to either a freshly created or existing function definition.

**Tie Back (Folding)**

First, the trace of previously derived definitions is checked to see if a semantically equivalent function application can be constructed. If an appropriate definition is found, the child expression is replaced with an application of the existing function. Any non-atomic arguments are also tied.

**Tie Down (Residuating)**

If no existing definition matches the subexpression, a fresh residual definition is produced and the child expression is replaced with an application to the function. The residual definition is then, itself, driven, as described in subsection 3.6.3.

If the child expression is just the name of an original program definition (not one that has been produced through supercompilation), supercompilation is started (subsection 3.6.1) on the definition matching that name.
3.6.6. Clean Up

Supercompilation terminates when no more expressions can be tied down. The set of all final definitions that can be reached from the redefined main function forms the supercompiled residual program.

However, supercompilation may create inefficient artifacts in the code. The process of tying down creates unnecessary indirections that could harm program performance. Driving can create primitive reducible expressions. Post-processing is required to rectify these issues.

Final Inlining

Some function bodies are inlined to reduce the number of function applications. We traverse the dependency graph in a bottom-up manner, inlining any application to a non-recursive function.

When an application is unfolded, it is simplified using transformation rules similar to those described in section 3.6.3 but with constant folding transformations included.

Dead Definition Removal

The dependency graph is followed to ensure only reachable definitions remain in the final residual program.

3.7. A Synergistic Effect of Supercompilation

The supercompiler fuses away intermediate data structures. In its original form, the function sumDouble [Listing 3.1] maps double over its list input, only to apply foldl plus to the newly constructed list to calculate the sum. The supercompiler fuses this composition to a residual function that does not produce the intermediate list but performs the double operation as it sums. For both conventional implementations and the Reduceron, fewer reductions are needed to construct and deconstruct data structures.

Some effects of supercompilation particularly suit the features of the Reduceron architecture. For example, the program in [Listing 3.1] cannot, as it stands, benefit from PRS during the sum function because the primitive addition is not apparent in the body of the higher-order foldl. However, if supercompiled, foldl plus is specialised to a first-order equivalent. A considerable reduction in clock cycles is obtained because PRS now applies.

The original sumDouble program evaluates 20,003 expressions by PRS, compared with the 29,995 for the supercompiled program. No more primitive reductions were performed.
3. A Supercompiler for the Reduceron

Listing 3.5: A supercompiled form of *sumDouble*.

```haskell
sumDouble = sumDoubleAc 0;
sumDoubleAc z xs = xs [sumDoubleAc#1 , sumDoubleAc#2] z;
sumDoubleAc#1 y ys alts z = sumDoubleAc ((+) z ((+) y y)) ys;
sumDoubleAc#2 alts z = z;
```

overall than we performed originally. A further performance gain is achieved on top of the fusion effects. Following supercompilation, the Reduceron takes 159,970 clock cycles to execute the program in [Listing 3.1] without PRS and only 39,996 clock cycles with PRS. Compared with the original program executed without PRS, this is a 87% decrease in execution time.

Listing 3.5 shows the combined effects of supercompilation fusion and specialisation on *sumDouble*. In *sumDoubleAc#1*, the dependency of the outer addition on the inner one means that PRS requires an extra clock cycle to evaluate the expression fully. However, overall cycles are still saved in comparison with the reduction of a separate function application.

3.8. A Potentially Obstructing Effect of Supercompilation

There are circumstances where the process of supercompilation might impede PRS. Consider [Listing 3.6] an extract from the *Queens* example. The function *safe* computes whether it is ‘safe’ to place a queen in rank *x*, at a distance of *d* files away from the queens currently placed on the board. These queens are specified by their rank positions in the list *qs*.

[Listing 3.7] shows the original and supercompiled definitions following case elimination and inlining. Notice that in *safe*, all of the primitive reducible expressions are in one case alternative. But in the supercompiled version, *safeSC*, the expressions are split over separate case alternatives and will be evaluated during different instantiations. This leads to a situation where the execution of the original can speculatively evaluate a number of expressions simultaneously, whereas *safeSC* evaluates them separately.

3.9. Primitive Redex Speculation Candidate Lifting

To alleviate this issue, primitive expressions can be lifted as far as their variables are bound. The lifting process can take into account the maximum number of PRS reductions at instantiation and only lift to where there is spare capacity.

However, if we naively lift all primitive redex expressions, we may cause duplicate computation to occur. The supercompiler is permitted to duplicate *code* as long as it does
3.9. Primitive Redex Speculation Candidate Lifting

Listing 3.6: Original and supercompiled piece safety in the n-queens problem.

```haskell
and x y = case x of { True -> y; False -> False );
safe x d qs = case qs of {
    Nil -> True;
    Cons q l -> and ((/=) x q) (and ((/=) x ((+) q d)) (and ((/=) x ((-) q d)) (safe x ((+) d 1) l)));
};
safeSC x d qs = case qs of {
    Nil -> True;
    Cons q l -> case (/=) x q of {
        True -> case (/=) x ((+) q d) of {
            True -> case (/=) x ((-) q d) of {
                True -> safeSC x ((+) d 1) l;
                False -> False
            }
        False -> False
    }
};
```

Listing 3.7: Listing 3.6 after case elimination.

```haskell
and v0 v1 = v0 [and#1, and#2] v1;
and#1 v0 v1 = False;
and#2 v0 v1 = v1;
safe v0 v1 v2 = v2 [safe#1, safe#2] v0 v1;
safe#1 v0 v1 v2 v3 v4 = let {
    v5 = (+) v4 1;
    v6 = (/=) v3 ((-) v0 v4);
    v7 = (/=) v3 ((+) v0 v4);
    v8 = (/=) v3 v0
} in v8 [and#1, and#2] (v7 [and#1, and#2] (v6 [and#1, and#2] (v1 [safe#1, safe#2] v3 v5)));
safe#2 v0 v1 v2 = True;
safeSC v0 v1 v2 = v2 [safeSC#7, safeSC#8] v0 v1;
safeSC#1 v0 v1 v2 v3 = False;
safeSC#2 v0 v1 v2 v3 = let { v4 = (+) v2 1 } in v3 [safeSC#7, safeSC#8] v1 v4;
safeSC#3 v0 v1 v2 v3 v4 = False;
safeSC#4 v0 v1 v2 v3 v4 = (/=) v1 ((-) v2 v3) [safeSC#1, safeSC#2] v1 v3 v4;
safeSC#5 v0 v1 v2 v3 v4 = False;
safeSC#6 v0 v1 v2 v3 v4 = (/=) v1 ((+) v2 v3) [safeSC#3, safeSC#4] v1 v2 v3 v4;
safeSC#7 v0 v1 v2 v3 v4 = (/=) v3 v0 [safeSC#5, safeSC#6] v3 v0 v4 v1;
safeSC#8 v0 v1 v2 = True;
```
not duplicate computation, under lazy evaluation. For example, our supercompiler may replicate bindings from outside a case expression down each case alternative. As only one alternative is evaluated, only one of the duplicate bindings will be evaluated under both lazy and speculative evaluation.

If replicated primitive redexes are lifted above a case distinction, they may be evaluated speculatively, taking away capacity that other primitive expressions could have used. We detect these replicated expressions and merge them into a single binding.

The rules used to lift primitive expressions are detailed in Figure 3.9. First the extract rule is repeatedly applied to put any primitive expressions into let bindings. Then, the lift rule is repeatedly applied to lift safe primitive expressions to spare capacity. The prxs x function returns a list of tuples. The tuple consists of: (a) a primitive binding inside the expression x; and (b) the expression x but with that binding removed. The tuples are ordered by the number of case statements are passed through to reach the primitive.

Our original supercompiler increased the execution time of PRS-enabled results for Queens by 16%. With the primitive redex lifting strategy applied, supercompilation decreases execution time by 33%.

3.10. Performance Results

Each example described in section 3.5 is compiled with six variations of the compilation pipeline. These are: a) normal compilation, b) normal compilation with PRS, c) supercompilation and normal compilation, d) supercompilation and normal compilation with PRS, e) supercompilation, primitive redex lift and normal compilation, f) supercompilation, primitive redex lift and normal compilation with PRS.

The compiled output is executed on a Reduceron simulator. The simulator returns various profiling measurements such as total clock cycles, the number of PRS evaluated expressions and the proportion of time spent on individual functions and reduction operations. Table 3.1 presents the performance of our test programs relative to that of pipeline a, the original program executed without PRS.

PRS Only — Primitive redex speculation alone (pipeline b) increases average performance by a factor of 1.27x. All but four of the examples achieve some improvement. The
3.10. Performance Results

Tree traversal is explicit.

\[
\begin{align*}
&\text{let } \{ \ldots ; v = x ; \ldots \} \text{ in } y \\
&\quad \xRightarrow{\text{extract}} \\
&\quad \text{let } \{ \ldots ; v = [\text{extract }' x]; \ldots \} \text{ in } [\text{extract } y] \\
&\quad \xRightarrow{\text{prsx}} p \\
&\quad \text{let } \{ v^F = p \} \text{ in } v^F \\
&\quad \xRightarrow{\sigma \, \text{map } \text{extract } \overline{y}} \\
&\quad \text{let } \overline{bs} \text{ in } x \\
&\quad \xRightarrow{\text{lift}} \\
&\quad \text{let } \{ prxs_b ; \overline{bs} \} \text{ in } prxs_x
\end{align*}
\]

where \( \overline{bs} \) contains less than two \( p \) expressions and 
\( (prxs_b, prxs_x) = (\text{head } \cdot \text{filter } (\text{noFreeWrt } x) \cdot prxs) \) \( x \).

\[
\begin{align*}
&\text{case } x \text{ of } \{ \ldots ; p \rightarrow y ; \ldots \} \\
&\quad \xRightarrow{(\text{lift}, \text{lift }')}
\end{align*}
\]

\[
\begin{align*}
&\text{case } [\text{lift } x] \text{ of } \{ \ldots ; p \rightarrow [\text{lift } x]; \ldots \} \\
&\quad \xRightarrow{\sigma \, \text{map } \text{lift }' \overline{ys}} \\
&\quad \text{let } \{ \ldots ; v = p ; \ldots \} \text{ in } y \\
&\quad \xRightarrow{\text{prsx}} \\
&\quad (v = p, \text{ let } \{ \ldots \} \text{ in } y)
\end{align*}
\]

Figure 3.9.: Primitive redex lifting rules.

39
3. A Supercompiler for the Reduceron

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Supercompiled</th>
<th>SC + PRS lift</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(a)</td>
<td>(b)</td>
<td>(c)</td>
</tr>
<tr>
<td>(\text{adjxo})</td>
<td>1.000</td>
<td>0.799</td>
<td>0.866</td>
</tr>
<tr>
<td>(\text{braun})</td>
<td>1.000</td>
<td>1.000</td>
<td>\textbf{0.769}</td>
</tr>
<tr>
<td>(\text{chichelli})</td>
<td>1.000</td>
<td>\textbf{0.990}</td>
<td>1.000</td>
</tr>
<tr>
<td>(\text{clausify})</td>
<td>1.000</td>
<td>0.445</td>
<td>1.000</td>
</tr>
<tr>
<td>(\text{knuthbendix})</td>
<td>1.000</td>
<td>0.900</td>
<td>0.896</td>
</tr>
<tr>
<td>(\text{MSS})</td>
<td>1.000</td>
<td>0.864</td>
<td>0.995</td>
</tr>
<tr>
<td>(\text{mate})</td>
<td>1.000</td>
<td>0.867</td>
<td>0.912</td>
</tr>
<tr>
<td>(\text{ordlist})</td>
<td>1.000</td>
<td>1.000</td>
<td>\textbf{0.662}</td>
</tr>
<tr>
<td>(\text{parts})</td>
<td>1.000</td>
<td>0.746</td>
<td>0.933</td>
</tr>
<tr>
<td>(\text{permsort})</td>
<td>1.000</td>
<td>0.962</td>
<td>0.861</td>
</tr>
<tr>
<td>(\text{ queens})</td>
<td>1.000</td>
<td>0.421</td>
<td>0.850</td>
</tr>
<tr>
<td>(\text{queens2})</td>
<td>1.000</td>
<td>0.996</td>
<td>0.989</td>
</tr>
<tr>
<td>(\text{sudoku})</td>
<td>1.000</td>
<td>0.936</td>
<td>0.955</td>
</tr>
<tr>
<td>(\text{taut})</td>
<td>1.000</td>
<td>1.004</td>
<td>\textbf{0.700}</td>
</tr>
<tr>
<td>(\text{while})</td>
<td>1.000</td>
<td>0.947</td>
<td>0.996</td>
</tr>
<tr>
<td>(\text{sumdouble})</td>
<td>1.000</td>
<td>0.652</td>
<td>0.695</td>
</tr>
<tr>
<td>(\text{sumsquares})</td>
<td>1.000</td>
<td>0.541</td>
<td>0.726</td>
</tr>
<tr>
<td>(\text{sumsumenum})</td>
<td>1.000</td>
<td>0.481</td>
<td>0.847</td>
</tr>
<tr>
<td>(\text{geometric mean})</td>
<td>1.000</td>
<td>0.788</td>
<td>0.871</td>
</tr>
</tbody>
</table>

|                  | \(\text{Geometric Mean}\) | 0.881         | \textbf{0.598} |

Table 3.1.: Execution time as multiples of that for pipeline \(a\), the program before supercompilation executed without PRS. (Best results are in bold.)

only example that suffers under PRS is \text{taut}, due to primitive redexes with dependencies being formed by the inliner. These cause the PRS process to take extra clock cycles for expressions that are never referenced. However, the drop in performance is only very slight.

Supercompilation Only — Supercompilation alone (pipeline \(c\)) boosts performance by a factor of 1.15x. Only three examples do not benefit from supercompilation. While we do not expect much improvement on \text{fib} due to its simple structure, the highly symbolic programs \text{chichelli} and \text{clausify} would be expected to benefit from fusion. Preliminary analysis of the residual programs seems to indicate that some functions are not being supercompiled within the contexts of their callers. This would prevent fusion from occurring.
### 3.10. Performance Results

<table>
<thead>
<tr>
<th>Relative Primitives by PRS</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Size</strong></td>
<td><strong>Cycles</strong></td>
<td><strong>Cases</strong></td>
</tr>
<tr>
<td><strong>Adjoxo</strong></td>
<td>1.500</td>
<td>0.507</td>
</tr>
<tr>
<td><strong>Braun</strong></td>
<td>2.526</td>
<td>0.769</td>
</tr>
<tr>
<td><strong>Chichelli</strong></td>
<td>1.494</td>
<td>1.008</td>
</tr>
<tr>
<td><strong>Clausify</strong></td>
<td>5.879</td>
<td>0.951</td>
</tr>
<tr>
<td><strong>Fib</strong></td>
<td>1.588</td>
<td>0.792</td>
</tr>
<tr>
<td><strong>KnuthBendix</strong></td>
<td>2.508</td>
<td>0.866</td>
</tr>
<tr>
<td><strong>MSS</strong></td>
<td>1.920</td>
<td>0.996</td>
</tr>
<tr>
<td><strong>Mate</strong></td>
<td>2.423</td>
<td>0.954</td>
</tr>
<tr>
<td><strong>OrdList</strong></td>
<td>3.887</td>
<td>0.678</td>
</tr>
<tr>
<td><strong>Parts</strong></td>
<td>1.733</td>
<td>1.009</td>
</tr>
<tr>
<td><strong>PermSort</strong></td>
<td>2.000</td>
<td>0.756</td>
</tr>
<tr>
<td><strong>Queens</strong></td>
<td>2.535</td>
<td>0.773</td>
</tr>
<tr>
<td><strong>Queens2</strong></td>
<td>2.068</td>
<td>0.966</td>
</tr>
<tr>
<td><strong>Sudoku</strong></td>
<td>1.793</td>
<td>0.870</td>
</tr>
<tr>
<td><strong>Taut</strong></td>
<td>2.057</td>
<td>0.856</td>
</tr>
<tr>
<td><strong>While</strong></td>
<td>3.030</td>
<td>1.062</td>
</tr>
<tr>
<td><strong>sumDouble</strong></td>
<td>0.450</td>
<td>0.200</td>
</tr>
<tr>
<td><strong>sumSquares</strong></td>
<td>0.900</td>
<td>0.378</td>
</tr>
<tr>
<td><strong>sumSumEnum</strong></td>
<td>1.435</td>
<td>0.943</td>
</tr>
</tbody>
</table>

| Geometric Mean | 1.936 | 0.759 | 0.790 | 7.3% | 21.7% |

Table 3.2.: Various metrics for the benchmark programs. Relative values are against the original program executed with PRS (pipeline $\mathcal{B}$).

**Supercompilation and PRS** — The combination of PRS and supercompilation (pipeline $\mathcal{D}$) largely gives results as expected (PRS factor $\times$ supercompilation factor) with a few notable exceptions. **Adjoxo**, **Parts**, **sumDouble** and **sumSquares** all show better than expected performance, mainly for the reasons described in section 3.7. However, **Queens** and **sumSumEnum** show considerably worse than expected performance, due to primitive expressions being separated over different case alternatives, as discussed in section 3.8.

**Supercompilation, Lifting and PRS** — The results of PRS, supercompilation and primitive redex lifting (pipeline $\mathcal{F}$) indicate that this strategy is effective. Every program except for **While** performs better than the original program without PRS. However, **Taut** performs significantly worse than under our original supercompiler strategy. We conjecture that this is due to imprecise detection of possible primitive redexes (Figure 3.8). Across all our test programs, this strategy boosts performance by an average factor of 1.67x.
3. A Supercompiler for the Reduceron

Table 3.2 gives another view of the impact of the supercompiler. It compares the number of Reduceron combinators produced (size), time taken to execute (cycles), number of case tables evaluated (cases) and the proportion of primitive operations performed by PRS. These are recorded for pipeline $\text{f}$ and shown as multiples of the results for pipeline $\text{b}$.

As would be expected, supercompilation can greatly increase the size of the compiled program. There does not seem to be a relationship between relative execution time performance and relative code size. A reduction in the number of cases tables evaluated indicates that fusion has taken place, as fewer data structures are being consumed.

In three cases, pipeline $\text{f}$ still produces programs that perform worse than under PRS alone (pipeline $\text{b}$). In comparison to the gains made by other programs, these are only very small performance loses. Both for Chichelli and for While, the proportion of primitive operations performed by PRS has actually fallen. It is currently unclear why Parts performs worse when it has a large increase in the number of PRS candidates and a small amount of fusion.

Despite these results, the current prototype of our supercompiler gives a geometric mean speed-up factor of 1.32x for programs executed under PRS.

3.11. Conclusions and Future Work

The sumDouble example was chosen to demonstrate the benefits of both PRS and supercompilation. However, we did not see the full magnitude of the combined effect of the two technologies.

Other examples, such as Queens, did not benefit from supercompilation. This led to the development of the primitive redex lifting strategy that has largely permitted these examples to benefit from the same effects as sumDouble. This strategy does not seem to produce benefits for all programs. Further investigation is required to discover why some results still do not improve and a small number get worse.

I hypothesise that the weak point in the primitive redex lifting procedure is the selection of primitive expressions to be lifted. A better static analysis is currently in development to predict guaranteed primitive redex expressions at instantiation time.

Still, based on the evidence detailed in this paper, it would appear that PRS and supercompilation can be synergistic, once certain primitive redexes are relocated to maximise design constraints.

There is further scope to exploit the Reduceron design characteristics with the supercompiler. A final inlining phase is required after supercompilation to reduce instantiations at run-time. The current method for selecting candidates for inlining is simplistic. Further performance gains can be made by an improved inlining strategy that considers the constraints on function bodies imposed by the design parameters of the Reduceron.
Future designs of the Reduceron will also permit even more primitive redexes to be speculatively evaluated in parallel. This will likely enable even further performance gains from supercompilation targeted at the Reduceron platform.
4. Mechanical Theorem Proving

This chapter summarises some of the experiences I have had with theorem provers and software verification. It represents investigations that are at a very preliminary stage.

4.1. Introduction

The literature survey suggests that to prove properties about a compiler of any reasonable functionality, proof assistants and mechanisation are essential. The proof not only benefits from automated checking but the process can be simplified through macro and tactical automation.

A large number of mechanised theorem proving environments and logics are available. In this report, I shall three commonly used proof assistants that are implemented as functional languages. This is because the current supercompiler prototype is is written in Haskell.

4.2. HOL Light

HOL Light is an interactive theorem prover and a member of the LCF/HOL family. It is implemented in Objective Caml with a focus 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.

HOL Light utilises two languages. The host, O’Caml, is used to instruct the theorem prover. Another language, is embedded into the top-level to specify the constructs that are to be reasoned about.

In keeping with the lightweight philosophy of the system, there are few comforts for the user. The theorem prover is generally interfaced through the interactive O’Caml interpreter, where the system is initialised by loading the HOL Light library into the top-level. This process takes a number of minutes. As there is no mechanism to remove introduced definitions, this needs to be repeated whenever a valid, but incorrect, HOL term is defined or a library file has been edited.

Proof scripts take the form of O’Caml, functional programs. These are executed to invoke the proof commands. O’Caml errors are returned for invalid operations and definitions.
4.3. Isabelle/HOL

As a result, proof scripts are quite verbose and do not always follow a natural description of proof steps.

The functional programming aspect to instructing HOL Light means that, in many cases, definitions must be passed into operations explicitly. This is not necessarily a disadvantage, as it makes it very clear to the reader which laws have been applied by which stages of the proof. It is a design choice that is not necessarily taken by other systems.

The tutorial document 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 \texttt{REWRITE_TAC} and \texttt{SIMP_TAC} beyond that \texttt{SIMP_TAC} is “more powerful ... /and/ will exploit contextual information.” (Harrison, 2007)

As a result of this and the obstructive nature of the environment, I have been finding it a little difficult to move beyond the tutorial examples. We do, however, have an experienced user of the system within the department and, with perseverance, I am sure that these issues can be overcome.

An implementation of the McCarthy and Painter (1967) proof, written by John Harrison, is included in the HOL Light distribution and reproduced in Appendix B. It is hoped that the study of this example will improve my understanding of how to apply mechanical theorem proving.

4.3. Isabelle/HOL

Isabelle is “a generic proof assistant” (Isabelle) written in ML. 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.

The Isabelle system provides the Isar language for directing the proof assistant. Isar aims “for proof text naturally understandable for both humans and computers.” (Isabelle) 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. Large amounts of documentation, tutorial and reference material exist, leading to an opposite problem of HOL Light. It is difficult to know which source is best for the information required. Fortunately, I have been put in contact with two heavy Isabelle users who are willing to guide me through any difficulties.

One interesting difference in design from HOL Light is that definitions, once defined, are stored to the environment to be implicitly invoked by the proof commands. For example, where in HOL Light one would invoke the induction tactic with a variable containing
the inductive properties of the datatype, Isabelle/HOL stores these to the environment and selects them automatically. This reduces the verbosity of the proof scripts. It is not clear whether proof scripts are easier or more difficult to read as a result.

I have been attempting to port a HOL Light implementation of the McCarthy and Painter (1967) proof to Isabelle/HOL. My limited knowledge of what some of the HOL Light constructs do and difficulty in finding the correct Isabelle/HOL equivalents is my current obstacle. The incomplete proof can be found in Appendix C.

### 4.3.1. Haskabelle

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 only appears to work on a subset of Haskell 98 programs (which is to be expected) and is targeted at proving partial correctness.

One issue is that the exact subset of Haskell 98 that is accepted is not explicitly defined. Even when the tool is run on a program that is beyond its capabilities, no errors or warnings are displayed. The definitions are merely missing in the resulting specifications. Furthermore, only a subset of the Haskell prelude has been translated, resulting in missing definitions even when only the appropriate language features have been used.

For example, I have processed a Haskell encoding of the McCarthy and Painter (1967) compiler example through the Haskabelle tool. Due to the simplicity of the data structures and constructs used, nearly all of the code that I wrote was reproduced in the final specification. However, some of the Prelude functions, some as basic as string equality, were not available.

Despite these misgivings, the tool has been useful, at the very least, as an educational tool to observe how to translate Haskell into Isabelle specifications by hand. It may even be useful for removing some of the more tedious tasks in the porting process. The project is still in its infancy. Future versions may mitigate these concerns.

### 4.3.2. Code Extraction

Isabelle includes a code extraction feature, by which Haskell and ML programs can be generated from Isabelle proofs. I have not yet experimented with this functionality myself but the literature (Haftmann, 2010) suggests it is a relatively straightforward procedure.

### 4.4. 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
4.5. Summary

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. It is implemented in O’Caml and provides a number of user interfaces. Experience has shown that the included GUI, CoqIDE, is highly unstable at the present time. The interface through the generic Proof General, on the other hand, provides nearly all of the functionality exposed by CoqIDE and rarely has issues.

Despite the underlying difference in theory, I have found the specification language in Coq to be fairly intuitive. One interesting feature is that inductive and co-inductive types (data and co-data) are explicitly separate objects, as are fixpoints and co-fixpoints (recursion and co-recursion). These may provide useful abstractions for reasoning about program termination properties.

The Coq community has a similar level of documentation, tutorial and reference material to the Isabelle/HOL community. While I do not have any specific contacts to refer to, I have found the #coq channel on Freenode to be generally quite helpful and welcoming.

Once again, taking McCarthy and Painter (1967) example, my implementation in Coq is still unfinished. However, I am closer to completing the Coq proof than to completing the Isabelle/HOL proof and understanding the HOL Light proof. The incomplete proof can be found in Appendix D.

4.4.1. Code Extraction

Similar to Isabelle/HOL, Coq can extract executable code from theorems. A number of target functional languages are supported, currently Objective Caml, Haskell and Scheme.

There does not appear to be a Coq equivalent to Haskabelle.

4.5. Summary

My knowledge and experience of theorem proving is still at a very early stage. The three systems discussed in this chapter are only a sample of the multitude of options available. With limited time, it may soon be necessary to focus my attention on one system.

Before I do, I would like to investigate the capabilities of Isabelle and Coq’s code extraction features, which would likely be useful in producing a verified compiler object code. I would also like to understand some of the theory behind (co-)data, (co-)recursion and (co-)induction to see if it relates to concepts such as the proof of driving termination under the homeomorphic relation.

Once I have a better grasp of these concepts and capabilities, I should be able to make an informed decision about which tool is best for my needs.
Part IV.

Research Proposal
5. Research Proposal

In this chapter, I lay out my immediate and long-term research goals. I detail the overall aims of my work and present a strategy for achieving them, along with the current progress through this strategy. To finish, in reference to this strategy, I plan my time for the next six months.

5.1. Overall Goals

The goal of my research is to implement a verified supercompiler which will become the optimising component in a verified stack for the core functional language and the Reduceron machine.

I hypothesise that the modular structure of the supercompiler, composed of many source-to-source transformations, will make mechanized theorem proving an appropriate technique for verifying its correctness. This hypothesis is supported by success in the mechanized verification of similar compilation technologies.

Within the next six months I aim to have an extensively tested and simplified prototype supercompiler. A subset of the correctness properties of the prototype should be verified.

5.2. Specific Aims

For my research, I aim to investigate a number of the topics surrounding supercompilation and optimising programs for the Reduceron, in a verified manner.

**Aim 1** Produce a supercompiler that is:

(a) Experimentally, an effective optimiser for Reduceron

(b) Of a simple design that is amenable to formal verification.

**Aim 2** Mechanically verify that the supercompiler must:

(a) Terminate for any well-formed input program.

(b) Maintain semantic equivalence.

**Aim 3** Classify the program constructs that benefit or suffer from the supercompilation under the Reduceron execution model.
5. Research Proposal

5.3. Strategy

5.3.1. Phase A — Verification Tools

<table>
<thead>
<tr>
<th>Dependencies: None</th>
<th>Related Aims: {2}</th>
<th>Aims Completed: None</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Estimate: 3 months</td>
<td>Progress: 1 month</td>
<td></td>
</tr>
</tbody>
</table>

A number of logics, theorem provers and proof assistants are available. Many are already being used to verify similar properties to those that my research is focused on. I feel I need to explore a number of the popular theorem proving environments that are available. While initially working off tutorials, I would likely make a better assessment by attempt to prove for myself some known properties of compilers.

In addition to testing the tools, I will investigate the communities surrounding the tools to discover contacts for support and information during my work.

By the end of Phase A, I should have selected a logic and theorem proving environment to complete my work and have some representative proofs in this system.

5.3.2. Phase B — A Supercompiler for the Reduceron

<table>
<thead>
<tr>
<th>Dependencies: None</th>
<th>Related Aims: {1,2,3}</th>
<th>Aims Completed: {1a}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Estimate: 6 months</td>
<td>Progress: 4 months</td>
<td></td>
</tr>
</tbody>
</table>

There are many open questions remaining in the topic of supercompilation relating to the specific design of the supercompiler. This include determining which termination conditions should be used and which simplifications should be applied.

In addition to solving these questions, there are challenges to implementing a supercompiler for a new language and a new execution architecture. The absence or inclusion of language or execution features may alter the results of existing designs.

Phase B produces a supercompiler that operates on Reduceron programs and provides speed-ups under the execution conditions.

5.3.3. Phase C — Simplifying and Restructuring the Supercompiler

<table>
<thead>
<tr>
<th>Dependencies: {A,B}</th>
<th>Related Aims: {1,2}</th>
<th>Aims Completed: {1b}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Estimate: 2 months</td>
<td>Progress: None</td>
<td></td>
</tr>
</tbody>
</table>

While Phase B will produce an empirically working supercompiler, for Phase D to be successful, the software will need to be structured in such a way that it is amenable to formal reasoning.

Drawing on the experience gathered in Phase A, the supercompiler may need to be reorganised and, very likely, simplified to make the verification process a tractable task.
5.3. Strategy

The result of Phase C is a piece of software that is in a form such that correctness properties can be inferred.

5.3.4. Phase D — Verifying the Supercompiler

<table>
<thead>
<tr>
<th>Dependencies: {A,C}</th>
<th>Related Aims: {1,2,3}</th>
<th>Aims Completed: {2}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Estimate: 12 months</td>
<td>Progress: None</td>
<td></td>
</tr>
</tbody>
</table>

With the knowledge gained from Phase A and the supercompiler implementations from Phases B and C, Phase D aims to attain the main goal of the research.

Experience from the preceding phases will largely direct how this phase will be organised. Based on the current prototype (chapter 3) and my present competency in verification (chapter 4), I can foresee the problem being decomposed as follows:

1. Semantic equivalence of most specific generalisation.
2.Termination of most specific generalisation.
4. Termination of simplification transformations.
5. Semantic equivalence of inlining.
7. Semantic equivalence of tying back.
8. Semantic equivalence of the supercompiler.
9. Termination of the supercompiler.

Through the course of these proofs, some shared lemmas may be discovered. Furthermore, a number of these proofs will depend on the successful verification of others.

Phase D will result in the mechanically verified artifact and the documented experience from the verification process.

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

<table>
<thead>
<tr>
<th>Dependencies: {B,C,D}</th>
<th>Related Aims: {1,3}</th>
<th>Aims Completed: {3}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Estimate: 4 months</td>
<td>Progress: 0.5 months</td>
<td></td>
</tr>
</tbody>
</table>

The goal of any optimising transformation is to make programs run more efficiently in terms of execution time or any other run-time resource. The supercompiler in development for this research is being targetted at a specific platform.

Therefore, given that we have a precise model for the execution of Reduceron programs, I intend to map out the program classes that benefit or suffer from this form of supercompilation. This should help create new supercompiler designs that compensate for any current deficiencies.

Phase E is currently an open-ended question but it should result in a supercompiler that optimises programs so that they take advantage of Reduceron-specific characteristics.
5. Research Proposal

Definitions should also be formed relating to the types of programs that benefit from the process.

5.4. Current Progress

What follows is a summary of the relevant phases. Detailed discussion of the current progress can be found in chapters 3 and 4.

Verification Tools  Investigations have begun into a variety of theorem provers and logics. The main proof-tools used, so far, have been Coq, HOL Light and Isabelle/HOL as well as the theorem extraction tool Haskabelle. I have been attempting to improve my competency through following tutorials, contacting members of the relevant communities and proving known theorems, such as the McCarthy and Painter (1967) example.

A Supercompiler for the Reduceron  The supercompiler for the Reduceron has been in development for some time and is now producing some interesting results. As it stands, the supercompiler, on average, improves program performance 1.32x. Still, a number of the benchmark programs are not performing quite as expected and further investigation is required.

Classifying Behaviour of the Supercompiler and the Reduceron  Through the work of Phase B, a number of links between the supercompiler and other Reduceron constructs are being discovered. For example, the synergistic effect of supercompilation and PRS. However, there are other parts of the Reduceron’s compilation and execution that are not yet being exploited.

5.5. Immediate plan

Over the next 6 months I intend to;

- 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. This will resulting in an empirically dependable program optimiser, improving on the performance results presented in chapter 3 by eliminating the regressing results. (2.5 months.)
5.5. Immediate plan

- 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 a number of simplified supercompilation components. (1.5 months.)
Bibliography


John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. Mathematical Aspects of Computer Science, 19:33–41, 1967. URL [http://books.google.com/books?hl=en&amp;lrr&amp;id=ynigSICjflYC&amp;oi=fnd&amp;pg=PA33&amp;dq=Correctness+of+a+compiler+for+arithmetic+expressions+%E2%88%97&amp;ots=i0LKhYeItf&amp;sig=dYfxCQ6npSU5nW_SUyMTOY72xAO](http://books.google.com/books?hl=en&lrr&id=ynigSICjflYC&oi=fnd&pg=PA33&dq=Correctness+of+a+compiler+for+arithmetic+expressions+%E2%88%97&ots=i0LKhYeItf&sig=dYfxCQ6npSU5nW_SUyMTOY72xAO).


Bibliography


Bibliography


A. Haskell Encoding of the McCarthy-Painter Proof

Adapted from the proof in [McCarthy and Painter (1967)](#).

```haskell
{- Basic data types -}
type Value = Int
data Address = Ac | Reg Int | Map Name
  deriving (Show, Eq)
type Name = String

{- Source AST -}
data Source = Const Value | Var Name | Add Source Source
  deriving Show

{- Target AST -}
data TargetInst = Li Value | Load Address | Sto Address | Sum Address
  deriving Show

type Target = [ TargetInst ]

{- Abstract and Concrete State -}
type Abstract = (Name -> Value)
type Concrete = (Address -> Value)
unbound x = error ("No mapping for name/address " ++ x ++ "."")

construct :: Abstract -> Concrete
construct s (Map v) = s v
construct s x = unbound (show x)

{- Memory write -}
write :: Address -> Value -> Concrete -> Concrete
write x n s = \y -> if y == x then n else s y

{- Source semantics -}
source :: Source -> Abstract -> Value
```

58
source (Const n) = n
source (Var v) = s = s v
source (Add x y) = s = source x s + source y s

{- Target semantics -}
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 r) s
step (Sto r) s = write r (s Ac) s
step (Sum r) s = write Ac (s Ac + s r) s

{- Compiler -}
compile :: Int -> Source -> Target
compile t (Const n) = [Li n]
compile t (Var v) = [Load (Map v)]
compile t (Add x y) = compile t x
++ [Sto (Reg t)]
++ compile (t + 1) y
++ [Sum (Reg t)]

Untouched Registers LEMMA:
\[ r < t \Rightarrow (\text{target} (\text{compile} t x) s) (\text{Reg} r) = = = s (\text{Reg} r) \]

Untouched Variables LEMMA:
\[ (\text{target} (\text{compile} t x) s) (\text{Map} v) = = = s (\text{Map} v) \]

{- Proof -}
sourceP :: Int -> Source -> Abstract -> Value
sourceP t (Const n) s
-- = target (compile t (Const n)) (construct s) Ac
-- = target [Li n] (construct s) Ac
-- = write Ac n (construct s) Ac
-- = if Ac == Ac then n else construct s

= n

sourceP t (Var v) s
-- = target (compile t (Var v)) (construct s) Ac
-- = target [Load (Map v)] (construct s) Ac
A. Haskell Encoding of the McCarthy-Painter Proof

```
{- Unfold 'target' and 'foldl' -}
\[= \text{step} (\text{Load} (\text{Map} v)) (\text{construct} s) \text{Ac} \]
{- Unfold 'step' -}
\[= \text{write} \text{Ac} ((\text{construct} s) (\text{Map} v)) (\text{construct} s) \text{Ac} \]
{- Unfold 'write' and then apply -}
\[= \text{if} \text{Ac} == \text{Ac} \text{then construct} s \text{(Map} v) \text{else construct} s \text{Ac} \]
{- Unfold 'if' and 'construct' -}
\[= s v \]
```

```
s\_sourceP ~ t \_ (\_Add x y) \_ v
\[= \text{target} (\text{compile} \_ t \_ \_ (\_ Add x y)) (\text{construct} s) \text{Ac} \]
{- Unfold 'compile' -}
\[= \text{target} (\text{compile} \_ t \_ \_ x \_ ++ [\text{Sto} (\_ \text{Reg} t)]) \]
\[++ \text{compile} (t + 1) y \_ ++ [\text{Sum} (\_ \text{Reg} t)]) \]
\[= (\text{construct} s) \text{Ac} \]
{- Unfold 'target' and 'step' -}
\[= \text{let} \]
\[s1 = \text{target} (\text{compile} \_ t \_ x) (\text{construct} s) \]
\[s2 = \text{write} (\_ \text{Reg} t) (s1 \_ \text{Ac}) s1 \]
\[s3 = \text{target} (\text{compile} (t + 1) y) s2 \_ \text{in} \]
\[\text{write} \text{Ac} (s3 \_ \text{Ac} + s3 (\_ \text{Reg} t)) \_ \text{Ac} \]
{- Unfold 'write' and 'if' -}
\[= \text{let} \]
\[s1 = \text{target} (\text{compile} \_ t \_ x) (\text{construct} s) \]
\[s2 = \text{write} (\_ \text{Reg} t) (s1 \_ \text{Ac}) s1 \]
\[s3 = \text{target} (\text{compile} (t + 1) y) s2 \_ \text{in} \]
\[s3 \_ \text{Ac} + s3 (\_ \text{Reg} t) \]
{- s2 and s3 equivalent for \text{Reg} t -}
{- s2 is equivalent to (\text{construct} s) for compiled code -}
\[= \text{let} \]
\[s1 = \text{target} (\text{compile} \_ t \_ x) (\text{construct} s) \]
\[s2 = \text{write} (\_ \text{Reg} t) (s1 \_ \text{Ac}) s1 \]
\[\_ \text{in} \]
\[\text{target} (\text{compile} (t + 1) y) (\text{construct} s) \text{Ac} + s2 (\_ \text{Reg} t) \]
{- Apply and unfold 'write' -}
\[= \text{target} (\text{compile} (t + 1) y) (\text{construct} s) \text{Ac} \]
\[+ \text{target} (\text{compile} \_ t \_ x) (\text{construct} s) \text{Ac} \]
{- Inductive hypothesis -}
\[= \text{sourceP} \_ (t + 1) \_ x \_ s + \text{sourceP} \_ t \_ y \_ s \]

\|--\_ sourceP ~ _ = source
```
B. HOL Light Encoding of the McCarthy-Painter Proof


(* *********************************************** *)
(* 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 *)
(* *********************************************** *)

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" ; ;
B. HOL Light Encoding of the McCarthy-Painter Proof

let $E \text{DEF} =$ new_recursive_definition exp\_RECURSION

\[
\begin{align*}
\text{'}(E \, (\text{Lit } n) \, s = n) \\
\text{'}(E \, (\text{Var } v) \, s = s \, v) \\
\text{'}(E \, (\text{Plus } e_1 \, e_2) \, s = E \, e_1 \, s + E \, e_2 \, s) \text{'};
\end{align*}
\]

(*
* The object language
*
* 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
* 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).
*)

let cell\_INDUCT, cell\_RECURSION =

define_type "cell = Acc
| Reg num";;

let inst\_INDUCT, inst\_RECURSION =

define_type "inst = LI num
| LOAD num
| STO num
| ADD num";;

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

let update\_def =

new\_definition 'update $x \, z \, s \, y =$ if ($y = x$) then $z$ else $s \, y$';;

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

let $S \text{DEF} =$ new_recursive_definition inst\_RECURSION

\[
\begin{align*}
\text{'}(S \, (LI \, n) \, s = update \, Acc \, n \, s) \\
\text{'}(S \, (LOAD \, r) \, s = update \, Acc \, (s \, (Reg \, r)) \, s) \\
\text{'}(S \, (STO \, r) \, s = update \, (Reg \, r) \, (s \, Acc) \, s) \\
\text{'}(S \, (ADD \, r) \, s = update \, Acc \, (s \, (Reg \, r) + s \, Acc) \, s) \text{'};
\end{align*}
\]

(*
* Next we give the semantics of a list of instructions with the
* semantic function $S'$. The execution of an instruction list
* in an initial state is given by executing the first instruction
* in the list in the initial state, which produce a new state $s_1$,
* and taking the execution of the rest of the list in $s_1$.
*)
let S'_DEF = new_recursive_definition list_RECURSION
  "(S' [ ] s = s)
  \ (S' (CONS inst rest) s = S' rest (S inst s))";;

(*
 * The compiler
 *
 *)

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
 ("pl pl2 s. S' (APPEND pl pl2) s = S' pl2 (S' pl1 s)")
 LIST_INDUCT_TAC THEN ASM_SIMP_TAC[S'_DEF; APPEND]);;

let UPDATE_DIFFERENT = prove
 ("!x y z s. (x = y) \rightarrow (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
 *
 *)

let Correctness = prove
 ("forall e, s, r. UPDATE (update x z s y = s y)",
 SIMP_TAC[update_def]);

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

let S'_APPEND = prove
 ("pl pl2 s. S' (APPEND pl pl2) s = S' pl2 (S' pl1 s)")
 LIST_INDUCT_TAC THEN ASM_SIMP_TAC[S'_DEF; APPEND]);;

let UPDATE_DIFFERENT = prove
 ("!x y z s. (x = y) \rightarrow (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 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
 * 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
 * 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'.
B. HOL Light Encoding of the McCarthy-Painter Proof

let correctness_condition = 
  ' !e map s s' r .
  ( !v . map v < r ) =>
  ( !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)))),

(* The Proof *)

(* The proof can be done by induction and careful application of SIMP_TAC[] using the lemmas isolated above. *)

(* 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. *)

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
   SIMP_TAC[ARITH_RULE ' (x < y => x < y + 1 \ (x = y)) /\ x < x + 1'; cellth;
   UPDATESAME; UPDATEDIFFERENT; GSYM SKOLEM_THM; EXISTS_REFL]);;
C. Partial Isabelle/HOL Encoding of the McCarthy-Painter Proof

The proof is incomplete, stopping at the keyword sorry.

theory McCarthy
imports HOL List
begin

  types string = "char list"

  datatype exp = Lit nat
               | Var string
               | Plus exp exp

  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"

  datatype cell = Acc
               | Reg nat

  datatype inst = Li nat
               | Load nat
               | Sto nat
               | Add nat

  definition update :: "cell => nat => (cell => nat) => (cell => nat)" where
    "update x z s y = (if y = x then z else s y)"

  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"

  primrec S ' : : "inst list => (cell => nat) => (cell => nat)" where
    "S' [] s = s"
    | "S' (inst#rest) s = S' rest (S inst s)"

  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]"

  lemma s'_append : "[p1 p2 s . S' (p1 @ p2) s = S' p2 (S' p1 s)"
  apply (rule allI)
  apply (induct_tac p1)
C. Partial Isabelle/HOL Encoding of the McCarthy-Painter Proof

apply auto
done

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 distinctness_cell: "!a. (Acc = Reg a)"
apply auto
done

lemma injectivity_cell: "!a b. (Reg a = Reg b) = (a = b)"
apply auto
done

lemma arith_rule: "\(\forall x :: \text{nat} \forall y :: \text{nat}.\)
  x < y \rightarrow x < y + 1 \land x \neq y"
apply auto
done

(*
  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:
  "\(\forall e m s s' r.\)
  (\(\forall v. m v < r\) \rightarrow
  \(\forall s. s v = s' (\text{Reg } (m v))\) \rightarrow
  (S' (C e m r) s' Acc = E e s) \land
  (\(\forall x < r. (S' (C e m r) s' (\text{Reg } x) = s' (\text{Reg } x))\))"
apply (rule allI)
apply (induct_tac e)
apply (simp_all add: update_def)
apply (simp_all add: update_same update_different s'_append)
sorry
D. Partial Coq Encoding of the McCarthy-Painter Proof

The proof is incomplete, stopping at the keyword Admitted.

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 |
| Reg : nat -> cell .

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 .
 intro x .
case x .
simpl . reflexivity .
simpl . intro n . rewrite <- beq_nat_refl . reflexivity .
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
D. Partial Coq Encoding of the McCarthy-Painter Proof

```
| 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 => Lin :: 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.

Lemma exec' cons : forall (e : target) (p : list target) (s : cell -> nat),
  exec' (e :: p) s = exec' p (exec e s).
intro e p s.
auto.
Qed.

Lemma exec' append : forall (p1 p2 : list target) (s : cell -> nat),
  exec' (p1 ++ p2) s = exec' p2 (exec' p1 s).
intro pl.
induction pl.
auto.
simpl.
intro p2 s.
rewrite IHpl.
reflexivity.
Qed.

Lemma update_same : forall (x : cell) (z : nat) (s : cell -> nat),
  (update x z s x = z).
intro x z s.
unfold update.
rewrite cell_beq_refl.
reflexivity.
Qed.

Lemma S_plus : forall (x : nat), S x = plus x (S 0).
intro x.
induction x.
simpl. reflexivity.
simpl. rewrite <- IHx. reflexivity.
Qed.

Lemma beq_nat_true_iff : forall x y, beq_nat x y = true <-> x = y.
Proof.
  split. apply beq_nat_true.
  intro s; subst; symmetry; apply beq_nat_refl.
Qed.

Lemma le_beq_nat : forall m n, m < n -> beq_nat m n = false.
intro s.
assert (m <> n). omega.
```
generalize H0.
generalize (beq_nat_true_iff m n).
destruct beq_nat; auto.
intros IFF NEQ. elim NEQ. apply IFF; auto.
Qed.

Lemma safe_regs : forall (e : source) (m : nat -> nat)
  (st : cell -> nat) (r x : nat),
  (forall v, m v < r) ->
  (exec' (compile e m (S r)) (update (Reg r) x st) Acc) =
  (exec' (compile e m (S r)) st Acc).

intros e m st r x H.
assert (forall v, beq_nat (m v) r = false). intros. apply le_beq_nat, H.
induction e.
simpl. apply update_same.
simpl. repeat (rewrite update_same). unfold update. simpl. rewrite H0. auto.
simpl. repeat (rewrite exec'append, exec'_cons). simpl.
repeat (rewrite IHe1).
Admitted.

repeat (rewrite update_same). rewrite IHe1.

Theorem correctness : forall (e : source) (m : nat -> nat)
  (ss : nat -> nat) (st : cell -> nat)
  (r : nat),
  (forall v, m v < r) ->
  (forall v, ss v = st (Reg (m v))) ->
  (exec' (compile e m r) st Acc = eval e ss) /
  (forall x (i : nat) st, x < r + i ->
   (exec' (compile e m (r + i)) st (Reg x) = st (Reg x))).

intros e m ss st r Hinitreg Hmapvars.
induction e.
split. simpl. rewrite update_same. reflexivity.
intros x i Hlessr. simpl. unfold update. simpl. reflexivity.
split. simpl. rewrite update_same. rewrite <- Hmapvars. reflexivity.
intros x i Hlessr. simpl. unfold update. simpl. reflexivity.
destruct IHe1 as [IHe1acc IHe1var]. destruct IHe2 as [IHe2acc IHe2var].
split. simpl. repeat (rewrite exec'append, exec'_cons). simpl.
rewrite update_same. repeat (rewrite IHe1acc). rewrite S_+.
rewrite IHe2var. rewrite update_same.

Admitted.