Formal verification of machine-code programs

Magnus O. Myreen

University of Cambridge
Computer Laboratory
Trinity College

December 2008

This dissertation is submitted for the degree of Doctor of Philosophy.

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text. This dissertation does not exceed the regulation length of 60 000 words, including tables and footnotes.

Magnus O. Myreen

Formal verification of machine-code programs

Magnus O. Myreen

Summary

Formal program verification provides mathematical means of increasing assurance for the correctness of software. Most approaches to program verification are either fully automatic and prove only weak properties, or alternatively are manual and labour intensive to apply; few target realistically modelled machine code. The work presented in this dissertation aims to ease the effort required in proving properties of programs on top of detailed models of machine code. The contributions are novel approaches for both verification of existing programs and methods for automatically constructing correct code.

For program verification, this thesis presents a new approach based on translation: the problem of proving properties of programs is reduced, via fully-automatic deduction, to a problem of proving properties of recursive functions. The translation from programs to recursive functions is shown to be implementable in a theorem prover both for simple while-programs as well as real machine code. This verification-after-translation approach has several advantages over established approaches of verification condition generation. In particular, the new approach does not require annotating the program with assertions. More importantly, the proposed approach separates the verification proof from the underlying model so that specific resource names, some instruction orderings and certain control-flow structures become irrelevant. As a result, proof reuse is enabled to a greater extent than in currently used methods. The scalability of this new approach is illustrated through the verification of ARM, x86 and PowerPC implementations of a copying garbage collector.

For construction of correct code, this thesis presents a new compiler which maps functions from logic, via proof, down to multiple carefully modelled commercial machine languages. Unlike previously published work on compilation from higher-order logic, this compiler allows input functions to be partially specified and supports a broad range of user-defined extensions. These features enabled the production of formally verified machine-code implementations of a LISP interpreter, as a case study.

The automation and proofs have been implemented in the HOL4 theorem prover, using a new machine-code Hoare triple instantiated to detailed specifications of ARM, x86 and PowerPC instruction set architectures.

"Making formal methods into normal methods."

— Peter Homeier

Publications

Parts of this thesis have been published; Chapters 2, 3, 4, 6 and 7 have been published as the papers listed below. Although the papers below include authors other than Magnus O. Myreen, this thesis includes nothing which is the outcome of work done in collaboration.

Refereed papers

[85] **Magnus O. Myreen** and Michael J. C. Gordon. *Verified Implementation of LISP on ARM, x86 and PowerPC.* In Theorem Proving in Higher-Order Logics (TPHOLs), 2009. Springer.

[91] **Magnus O. Myreen**, Konrad Slind and Michael J. C. Gordon. *Extensible proof-producing compilation.* In Compiler Construction (CC), 2009. Springer.

[109] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, **Magnus O. Myreen** and Jade Alglave. *The Semantics of x86-CC Multiprocessor Machine Code.* In Principles of Programming Languages (POPL). 2009. ACM. (This paper uses the author's work on programming and evaluating an x86 semantics in HOL4.)

[90] **Magnus O. Myreen**, Konrad Slind and Michael J. C. Gordon. *Machine-code verification for multiple architectures – An application of decompilation into logic.* In Formal Methods in Computer-Aided Design (FMCAD), 2008. IEEE.

[89] **Magnus O. Myreen** and Michael J. C. Gordon. *Transforming Programs into Recursive Functions.* In Brazilian Symposium on Formal Methods (SBMF), 2008. Elsevier.

[88] **Magnus O. Myreen** and Michael J. C. Gordon. *Hoare Logic for Realistically Modelled Machine Code.* In Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2007. Springer.

[86] **Magnus O. Myreen**, Anthony C. J. Fox and Michael J. C. Gordon. *Hoare Logic for ARM Machine Code.* In Fundamentals of Software Engineering (FSEN), 2007. Springer.

Non-refereed papers

[84] **Magnus O. Myreen**. *Verification of LISP Interpreters.* In TPHOLs Emerging trends. 2008. Department of Electrical and Computer Engineering, the University of Concordia.

[87] **Magnus O. Myreen** and Michael J. C. Gordon. *Verification of Machine Code Implementations of Arithmetic Functions for Cryptography.* In TPHOLs Emerging trends, Report 367/07. 2007. Department of Computer Science, University of Kaiserslautern.

Acknowledgments

I would like to thank my supervisor Professor Mike Gordon for the freedom he has given me to explore my research interests, and for the many opportunities he has given to present my work and to network with the research community. I also truly appreciate that Mike has always made himself available for in-depth discussions or simply just a casual chat.

I am indebted to many others for comments, discussions, criticism and encouraging enthusiasm: Behzad Akbarpour, Hasan Amjad, Ralph-Johan Back, Nick Benton, Richard Bornat, Aaron Coble, Boris Feigin, Anthony Fox, Alexey Gotsman, Steve Hand, Peter Homeier, Warren Hunt, Joe Hurd, Matt Kaufmann, John Matthews, J Moore, Alan Mycroft, Michael Norrish, Peter O'Hearn, Scott Owens, Matthew Parkinson, Larry Paulson, Jeff Sanders, Susmit Sarkar, Peter Sewell, Zhong Shao, Konrad Slind, Thomas Tuerk and Lu Zhao.

Aaron Coble and Thomas Tuerk deserve special thanks for carefully reading through early drafts of this dissertation. I am grateful for many comments Mike Gordon offered on the later versions. My PhD examiners, Xavier Leroy and Peter Sewell, gave helpful advice on the final version of the thesis.

Kieu Lien Nguyen has loved and cared for me throughout my PhD.

I also wish to thank my parents Eili and Bertel.

I am grateful for the financial support provided by a scholarship from Osk. Huttusen säätiö, Finland, as well as the EPSRC, UK, for a grant which paid my university fees.

Contents

Chapter 1

Introduction

1.1 Motivation

Computer programs are written based on human intuition, which inevitably results in programming errors. Current practice is to test programs on various sample inputs in the hope of finding any possibility of incorrect program behaviour. Since no amount of testing will cover all cases, this practice is wholly unsatisfactory where faults in programs are costly or even dangerous, e.g. the aviation, automotive and security industries. In the computer science community, it has become generally understood that high assurance of the absence of faults requires mathematical proof.

Early research on proving properties of programs was pioneered in the 1960s and 1970s by Hoare, Floyd, Dijkstra and McCarthy among others [32, 39, 51, 73]. Their proofs were, and most of today's program verification is, carried out under assumptions that are not true for real programs, such as "computer integers are unbounded" and "memory is infinitely large".

This dissertation concerns trustworthy program verification with respect to accurate models of the underlying hardware, i.e. how to prove properties of programs by sound mathematical methods, *formal methods*, without making simplifying assumptions. All proofs are derived by the sound inference rules of higher-order logic.

1.2 Highlights in the history of software verification

The history of software verification is relatively short but diverse. This section briefly describes some of the highlights in its history most relevant to this thesis. Section 1.3 will focus on the advances that have led to the current state-of-the-art in verification of machine-code programs.

In 1949, Turing [115] introduced some of the key concepts of the field of software verification, e.g. assertions. In the 1960s, von Neumann, Floyd, Hoare and McCarthy [41, 39, 51, 73] pioneered methods that shaped the basics of program verification, e.g. use of invariants.

Hoare's seminal paper [51] of 1969 was particularly influential; Hoare showed that one can reason about programs in a structured logical fashion by presenting an axiomatic logic which can build and prove properties of programs from the ground up.

Dijkstra was another significant contributor who, among other things, introduced in 1975 the idea of weakest preconditions [33], i.e. predicate transformers that set a solid foundation for verification condition generation (VCG) [57]. Given a program annotated with logical assertions, a VCG will present the user with verification conditions in the form of mathematical formulae, which if proved imply that all the assertions are logically consistent.

Back introduced the concept of stepwise refinement, as part of the refinement calculus [3, 5]. Back proposed that programs are to be developed in logical steps in a top-down manner from high-level specifications. Back and von Wright later extended their original approach to support refinement of data representations, called data refinement [4]. Morgan [80] developed a similar approach of refinement, which was published in an influential book [81].

In 1977, Cousot and Cousot [29] invented a completely different approach to program verification, called abstract interpretation. In this approach, properties are proved for abstractions of programs, in a manner which allows proved properties to relate back to the original programs — an aspect shared with the method proposed in this thesis. The problem of finding the right level of abstraction was later automated in the form of counter-example-guided abstraction refinement (CEGAR) [26]. In CEGAR, proof-failures automatically generate potential counter-examples, which are used for refining the level of abstraction. CEGAR is particularly well-suited for rather weak properties, e.g. bounds on variables and code reachability, but less applicable to proofs of full functional correctness, which is the topic of this thesis.

In 1989, Gordon [44] showed how a general purpose Hoare logic can be embedded into the logic of a theorem prover. Gordon derived the Hoare axioms from the a program semantics inside a theorem prover, and implemented a verification condition generator as theorem prover tactics. The benefits are a clean abstraction of Hoare logic as well as assurance that each step of the process is sound with respect to the program semantics.

Based on the foundational work mentioned above, a large and diverse body of work emerged over the last twenty years. Particularly, successful was work that took advantage of vast improvements in supporting tools such as model checking [35, 103], SMT solving [94] and theorem proving [110, 119, 97, 10]. Case studies of previously unprecedented scale were undertaken [47, 63, 118, 14], and tools for program verification were developed with higher levels of automation, targeting increasingly realistic languages [52, 16, 61, 62, 28, 117].

A highlight of fully automatic verification is Microsoft's SLAM tool, introduced in 2002 [6]. SLAM implements CEGAR and has been able to point out errors in very large programs. Approaches based on fully automatic verification has since come to emphasise bug finding with implementations based on strong heuristics and often sacrifice soundness. In contrast, work on user-guided approaches have concentrated on assurance of lack of errors, general applicability and solid semantics.

However, the difference in emphasis of automatic and user-guided approaches is not absolute. For instance, separation logic [105], a recent extension to Hoare logic which will

be described in Section 1.3, introduced new methods for tackling pointer aliasing and concurrency. This method has made contributions both to fully automatic tools [120] and to user-guided approaches based on detailed program semantics [114]. Furthermore, there seems to be applications where the automatic and user-guided approaches could complement each other, an example scenario is outlined in Chapter 8.

1.3 Background on verification of machine code

This section outlines some of the key advances that have led to the current state-of-the-art for verification with respect to realistic models of the underlying hardware. Emphasis is given to work where proofs have been mechanically checked.

Goldstein and von Neumann's paper [41] of 1961 appears to be the first to discuss specification and correctness of machine-code programs. Maurer [71] was the first to apply program verification to machine code. Maurer used Floyd's method of inductive assertions and addressed the challenges posed by machine code's ability to modify itself. Clutterbuck and Carré [27] applied a Floyd-Hoare-style verification condition generator to machine code and argued for the importance of verification at the machine-code level. In 1987, Bevier [12] mechanically verified hundreds of lines of machine code for a realistic von Neumann machine.

In 1967, McCarthy and Painter [75] took a different approach; instead of analysing machine code, they proved the correctness of a compiler for expression evaluation. Later, Polak [101] mechanically verified a compiler from a Pascal-like language to an imaginary target machine. As part of the 'CLI short stack' [11] of 1989, Moore and Young verified compilers that target a more realistic von Neumann architecture.

In 1996, Boyer and Yu [16] achieved an impressive milestone by showing that one can verify programs with respect to a detailed operational semantics of a commercial processor. They verified programs using the bare operational semantics of a model of the Motorola MC68020. The proofs, which were carried out in the Boyer-Moore theorem prover Nqthm [55], were long and labour intensive.

A year later, proof-carrying code (PCC) by Necula [92] and typed assembly language (TAL) by Morrisett et al. [83] ignited renewed interest in machine-code verification. Necula introduced the concept of attaching proofs to code in order to enable the code consumer to check that assembly code adheres to certain safety policies, mainly type safety. PCC and TAL apply ideas from type systems to assembly programs. Programs are annotated, automatically as part of compilation, with types according to typing rules; a well-typed program implies that certain safety properties hold. Original incarnations of TAL and PCC have hard-wired machine-specific typing rules. Appel [1] improved on PCC by introducing foundational PCC (FPCC), which removes the need to hard-wire error-prone [59] typing rules and reduced the trusted computing base. TAL has been applied to x86 code [82] and FPCC to assembly code for Sun Sparc [112]. Work on TAL, PCC and FPCC mainly considers techniques that are applied automatically and ensure only weak safety properties. In contrast, this thesis concerns verification of full functional correctness.

Inspired by PCC and FPCC, Shao's group at Yale has developed programming logics to handle stronger properties of machine code, in particular, functional correctness. Shao's group has constructed many different Hoare logics to handle various aspects of low-level code including: embedded code pointers [95], context switching [96], dynamic memory management [76], (intentionally) self-modifying code [18] and hardware interrupts [36]. This work was carried out in the Coq proof assistant [10] based on slightly idealised models of MIPS and, for some projects, x86.

In 2006, Matthews et al. [70] published a paper on mechanised Hoare logic reasoning for machine code in the form of verification condition generator (VCG) inside the ACL2 theorem prover [56]. Hardin et al. have shown that this VCG-based method is applicable to verification of machine code for a very extensive proprietary model of the Rockwell Collins AAMP7G microprocessor, a processor model whose microcode has been formally proved secure using ACL2 [47].

Meanwhile, Leroy [63] reached a milestone in compiler verification; he proved, using Coq, the correctness of an optimising compiler which takes a significant subset of C, called Clight, as input and produces PowerPC assembly code as output. In separate work on compilation, Li and Slind et al. [64] showed that one can compile programs, with proof, directly from the logic of a theorem prover, the HOL4 theorem prover.

Many of the more recent verification efforts, in particular those by Shao's group, make use of a new extension to Hoare logic, called separation logic. Separation logic, which was invented, around 2000, by Reynolds, O'Hearn, Ishtiaq and Yang [106, 53, 105] has introduced new techniques that ease verification of code with explicit pointers. Inspired by Burstall [17], separation logic emphasises 'local reasoning' with which they solve the frame problem [72], i.e. how to state and prove that 'nothing else changed'. Boyer and Yu's proofs [16] were very labour intensive, partly as a result of the frame problem. The frame problem and local reasoning will be explained in later chapters. Successful fully-automatic shape-analysis tools have been implemented using separation logic [9, 120], but none of them target realistically modelled machine code.

In summary, there is renewed interest in assurances that actual machine code is correct. Weak properties of safety have been proved automatically, but stronger properties of functional correctness have required manual and labour intensive methods. This thesis explores new techniques for automating proofs of strong properties down to the level of realistically modelled machine code.

1.4 Contributions

This thesis makes contributions both to approaches for verification of existing programs as well as methods for automatically constructing correct code.

For program verification, a new approach based on translation is presented: the problem of proving properties of programs is reduced, via fully automatic deduction, to a problem of proving properties of recursive functions. The translation from programs to recursive

functions is shown to be implementable in a theorem prover for both simple while-programs and real machine code. This verification-after-translation approach has several advantages over established approaches of verification condition generation. In particular, the proposed approach does not require the user to understand the program semantics or instruction set architecture, as all user interaction can be deferred until after translation. There is no need to annotate programs with invariants; instead verifiers can use induction, natural to theorem provers, in verification proofs. The deferral of manual proofs separates verification proofs from the underlying program semantics so that specific resource names, some instruction orderings and certain control-flow structures become irrelevant. As a result, proof reuse is enabled to a greater extent than in currently used methods. The scalability of the proposed approach is illustrated through the verification of ARM, x86 and PowerPC implementations of a copying garbage collector.

For construction of correct code, a new compiler is described which maps functions from logic, via proof, down to multiple carefully modelled commercial machine languages. Li and Slind et al. [64, 66] have explored proof-producing compilation from logic before, but unlike previous work, the compiler presented here allows input functions to be partially specified and supports a broad range of user-defined extensions. These features enabled the production of formally verified machine-code implementations of a LISP interpreter, as a case study.

The automation and proofs presented here have been implemented in the HOL4 theorem prover, using a new machine-code Hoare triple instantiated to detailed specifications of ARM, x86 and PowerPC instruction set architectures.

1.5 Thesis structure

Each chapter is written to stand independently of the others. Some of these chapters correspond very closely to a selection of papers by the author, listed on page 7. Chapters 2, 3, 4, 6 and 7 have been published as papers [89], [88, 86], [90], [91] and [85], respectively.

Chapter 2 presents a new technique by which programs in a simple while-language can be translated, via fully-automatic deduction, into tail-recursive functions defined in the language of a theorem prover. This translation allows the user to prove properties of tail-recursive functions rather than while-programs.

Chapter 3 defines a Hoare logic designed to fit on top of accurate models of machine languages. Its judgements provide concise specifications of functional correctness, termination and resource usage for ARM, PowerPC and x86 machine code.

Chapter 4 combines the ideas of transforming programs, via deduction, into the recursive functions of Chapter 2 with the machine-code Hoare triple of Chapter 3. The result is a fully-automatic proof-producing decompiler from machine code to tail-recursive functions.

Chapter 5 presents a case study that uses the decompiler from Chapter 4 to verify the correctness of multiple machine-code implementations of a memory allocator with a built-in Cheney collector. The proofs proceed by refinements from higher levels of abstraction.

Chapter 6 develops a proof-producing compiler using decompilation. The compiler maps HOL functions to ARM, PowerPC and x86 machine code, applies multiple optimising transformations to its input and supports a wide range of user-defined extensions.

Chapter 7 presents a case study which shows that the proposed techniques can be applied to construct verified applications. The compiler from Chapter 6 is used to construct correct machine-code implementations of an interpreter for a subset of LISP.

Chapter 8 concludes with a summary of contributions, outlines possible applications of the work presented and discusses directions for future work.

1.6 Proof assistant: HOL4

Commercial machine languages support a large number of instructions and a multitude of features. As a result, realistic models of such languages are inherently large. The definitions of the models used in this work are many thousands of lines long, hence proving properties of such large models inevitably requires a mechanical proof assistant.

The work presented here was developed using the HOL4 theorem prover [110], which implements a higher-order logic based on Church's simple theory of types [25]. In HOL4, the user proves theorems by steering the system with proof tactics and proof rules, or custom built ML programs [43]. HOL4 prevents false statements from being proved by enforcing, through the ML type system, restrictions by which all proofs must pass the logical core of HOL4. This logical core is an ML module which implements the basic inference rules of higher-order logic.

All of the proofs and automation described in this thesis have been implemented using HOL4. The proof scripts were, at the time of writing, available as part of the HOL4 sources

`http://hol.sourceforge.net/`

in the publicly available source repository under `HOL/examples/machine-code`.

1.7 Notation

Constants are written in sans-serif, e.g. div and mod, variables in slanted text, e.g. x and y. True and false are T and F, respectively. Abstractions are written $\lambda x.\, t$ and function application is written either $f(x)$ or $f\ x$.

Chapter 2

Transforming programs into recursive functions

This chapter presents a new proof-assistant based approach to program verification: programs are translated, via fully-automatic deduction, into tail-recursive functions defined in the logic of a theorem prover. This approach improves on well-established methods based on Hoare logic and verification condition generation (VCG) by removing the need to annotate programs with assertions.

The automatically provable correspondence between programs and recursive functions is one of the core ideas that make the work in this dissertation possible. This chapter shows how the idea can be applied in a simple setting; subsequent chapters apply this translation technique to realistically modelled machine code.

2.1 Introduction

Program verification is commonly done by reasoning in programming logics or by generating verification conditions. These approaches serve as interfaces from programs to 'mathematics'. Programming logics provide an axiomatic formal system for reasoning about programs, whilst verification condition generators (VCGs) extract conditions, typically as first order formulae, sufficient for properties of programs to hold.

This chapter presents an approach in which programs are translated, via fully-automatic deduction, into recursive functions defined directly in the native logic of a theorem prover. The approach has a number of benefits for program verification:

1. makes subsequent proofs natural for the theorem prover, e.g. allows use of induction;
2. reduces the verification problem to its essence, the computation;
3. allows reuse of already proved algorithms, e.g. from library theories;

4. facilitates use of already existing automation, e.g. existing decision procedures;

5. is easier to implement than a trustworthy, i.e. proved or proof-producing, VCG;

6. provides a back-end that can support verification for different languages.

To give the reader a sense of what these translations are, one such translation is illustrated using an example by McCarthy [73]. McCarthy points out the correspondence between imperative programs and recursive functions by showing that the following program

```
y := 1;
while 0 < x do
  y := x × y;
  x := x - 1;
end
```

implements the function f below.[1] Here g corresponds to the `while` loop.

$$\begin{aligned} \mathsf{f}(x,y) &= \mathsf{let}\ y = 1\ \mathsf{in}\ \mathsf{g}(x,y) \\ \mathsf{g}(x,y) &= \mathsf{if}\ \neg(0 < x)\ \mathsf{then}\ (x,y)\ \mathsf{else} \\ &\qquad \mathsf{let}\ y = x \times y\ \mathsf{in} \\ &\qquad \mathsf{let}\ x = x - 1\ \mathsf{in} \\ &\qquad\quad \mathsf{g}(x,y) \end{aligned}$$

This chapter's contribution is a technique which uses a theorem prover to perform such translations fully automatically by mechanised deduction from a formal semantics of the programming language. The technique automatically creates a proof certifying that the recursive functions represent the semantics of the program. More specifically, this chapter:

1. shows how a translation into recursive functions can be constructed by instantiations of proof rules for both partial- (Section 2.5) and total-correctness (Section 2.6);

2. states termination conditions in a way which makes them naturally a consequence of any proof by induction used for verification (Section 2.6.1 and 2.7); and

3. illustrates advantages of proof by induction over established methods based on verification condition generation (Sections 2.7 and 2.8).

The automatically-provable correspondence between programs and recursive functions is one of the core ideas that make the work presented in this dissertation possible. This chapter shows how the idea can be applied in a simple setting; subsequent chapters apply this translation technique to realistically modelled machine code.

[1] McCarthy used a different notation for both the program and the function.

2.2 Semantics of a simple language

Consider programs constructed from the following grammar. Here i stands for integers and v for strings.

$$\begin{array}{rcl} e & ::= & \mathsf{Const}\ i \mid \mathsf{Var}\ v \mid \mathsf{Plus}\ e\ e \mid \mathsf{Sub}\ e\ e \mid \ldots \\ b & ::= & \mathsf{Not}\ b \mid \mathsf{Equal}\ e\ e \mid \mathsf{Less}\ e\ e \mid \ldots \\ prog & ::= & \mathsf{Skip} \mid \mathsf{Assign}\ v\ e \mid \mathsf{Seq}\ prog\ prog \mid \mathsf{Cond}\ b\ prog\ prog \mid \mathsf{While}\ b\ prog \end{array}$$

In HOL, the program from Section 2.1 above is encoded as:

```
Seq (Assign "y" (Const 1))
    (While (Less (Const 0) (Var "x"))
        (Seq (Assign "y" (Times (Var "x") (Var "y")))
             (Assign "x" (Sub (Var "x") (Const 1)))))
```

This chapter builds on work by Camilleri and Melham [19], who define in HOL a big-step operational semantics for such programs. This presentation considers states that are (finite) partial functions from variable names (of type :string) to integers (type :int). Let neval and beval define evaluations over expressions.

$$\begin{array}{l} \mathsf{neval}\ (\mathsf{Const}\ k)\ state = k \\ \mathsf{neval}\ (\mathsf{Var}\ v)\ state = state(v) \\ \mathsf{neval}\ (\mathsf{Plus}\ e_1\ e_2)\ state = \mathsf{neval}\ e_1\ state + \mathsf{neval}\ e_2\ state \\ \ldots \\ \\ \mathsf{beval}\ (\mathsf{Not}\ b)\ state = \neg\,(\mathsf{beval}\ b\ state) \\ \mathsf{beval}\ (\mathsf{Equal}\ e_1\ e_2)\ state = (\mathsf{neval}\ e_1\ state = \mathsf{neval}\ e_2\ state) \\ \ldots \end{array}$$

The semantics of programs is defined by eval $program\ s_1\ s_2$, which relates s_1 to s_2 if and only if state s_1 can be transformed into state s_2 by an execution of $program$. Here $s[v \mapsto x]$ updates state s so that it maps v to x. Let eval be the least relation satisfying the following:

$$\mathsf{eval}\ \mathsf{Skip}\ s\ s$$

$$\mathsf{eval}\ (\mathsf{Assign}\ v\ e)\ s\ (s[v \mapsto \mathsf{neval}\ e\ s])$$

$$\mathsf{eval}\ c_1\ s_1\ s_2 \wedge \mathsf{eval}\ c_2\ s_2\ s_3 \Rightarrow \mathsf{eval}\ (\mathsf{Seq}\ c_1\ c_2)\ s_1\ s_3$$

$$\mathsf{eval}\ c_1\ s_1\ s_2 \wedge \mathsf{beval}\ b\ s_1 \Rightarrow \mathsf{eval}\ (\mathsf{Cond}\ b\ c_1\ c_2)\ s_1\ s_2$$

$$\mathsf{eval}\ c_2\ s_1\ s_2 \wedge \neg\,\mathsf{beval}\ b\ s_1 \Rightarrow \mathsf{eval}\ (\mathsf{Cond}\ b\ c_1\ c_2)\ s_1\ s_2$$

$$\neg\,\mathsf{beval}\ b\ s \Rightarrow \mathsf{eval}\ (\mathsf{While}\ b\ c)\ s\ s$$

$$\mathsf{eval}\ c\ s_1\ s_2 \wedge \mathsf{eval}\ (\mathsf{While}\ b\ c)\ s_2\ s_3 \wedge \mathsf{beval}\ b\ s_1 \Rightarrow \mathsf{eval}\ (\mathsf{While}\ b\ c)\ s_1\ s_3$$

2.3 Hoare logic

Specifications will be expressed here using Hoare triples [51]. A standard *partial-correctness* Hoare triple, $\{p\}\, c\, \{q\}$, is written here as $\mathsf{Spec}\ p\ c\ q$ and given the semantics defined by:

$$\mathsf{Spec}\ p\ c\ q \quad = \quad \forall s_1 s_2.\ p\ s_1 \wedge \mathsf{eval}\ c\ s_1\ s_2 \Rightarrow q\ s_2$$

Informally, this means: if precondition p is satisfied by some current state s_1 then postcondition q holds for any state s_2 reachable by an execution of program c.

Notice that Spec does not guarantee that there exists any reachable final state s_2, i.e. Spec does not ensure termination. The following *total-correctness* Hoare triple requires termination for any state s_1 which satisfies precondition p:

$$\mathsf{TotalSpec}\ p\ c\ q \quad = \quad \mathsf{Spec}\ p\ c\ q \wedge \forall s_1.\ p\ s_1 \Rightarrow \exists s_2.\ \mathsf{eval}\ c\ s_1\ s_2$$

2.4 A thin layer of separation logic

A separating conjunction (borrowed from separation logic [105]) will be used to aid automation. The separating conjunction $p * q$ is true for state s, a partial function from variable names to values, if s can be split into two disjoint states s_1, s_2 such that p holds for s_1 and q holds for s_2:

$$\begin{array}{l}\mathsf{split}\ s\ (s_1, s_2) = \\ \quad (\mathsf{domain}\ s_1 \cup \mathsf{domain}\ s_2 = \mathsf{domain}\ s) \wedge (\mathsf{domain}\ s_1 \cap \mathsf{domain}\ s_2 = \{\}) \wedge \\ \quad (\forall v.\ v \in \mathsf{domain}\ s \Rightarrow (s(v) = \mathsf{if}\ v \in \mathsf{domain}\ s_1\ \mathsf{then}\ s_1(v)\ \mathsf{else}\ s_2(v)))\end{array}$$

$$(p * q)\ s = \exists s_1\ s_2.\ \mathsf{split}\ s\ (s_1, s_2) \wedge p\ s_1 \wedge q\ s_2$$

The $*$-operator is both associative and commutative.

New versions of Hoare triples are defined using the separating conjunction:

$$\begin{array}{rcl}\mathsf{SepSpec}\ p\ c\ q & = & \forall r.\ \mathsf{Spec}\ (p * r)\ c\ (q * r) \\ \mathsf{SepTotalSpec}\ p\ c\ q & = & \forall r.\ \mathsf{TotalSpec}\ (p * r)\ c\ (q * r)\end{array}$$

These definitions incorporate the semantic idea underlying the frame rule of separation logic, but differ because, in separation logic, the $*$-operator is used to separate heap assertions but not stack assertions. In contrast, here the separating conjunction is used to separate arbitrary resources; this bears some similarity to the idea of 'variable as resource' [98].

The remaining sections discuss proof rules that are derived from the definitions of SepSpec and SepTotalSpec. One such rule, the frame rule, allows any assertion r to be added (unconditionally) to specifications:

$$\forall r.\ \ \mathsf{SepSpec}\ p\ c\ q \Rightarrow \mathsf{SepSpec}\ (p * r)\ c\ (q * r)$$

$$\forall r.\ \ \mathsf{SepTotalSpec}\ p\ c\ q \Rightarrow \mathsf{SepTotalSpec}\ (p * r)\ c\ (q * r)$$

The following example will illustrate this. Let var v x s assert that variable v has value x in state s:

$$\mathsf{var}\ v\ x\ s\ =\ (\mathsf{domain}\ s = \{v\}) \wedge (s(v) = x)$$

Now the assignment `b := a + b` has the following specification. Here the precondition assumes that variable `"a"` has initial value a and that `"b"` has value b. The postcondition states that the assignment updates variable `"b"` to $a + b$.

$$\begin{aligned}\mathsf{SepSpec}\quad & (\mathsf{var}\ \texttt{"a"}\ a * \mathsf{var}\ \texttt{"b"}\ b)\\ & (\mathsf{Assign}\ \texttt{"b"}\ (\mathsf{Plus}\ (\mathsf{Var}\ \texttt{"a"})\ (\mathsf{Var}\ \texttt{"b"})))\\ & (\mathsf{var}\ \texttt{"a"}\ a * \mathsf{var}\ \texttt{"b"}\ (a+b))\end{aligned}$$

The frame rule allows us to infer that resources not mentioned in the specification are left untouched, e.g. one can instantiate r in the frame rule with var `"c"` c and, thus have:

$$\begin{aligned}\mathsf{SepSpec}\quad & (\mathsf{var}\ \texttt{"a"}\ a * \mathsf{var}\ \texttt{"b"}\ b * \mathsf{var}\ \texttt{"c"}\ c)\\ & (\mathsf{Assign}\ \texttt{"b"}\ (\mathsf{Plus}\ (\mathsf{Var}\ \texttt{"a"})\ (\mathsf{Var}\ \texttt{"b"})))\\ & (\mathsf{var}\ \texttt{"a"}\ a * \mathsf{var}\ \texttt{"b"}\ (a+b) * \mathsf{var}\ \texttt{"c"}\ c)\end{aligned}$$

Instantiating the frame rule with var `"a"` x would give a false precondition, and thence a vacuously true statement (var `"a"` $a *$ var `"a"` $x = \lambda state.\ \mathsf{F}$). An example of what a SepSpec specification means in terms of eval is given in Section 2.7.1.

Boyer and Yu's proofs [16] had to routinely specify that 'no other part of the state was modified'. By using the frame rule, such statements become unnecessary, since all resources that are not mentioned in the precondition are guaranteed to stay unchanged. The frame rule makes reasoning 'local' since specifications will only mention resources that are used.

2.5 Constructing specifications for partial-correctness

This section presents proof rules for SepSpec, which are proved from its definition, and illustrates how these rules can be used to automatically derive functions describing the effect of programs.

The proposed algorithm proceeds in a bottom-up manner based on the structure of programs (defined in Section 2.2). For instance, when McCarthy's example is traversed,

```
Seq (Assign "y" (Const 1))
    (While (Less (Const 0) (Var "x"))
        (Seq (Assign "y" (Times (Var "x") (Var "y")))
             (Assign "x" (Sub (Var "x") (Const 1)))))
```

functions will be generated for the Assign-constructs first, then for the innermost Seq, followed by the While-construct and finally the outermost Seq. The derived SepSpec theorems will be of the following form:

$$\begin{aligned}\mathsf{SepSpec}\quad & (\mathsf{var}\ n_1\ v_1 * \ldots * \mathsf{var}\ n_k\ v_k)\\ & \mathit{program}\\ & (\mathsf{let}\ v_1 \ldots v_k = f(v_1 \ldots v_k)\ \mathsf{in}\ (\mathsf{var}\ n_1\ v_1 * \ldots * \mathsf{var}\ n_k\ v_k))\end{aligned}$$

2.5.1 Assignments

Assignments update the value of a variable and can always be implemented by a let-expression, e.g. Assign "b" (Plus (Var "a") (Var "b")) is captured by let $b = a + b$ in (a, b). One expresses this fact by the following theorem:

$$\begin{aligned}\mathsf{SepSpec}\quad & (\mathsf{var}\ \texttt{"a"}\ a * \mathsf{var}\ \texttt{"b"}\ b)\\ & (\mathsf{Assign}\ \texttt{"b"}\ (\mathsf{Plus}\ (\mathsf{Var}\ \texttt{"a"})\ (\mathsf{Var}\ \texttt{"b"})))\\ & (\mathsf{let}\ b = a + b\ \mathsf{in}\ (\mathsf{var}\ \texttt{"a"}\ a * \mathsf{var}\ \texttt{"b"}\ b))\end{aligned}$$

2.5.2 Sequential composition

Sequential composition is introduced using the following proof rule:

$$\mathsf{SepSpec}\ p\ c_1\ m\ \wedge\ \mathsf{SepSpec}\ m\ c_2\ q\ \Rightarrow\ \mathsf{SepSpec}\ p\ (\mathsf{Seq}\ c_1\ c_2)\ q$$

Given specifications for two programs, say, program1 and program2,

$$\begin{aligned}\mathsf{SepSpec}\quad & (\mathsf{var}\ \texttt{"a"}\ a * \mathsf{var}\ \texttt{"b"}\ b * \mathsf{var}\ \texttt{"c"}\ c)\\ & \mathsf{program1}\\ & (\mathsf{let}\ (b, c) = \mathsf{f}_1(a, b, c)\ \mathsf{in}\\ & \quad (\mathsf{var}\ \texttt{"a"}\ a * \mathsf{var}\ \texttt{"b"}\ b * \mathsf{var}\ \texttt{"c"}\ c))\end{aligned}$$

$$\begin{aligned}\mathsf{SepSpec}\quad & (\mathsf{var}\ \texttt{"a"}\ a * \mathsf{var}\ \texttt{"b"}\ b)\\ & \mathsf{program2}\\ & (\mathsf{let}\ b = \mathsf{f}_2(a, b)\ \mathsf{in}\\ & \quad (\mathsf{var}\ \texttt{"a"}\ a * \mathsf{var}\ \texttt{"b"}\ b))\end{aligned}$$

such specifications can be composed by first using the frame rule to insert var "c" c into the second specification and then instantiating the second specification with the let-update from the first specification, followed by an application of the composition rule:

$$\begin{aligned}\mathsf{SepSpec}\quad & (\mathsf{var}\ \texttt{"a"}\ a * \mathsf{var}\ \texttt{"b"}\ b * \mathsf{var}\ \texttt{"c"}\ c)\\ & (\mathsf{Seq}\ \mathsf{program1}\ \mathsf{program2})\\ & (\mathsf{let}\ (b, c) = \mathsf{f}_1(a, b, c)\ \mathsf{in}\\ & \ \mathsf{let}\ b = \mathsf{f}_2(a, b)\ \mathsf{in}\\ & \quad (\mathsf{var}\ \texttt{"a"}\ a * \mathsf{var}\ \texttt{"b"}\ b * \mathsf{var}\ \texttt{"c"}\ c))\end{aligned}$$

2.5.3 Conditional statements

Conditional execution is based on a guard, e.g. (Less (Var "a") (Const 5)). Let SepGuard p g b require that function g is equivalent to the guard b, if the resource assertion p holds:

$$\mathsf{SepGuard}\ p\ g\ b\quad =\quad \forall r\ s\ x.\ (p\ x * r)\ s \Rightarrow (g\ x = \mathsf{beval}\ b\ s)$$

As an example, $(\lambda a.\ a < 5)$ is related to (Less (Var "a") (Const 5)):

$$\mathsf{SepGuard}\ (\lambda a.\ \mathsf{var}\ \texttt{"a"}\ a)\ (\lambda a.\ a < 5)\ \ (\mathsf{Less}\ (\mathsf{Var}\ \texttt{"a"})\ (\mathsf{Const}\ 5))$$

Functions describing conditionals are constructed using the following rule:

$$\begin{array}{l}\mathsf{SepGuard}\ p\ g\ b\ \Rightarrow\\ \mathsf{SepSpec}\ (p\ x)\ c_1\ (p\ y)\ \Rightarrow\\ \mathsf{SepSpec}\ (p\ x)\ c_2\ (p\ z)\ \Rightarrow\\ \mathsf{SepSpec}\ (p\ x)\ (\mathsf{Cond}\ b\ c_1\ c_2)\ (p\ (\mathsf{if}\ g\ x\ \mathsf{then}\ y\ \mathsf{else}\ z))\end{array}$$

As an example:

$$\begin{array}{l}\mathsf{SepSpec}\ (\mathsf{var}\ \texttt{"a"}\ a)\\ \quad(\mathsf{Cond}\ (\mathsf{Less}\ (\mathsf{Var}\ \texttt{"a"})\ (\mathsf{Const}\ 5))\\ \quad\quad(\mathsf{Assign}\ \texttt{"a"}\ (\mathsf{Const}\ 1))\\ \quad\quad(\mathsf{Assign}\ \texttt{"a"}\ (\mathsf{Plus}\ (\mathsf{Var}\ \texttt{"a"})\ (\mathsf{Const}\ 1))))\\ \quad(\mathsf{let}\ a = (\mathsf{if}\ a < 5\ \mathsf{then}\ (\mathsf{let}\ a = 1\ \mathsf{in}\ a)\\ \quad\quad\quad\quad\quad\quad\quad\quad\mathsf{else}\ \ (\mathsf{let}\ a = a + 1\ \mathsf{in}\ a))\ \mathsf{in}\ (\mathsf{var}\ \texttt{"a"}\ a))\end{array}$$

2.5.4 While loops

A recursive function is needed for describing the behaviour of While b c. For this purpose, a tail-recursive function while is defined as follows:

$$\mathsf{while}\ guard\ f\ x \quad = \quad \mathsf{if}\ guard\ x\ \mathsf{then}\ (\mathsf{while}\ guard\ f\ (f\ x))\ \mathsf{else}\ x$$

Moore and Manolios showed that such a function can be defined in logic without a termination proof [69].[2] The while function corresponds to the effect of a while loop, as can be seen from the following loop rule:

$$\begin{array}{l}\mathsf{SepGuard}\ p\ guard\ b\ \Rightarrow\\ (\forall x.\ \mathsf{SepSpec}\ (p\ x)\ c\ (p\ (f\ x)))\ \Rightarrow\\ (\forall x.\ \mathsf{SepSpec}\ (p\ x)\ (\mathsf{While}\ b\ c)\ (p\ (\mathsf{while}\ guard\ f\ x)))\end{array}$$

Before presenting a sketch of the proof of this rule, an example will illustrate its use. Let program c = Assign "a" (Plus (Var "a") (Const 1)) and b = (Less (Var "a") (Const 5)); then one can reverse beta conversions to make the assignment

$$\begin{array}{l}\mathsf{SepSpec}\ ((\lambda a.\ \mathsf{var}\ \texttt{"a"}\ a)\ a)\\ \quad\quad(\mathsf{Assign}\ \texttt{"a"}\ (\mathsf{Plus}\ (\mathsf{Var}\ \texttt{"a"})\ (\mathsf{Const}\ 1))\\ \quad\quad((\lambda a.\ \mathsf{var}\ \texttt{"a"}\ a)\ ((\lambda a.\ \mathsf{let}\ a = a + 1\ \mathsf{in}\ a)\ a))\end{array}$$

fit the loop rule. If add_loop is defined by

$$\mathsf{add\_loop} \quad = \quad \mathsf{while}\ (\lambda a.\ a < 5)\ (\lambda a.\ \mathsf{let}\ a = a + 1\ \mathsf{in}\ a)$$

then the loop rule produces:

$$\begin{array}{l}\mathsf{SepSpec}\ (\mathsf{var}\ \texttt{"a"}\ a)\\ \quad(\mathsf{While}\ (\mathsf{Less}\ (\mathsf{Var}\ \texttt{"a"})\ (\mathsf{Const}\ 5))\ (\mathsf{Assign}\ \texttt{"a"}\ (\mathsf{Plus}\ (\mathsf{Var}\ \texttt{"a"})\ (\mathsf{Const}\ 1))))\\ \quad(\mathsf{let}\ a = \mathsf{add\_loop}\ a\ \mathsf{in}\ (\mathsf{var}\ \texttt{"a"}\ a))\end{array}$$

[2]The author is grateful to Lawrence C. Paulson for pointing out this result, which had already been formalised in a HOL4 library called `whileTheory`.

An equation for add_loop can be proved automatically by unfolding while:

$$\begin{aligned}
& \mathsf{add_loop}\ a \\
= \ & \mathsf{while}\ (\lambda a.\ a < 5)\ (\lambda a.\ \mathsf{let}\ a = a + 1\ \mathsf{in}\ a)\ a \\
= \ & \mathsf{if}\ (\lambda a.\ a < 5)\ a\ \mathsf{then} \\
& \quad \mathsf{while}\ (\lambda a.\ a < 5)\ (\lambda a.\ \mathsf{let}\ a = a + 1\ \mathsf{in}\ a)\ ((\lambda a.\ \mathsf{let}\ a = a + 1\ \mathsf{in}\ a)\ a)\ \mathsf{else}\ a \\
= \ & \mathsf{if}\ a < 5\ \mathsf{then}\ \mathsf{add_loop}\ (\mathsf{let}\ a = a + 1\ \mathsf{in}\ a)\ \mathsf{else}\ a \\
= \ & \mathsf{if}\ a < 5\ \mathsf{then}\ (\mathsf{let}\ a = a + 1\ \mathsf{in}\ \mathsf{add_loop}\ a)\ \mathsf{else}\ a
\end{aligned}$$

2.5.5 Proof of loop rule

The proof of the loop rule is based on a case split on whether a state can be reached where the guard for while is made false by repeatedly executing the body of the loop. Let funpow $f\ n\ x$ be the value of n applications of f to x:

$$\begin{aligned}
\mathsf{funpow}\ f\ 0\ x &= x \\
\mathsf{funpow}\ f\ (n{+}1)\ x &= \mathsf{funpow}\ f\ n\ (f\ x)
\end{aligned}$$

The proof is based on a case split on:

$$\exists n.\ \neg\, guard\ (\mathsf{funpow}\ f\ n\ x)$$

If there exists such an n, then the loop is unfolded n times to get the desired result, otherwise the loop does not terminate, making the partial-correctness Hoare triple SepSpec vacuously true.

2.5.6 McCarthy's example

The theorem stating the correspondence between the function and the program in McCarthy's example (from Section 2.1, where f is defined) is the following:

```
SepSpec (var "x" x * var "y" y)
        (Seq (Assign "y" (Const 1))
             (While (Less (Const 0) (Var "x"))
                    (Seq (Assign "y" (Times (Var "x") (Var "y")))
                         (Assign "x" (Sub (Var "x") (Const 1))))))
        (let (x, y) = f(x, y) in (var "x" x * var "y" y))
```

2.6 Constructing specifications for total-correctness

Most of the proof rules and hence most of the derivations of the executed function are exactly the same for total-correctness. The required change is that termination for loops must be assumed, i.e. for a loop described by while *guard f x*, one needs to assume that some number of applications of f to x will eventually turn *guard* to false:

$$\exists n.\ \neg\, guard\ (\mathsf{funpow}\ f\ n\ x)$$

This necessary assumption in the loop rule introduces a precondition, which needs to be threaded through the entire development.

2.6.1 While loops

The precondition of the loop rule is defined with the following format. Here terminates states that the loop described by while $guard\ f\ x$ terminates and that each execution of the loop body can assume the invariant $side$.

$$\begin{aligned}&\mathsf{terminates}\ f\ guard\ side\ x\ =\\ &\quad(\exists n.\ \neg guard\ (\mathsf{funpow}\ f\ n\ x))\ \wedge\\ &\quad(\forall k.\ (\forall m.\ m \leq k \Rightarrow guard\ (\mathsf{funpow}\ f\ m\ x)) \Rightarrow side\ (\mathsf{funpow}\ f\ k\ x))\end{aligned}$$

The following theorem states that terminates can be unfolded like a recursive function:

$$\mathsf{terminates}\ f\ guard\ side\ x\ \ =\ \ (guard\ x\ \Rightarrow\ side\ x\ \wedge\ \mathsf{terminates}\ f\ guard\ side\ (f\ x))$$

This unfolding is particularly useful in proofs by induction where the base case makes $guard$ false and the step case unfolds terminates once, i.e. this unfolding makes the termination condition follow from any induction used for verification of while.

The following induction principle is proved from the definition of terminates.

$$\begin{aligned}\forall\varphi.\quad&(\forall x.\ guard\ x \wedge side\ x \wedge \varphi\ (f\ x) \Rightarrow \varphi\ x)\ \wedge\\ &(\forall x.\ \neg guard\ x \Rightarrow \varphi\ x) \Rightarrow\\ &(\forall x.\ \mathsf{terminates}\ f\ guard\ side\ x \Rightarrow \varphi\ x)\end{aligned}$$

The induction principle is used in proving a rule for While:

$$\begin{aligned}&\mathsf{SepGuard}\ p\ guard\ b\ \Rightarrow\\ &(\forall x.\ guard\ x \wedge side\ x \Rightarrow \mathsf{SepTotalSpec}\ (p\ x)\ c\ (p\ (f\ x)))\ \Rightarrow\\ &(\forall x.\ \mathsf{terminates}\ f\ guard\ side\ x \Rightarrow \mathsf{SepTotalSpec}\ (p\ x)\ (\mathsf{While}\ b\ c)\ (p\ (\mathsf{while}\ guard\ f\ x)))\end{aligned}$$

This loop rule is a consequence of the induction arising from terminates with φ as:

$$\lambda x.\ \mathsf{SepSpec}\ (p\ x)\ (\mathsf{While}\ b\ c)\ (p\ (\mathsf{while}\ g\ f\ x))$$

The total-correctness specification used to illustrate the partial-correctness loop rule will have a precondition of the following form.

$$\mathsf{add\_loop\_pre}\ \ =\ \ \mathsf{terminates}\ (\lambda a.\ \mathsf{let}\ a = a + 1\ \mathsf{in}\ a)\ (\lambda a.\ a < 5)\ (\lambda a.\ \mathsf{T})$$

The While-rule gives the following:

```
add_loop_pre a ⇒
SepTotalSpec (var "a" a)
             (While (Less (Var "a") (Const 5))
                (Assign "a" (Plus (Var "a") (Const 1)))
             (let a = add_loop a in (var "a" a))
```

The precondition, i.e. the termination condition, can be stated as the following rewrite, which can be unfolded until the guard $a < 5$ becomes false.

$$\mathsf{add\_loop\_pre}\ a \quad = \quad (a < 5 \Rightarrow \mathsf{let}\ a = a + 1\ \mathsf{in}\ \mathsf{add\_loop\_pre}\ a)$$

For this case, it is easy to prove $\forall a.\ \mathsf{add\_loop\_pre}\ a$.

2.6.2 Propagating termination conditions

For total-correctness, special termination conditions need to be passed through the rules for composing specifications. The loop rule allows accumulation of side-conditions in terminates. For sequential composition the side conditions are composed, e.g.

$$\begin{array}{rcl} side_1\ x & \Rightarrow & \mathsf{SepTotalSpec}\ (p\ x)\ c_1\ (\mathsf{let}\ x = f_1\ x\ \mathsf{in}\ p\ x) \\ side_2\ x & \Rightarrow & \mathsf{SepTotalSpec}\ (p\ x)\ c_2\ (\mathsf{let}\ x = f_2\ x\ \mathsf{in}\ p\ x) \end{array}$$

produces the following, using the rule for sequential composition.

$$\begin{array}{l} side_1\ x \wedge (\mathsf{let}\ x = f_1\ x\ \mathsf{in}\ side_2\ x) \Rightarrow \\ \mathsf{SepTotalSpec}\ (p\ x) \\ \qquad\qquad (\mathsf{Seq}\ c_1\ c_2) \\ \qquad\qquad (\mathsf{let}\ x = f_1\ x\ \mathsf{in}\ (\mathsf{let}\ x = f_2\ x\ \mathsf{in}\ p\ x)) \end{array}$$

The rule for conditional statements introduces a conditional into the side condition:

$$\begin{array}{l} (\mathsf{if}\ h\ x\ \mathsf{then}\ side_1\ x\ \mathsf{else}\ side_2\ x) \Rightarrow \\ \mathsf{SepTotalSpec}\ (p\ x) \\ \qquad\qquad (\mathsf{Cond}\ g\ c_1\ c_2) \\ \qquad\qquad (\mathsf{let}\ x = (\mathsf{if}\ h\ x\ \mathsf{then}\ f_1\ x\ \mathsf{else}\ f_2\ x)\ \mathsf{in}\ p\ x) \end{array}$$

Assignments stay the same:

$$\forall p\ v\ e\ q.\ \ \mathsf{SepTotalSpec}\ p\ (\mathsf{Assign}\ v\ e)\ q \quad = \quad \mathsf{SepSpec}\ p\ (\mathsf{Assign}\ v\ e)\ q$$

2.6.3 McCarthy's example

The following is the accumulated precondition for the initial example program, when a total-correctness specification is derived.

$$\begin{array}{rcl} \mathsf{f_{pre}}(x, y) & = & \mathsf{let}\ y = 1\ \mathsf{in}\ \mathsf{g_{pre}}(x, y) \\ \mathsf{g_{pre}}(x, y) & = & (0 < x \Rightarrow \mathsf{let}\ y = x \times y\ \mathsf{in} \\ & & \qquad \mathsf{let}\ x = x - 1\ \mathsf{in} \\ & & \qquad\quad \mathsf{g_{pre}}(x, y)) \end{array}$$

2.7 Proving conditional termination

Consider the following program, which stores `n div 2` in `a` if `n` is initially even and non-negative. The program fails to terminate if `n` is odd or negative.

```
a := 0;
while (n ≠ 0) do
  a := a + 1;
  n := n - 2
end
```

When the code above is translated into recursive functions, function d is generated

$$\begin{aligned} \mathsf{d}(a,n) &= \mathsf{let}\ a = 0\ \mathsf{in}\ \mathsf{dl}(a,n) \\ \mathsf{dl}(a,n) &= \mathsf{if}\ n = 0\ \mathsf{then}\ (a,n)\ \mathsf{else} \\ &\quad\ \mathsf{let}\ a = a+1\ \mathsf{in} \\ &\quad\ \mathsf{let}\ n = n-2\ \mathsf{in} \\ &\quad\ \mathsf{dl}(a,n) \end{aligned}$$

and the following precondition $\mathsf{d}_{\mathsf{pre}}$ specifies a sufficient condition for termination.

$$\begin{aligned} \mathsf{d}_{\mathsf{pre}}(a,n) &= \mathsf{let}\ a = 0\ \mathsf{in}\ \mathsf{dl}_{\mathsf{pre}}(a,n) \\ \mathsf{dl}_{\mathsf{pre}}(a,n) &= (n \neq 0 \Rightarrow \mathsf{let}\ a = a+1\ \mathsf{in} \\ &\qquad\qquad\qquad\ \mathsf{let}\ n = n-2\ \mathsf{in} \\ &\qquad\qquad\qquad\ \mathsf{dl}_{\mathsf{pre}}(a,n)) \end{aligned}$$

One can prove, by a straight-forward induction on n (a 4-line HOL4 proof), the following property of the loop dl and its precondition $\mathsf{dl}_{\mathsf{pre}}$,

$$\forall n\ a.\ \ 0 \leq n\ \Rightarrow\ \mathsf{dl}_{\mathsf{pre}}(a, 2\times n)\ \wedge\ (dl(a, 2\times n) = (n+a, 0))$$

from which it is easy (3 lines of HOL4) to prove termination and functional correctness of the translated function d, i.e.

$$\forall n\ a.\ \ 0 \leq n\ \wedge\ \mathsf{even}\ n\ \Rightarrow\ \mathsf{d}_{\mathsf{pre}}(a,n)\ \wedge\ (d(a,n) = (n\ \mathsf{div}\ 2, 0))$$

Let this theorem be called d_spec.

2.7.1 Relating verification proof to automatically derived theorem

The following theorem was derived when the original program (which will be referred to as d_program) was translated into recursive functions d and $\mathsf{d}_{\mathsf{pre}}$.

$$\begin{aligned} &\mathsf{d}_{\mathsf{pre}}(a,n)\ \Rightarrow \\ &\mathsf{SepTotalSpec}\ (\mathsf{var}\ \texttt{"a"}\ a\ *\ \mathsf{var}\ \texttt{"n"}\ n) \\ &\qquad\qquad \mathsf{d\_program} \\ &\qquad\qquad (\mathsf{let}\ (a,n) = \mathsf{d}(a,n)\ \mathsf{in}\ (\mathsf{var}\ \texttt{"a"}\ a\ *\ \mathsf{var}\ \texttt{"n"}\ n)) \end{aligned}$$

To justify that d_spec is a verification proof of d_program, note that d_spec implies

$$\begin{array}{l} 0 \leq n \wedge \mathsf{even}\ n \Rightarrow \\ \mathsf{SepTotalSpec}\ (\mathsf{var}\ \texttt{"a"}\ a * \mathsf{var}\ \texttt{"n"}\ n) \\ \qquad\qquad \mathsf{d\_program} \\ \qquad\qquad (\mathsf{var}\ \texttt{"a"}\ (n\ \mathsf{div}\ 2) * \mathsf{var}\ \texttt{"n"}\ 0) \end{array}$$

which by expansion of various definitions is equivalent to the following statement:

$$\begin{array}{ll} \forall s_1\ s_2. & 0 \leq s_1(\texttt{"n"}) \wedge \mathsf{even}\ s_1(\texttt{"n"}) \wedge \{\texttt{"a"}, \texttt{"n"}\} \subseteq \mathsf{domain}\ s_1 \Rightarrow \\ & (\exists s_2.\ \mathsf{eval}\ s_1\ s_2\ \mathsf{d\_program}) \wedge \\ & (\forall s_2.\ \mathsf{eval}\ s_1\ s_2\ \mathsf{d\_program} \Rightarrow (s_2 = s_1[\texttt{"a"} \mapsto (s_1(\texttt{"n"})\ \mathsf{div}\ 2)][\texttt{"n"} \mapsto 0])) \end{array}$$

Thus, the user need only prove d_spec in order to imply termination and functional correctness of the original imperative program d_program.

2.8 Comparing different approaches

The previous section presented an example verification using this new method of "deductive translation into recursive functions". This section compares the proof above with well-established techniques based on verification condition generators (VCGs), reasoning directly using a Hoare logic, and direct ad hoc proofs on top of the semantics eval.

2.8.1 Using a verification condition generator

To verify a program using a VCG, one starts by annotating the code with assertions. For the program above (d_program) one needs to invent a precondition, a postcondition and, for the loop, an invariant and a variant (the result is to be a total-correctness specification). Here n is logical variable (the initial values of variable "n").

$$\begin{array}{rcl} \mathsf{pre}\ n & = & \lambda s.\ (s(\texttt{"n"}) = n) \wedge \mathsf{even}\ n \wedge 0 \leq n \\ \mathsf{post}\ n & = & \lambda s.\ (s(\texttt{"a"}) = n\ \mathsf{div}\ 2) \wedge (s(\texttt{"n"}) = 0) \\ \mathsf{inv}\ n & = & \lambda s.\ (n = 2 \times s(\texttt{"a"}) + s(\texttt{"n"})) \wedge \mathsf{even}\ s(\texttt{"n"}) \wedge 0 \leq s(\texttt{"n"}) \\ \mathsf{variant} & = & \lambda s.\ s(\texttt{"n"}) \end{array}$$

The program with the annotations:

```
{ pre n }
a := 0;
while (n ≠ 0) do  { inv n } [ variant ]
  a := a + 1;
  n := n - 2
end
{ post n }
```

A VCG produces the following verification conditions:

$$\forall n\ s.\ \ \mathsf{pre}\ n\ (s[\texttt{"a"} \mapsto 0]) \Rightarrow \mathsf{inv}\ n\ s \wedge 0 \leq \mathsf{variant}\ s$$

$$\forall n\ s.\ \ \mathsf{let}\ s' = s[\texttt{"n"} \mapsto s(\texttt{"n"}) - 2][\texttt{"a"} \mapsto s(\texttt{"a"}) + 1]\ \mathsf{in}$$
$$(\mathsf{inv}\ n\ s \wedge (s(\texttt{"n"}) \neq 0) \Rightarrow \mathsf{inv}\ n\ s' \wedge \mathsf{variant}\ s' < \mathsf{variant}\ s)$$

$$\forall n\ s.\ \ \mathsf{inv}\ n\ s \wedge (s(\texttt{"n"}) = 0) \Rightarrow \mathsf{post}\ n\ s$$

$$\forall n\ s.\ \ \mathsf{inv}\ n\ s \Rightarrow 0 \leq \mathsf{variant}\ s$$

If these verification conditions are proved by the user, in some way, then the following total-correctness theorem is true:

$$\forall n.\ \ \mathsf{TotalSpec}\ (\mathsf{pre}\ n)\ \mathsf{d\_program}\ (\mathsf{post}\ n)$$

Some noteworthy difference between the proposed approach and the VCG approach:

1. the VCG proof requires more user input: stating the assertions for the VCG proof requires more writing than the proof goals stated in the previous section;

2. the VCG proof requires the user to invent an invariant expression describing the relationship between the intermediate values and the initial value n,

   $$n = 2 \times s(\texttt{"a"}) + s(\texttt{"n"})$$

   while the proof by induction, from the previous section, only required stating the desired result of executing the remaining part of the loop:

   $$\mathsf{dl}(a, 2 \times n) = (n + a, 0)$$

3. the VCG proof uses a variant where the previous section uses an induction;

4. the VCG proof deals directly with the state s, while the verification proof from the previous section only concerns logical variables (and is, as a result, reusable for similar code based on a different definitions of 'state').

And finally, a difference which is more to do with convention rather than a real limitation of the VCG approach:

5. the VCG approach led to a statement which does not say anything about variables other than `"a"` and `"n"`; although it is obvious, by looking at the program, that variables not occurring in the program, say `"b"` and `"m"`, are left untouched. The definitions pre, post and inv, which were defined in a conventional manner, would need to be augmented with an explicit frame assertion, e.g. relating values of state s to some initial state s_0:

   $$\lambda s.\ \ \ldots\ \wedge\ \forall v.\ v \notin \{\texttt{"n"}, \texttt{"a"}\} \Rightarrow (s(v) = s_0(v))$$

   In contrast, the approach proposed here produces, by default, theorems which state that 'nothing else changed'.

The example program d_program intentionally coincides with the example Moore uses to illustrate his approach to verification using a VCG [79]. Moore suggests that the user defines a function halfa (which is essentially the same as d from Section 2.7) and that the user writes annotations which state that halfa is executed by the program he aims to verify. He proves using a VCG that his program executes halfa. In the approach presented here, 'halfa' is derived completely automatically and then the user went further to prove a non-recursive specification for the original program.

2.8.2 Using a Hoare logic directly

The VCG method automates Hoare logic. Doing VCG-style proofs without a trusted VCG is labour intensive. This section demonstrates how d_program, from Section 2.7, can be proved manually using Hoare logic. Definitions pre, post, inv and variant from above are used in this section and it is assumed that the verification conditions from above have been proved.

The manual Hoare-logic proof will make use of the following five proof rules: assignment, precondition strengthening, postcondition weakening, sequence and while.

$$\mathsf{TotalSpec}\ (\lambda s.\ p\ (s[v \mapsto (\mathsf{neval}\ e\ s)])\ (\mathsf{Assign}\ v\ e)\ p$$

$$(\forall s.\ p'\ s \Rightarrow p\ s) \wedge \mathsf{TotalSpec}\ p\ c\ q \Rightarrow \mathsf{TotalSpec}\ p'\ c\ q$$

$$(\forall s.\ q\ s \Rightarrow q'\ s) \wedge \mathsf{TotalSpec}\ p\ c\ q \Rightarrow \mathsf{TotalSpec}\ p\ c\ q'$$

$$\mathsf{TotalSpec}\ p\ c_1\ q \wedge \mathsf{TotalSpec}\ q\ c_2\ r \Rightarrow \mathsf{TotalSpec}\ p\ (\mathsf{Seq}\ c_1\ c_2)\ r$$

$$\begin{array}{l}(\forall s.\ i\ s \Rightarrow 0 \leq f(s)) \wedge \\ (\forall v.\ \mathsf{TotalSpec}\ (\lambda s.\ i\ s \wedge \mathsf{beval}\ b\ s \wedge (f(s) = v))\ c\ (\lambda s.\ i\ s \wedge (f(s) < v))) \Rightarrow \\ \mathsf{TotalSpec}\ i\ (\mathsf{While}\ b\ c)\ (\lambda s.\ i\ s \wedge \neg(\mathsf{beval}\ b\ s))\end{array}$$

Appropriate instantiations of the assignment rule and sequence rule give:

$$\mathsf{TotalSpec}\ (\lambda s.\ \mathsf{inv}\ n\ (s[\texttt{"a"} \mapsto 0]))\ (\mathsf{Assign}\ \texttt{"a"}\ (\mathsf{Const}\ 0))\ (\mathsf{inv}\ n)$$

$$\begin{array}{l}\mathsf{TotalSpec}\ (\lambda s.\ \mathsf{let}\ s' = s[\texttt{"n"} \mapsto s(\texttt{"n"}) - 2][\texttt{"a"} \mapsto s(\texttt{"a"}) + 1]\ \mathsf{in}\ (\mathsf{inv}\ n\ s' \wedge (\mathsf{variant}\ s' = v))) \\ \quad (\mathsf{Seq}\ (\mathsf{Assign}\ \texttt{"a"}\ (\mathsf{Plus}\ (\mathsf{Var}\ \texttt{"a"})\ (\mathsf{Const}\ 1))) \\ \qquad (\mathsf{Assign}\ \texttt{"n"}\ (\mathsf{Sub}\ (\mathsf{Var}\ \texttt{"n"})\ (\mathsf{Const}\ 2))))) \\ \quad (\lambda s.\ \mathsf{inv}\ n\ s \wedge (\mathsf{variant}\ s = v))\end{array}$$

Assuming that the second verification condition from above has been proved, the while rule can be applied to the above theorem to prove the following:

$$\begin{array}{l}\mathsf{TotalSpec}\ (\mathsf{inv}\ n) \\ \quad (\mathsf{While}\ (\mathsf{Not}\ (\mathsf{Equal}\ (\mathsf{Var}\ \texttt{"n"})\ (\mathsf{Const}\ 0))) \\ \qquad (\mathsf{Seq}\ (\mathsf{Assign}\ \texttt{"a"}\ (\mathsf{Plus}\ (\mathsf{Var}\ \texttt{"a"})\ (\mathsf{Const}\ 1))) \\ \qquad\quad (\mathsf{Assign}\ \texttt{"n"}\ (\mathsf{Sub}\ (\mathsf{Var}\ \texttt{"n"})\ (\mathsf{Const}\ 2))))) \\ \quad (\lambda s.\ \mathsf{inv}\ n\ s \wedge (s(\texttt{"n"}) = 0))\end{array}$$

which by application of the sequence rule followed by precondition strengthening and post-condition weakening proves:

$$\forall n.\ \ \mathsf{TotalSpec}\ (\mathsf{pre}\ n)\ \mathsf{d\_program}\ (\mathsf{post}\ n)$$

In contrast, the proposed approach of translation-then-verification automates many mundane details and delivers to the verifier a function characterising the computation, which is the essence of the problem.

2.8.3 Using the relational semantics directly

It is, of course, possible to prove the desired property manually using only the definition of eval without any of the infrastructure for Hoare logic, VCGs or translation into functions.

$$\begin{aligned}\forall s_1\ s_2.\quad & 0 \leq s_1(\texttt{"n"}) \wedge \mathsf{even}\ s_1(\texttt{"n"}) \wedge \{\texttt{"a"}, \texttt{"n"}\} \subseteq \mathsf{domain}\ s_1 \Rightarrow \\ & (\exists s_2.\ \mathsf{eval}\ s_1\ s_2\ \mathsf{d\_program}) \wedge \\ & (\forall s_2.\ \mathsf{eval}\ s_1\ s_2\ \mathsf{d\_program} \Rightarrow (s_2 = s_1[\texttt{"a"} \mapsto (s_1(\texttt{"n"})\ \mathsf{div}\ 2)][\texttt{"n"} \mapsto 0]))\end{aligned}$$

The manual proof would have a similar structure to the proof described in Section 2.7; induction would, in particular, be used in a similar way to unroll the loop. However, each step of the proof would be much more verbose and the resulting proof would be completely tied to the definition of 'state'.

Reasoning directly at the level of operational semantics will not work for detailed machine-code proofs, which is the topic of the subsequent chapters. Detailed operational semantics of machine code are too verbose to deal with manually.

2.9 Discussion of related work

The previous section compared this chapter's approach to program verification with approaches based on reasoning using the operational semantics directly, using a Hoare logic manually and VCGs based approaches. Hoare logics are reasonably easily implemented in a theorem prover, but tend to be labour intensive to use manually. Trustworthy verification condition generators are, on the other hand, easier to use, but harder to implement [52, 79, 70]. Performing manual proofs directly on top of the operational semantics is possible, however such proofs do not scale well to more complicated operational semantics. In contrast, the method presented here seems to requires less user input, provides stronger specifications and is readily implementable. The automatic translator has been implemented as a 400-line ML program.

Representing imperative programs as recursive functions, due to McCarthy [73], was key in this work. Ideas from separation logic [105] were used to aid automation by keeping specifications free of side-conditions; its frame rule made it possible to implicitly state that 'nothing else changed'. The HOL4 theorem prover [110] was used as the programming environment: the ML program automatically steers HOL4 to produce the extracted functions.

The net effect of the presented translation into functional programs bears some resemblance to automation developed by Filliâtre for verification of imperative programs using the Coq proof assistant [37]. Filliâtre's approach differs in that it requires the user to annotate programs with invariants before translation can be performed. In contrast, the approach presented here is fully automatic and requires no annotations of original programs. However, the comparison is not completely fair as Filliâtre considers more complex program constructs than those used here, Section 2.2.

Chapter 3

Specifications for realistically modelled machine instructions

This chapter presents a programming logic that has been designed to fit on top of accurate models of machine languages. In contrast to previous work, this logic is simultaneously applicable to multiple, detailed, off-the-shelf models of instruction set architectures, yielding concise total-correctness specifications for ARM, PowerPC and x86 machine code. The logic developed in this chapter will be used in Chapter 4 to adapt automation from Chapter 2 to deal with machine code.

3.1 Introduction

Computer programs execute on processor hardware such as x86, ARM, MIPS, PowerPC and Motorola. Program verification is, almost without exception, based on highly simplified models which, for example, assume that numbers can be arbitrarily large, stacks are unbounded and compilers keep code and data separate – assumptions that are not true for modern hardware.

Reasoning down to the level of real machine code introduces several mathematically unclean features that are not present in simplified high-level languages.

1. *On real machines all types are bounded*: memory, stacks and even integers are bounded; as a consequence programs cannot assume an arbitrarily large stack. Furthermore some familiar arithmetic properties do not hold for machine integers, e.g. it is not the case that $\forall v\ w.\ 0 \leq w \Rightarrow v \leq v + w$, if v and w are machine integers.

2. *Machine code operates over a heterogeneous state*, which consists of various machine-specific registers, status bits/flags, special-purpose registers, memory segments and operation modes, instead of a straight-forward mapping from variable names to values.[1]

[1] Compare for instance with the 'standard' definition of program semantics eval from Chapter 2.

3. *Machine code is generally less structured than high-level languages*:

   (a) Individual instructions rarely execute a single assignment; most instructions update registers as well as status bits and possibly also make memory accesses.

   (b) Code and data live in the same memory, meaning that programs can accidentally (or intentionally) rewrite themselves.[2]

   (c) Control flow is determined by updates to a register (the program counter), which holds the address of the next instruction to be executed.

Programming at the machine-code level is not to be encouraged as compilers generally produce better and more maintainable code than humans. However, compilers are complex programs that apply multiple optimisations; hence their correctness cannot be taken for granted. Therefore, proof of program correctness need to be mapped down to the machine-code level.

This chapter defines a machine-code Hoare triple which lifts reasoning from the tedious, formal definition of a machine language to a manageable level on top of which automation can be developed. The method makes concise specifications of functional behaviour and termination as well as resource usage. A novel aspect of this work is that the Hoare triple simultaneously fits on top of multiple detailed models. This chapter presents instantiations to publicly available specifications of ARM, PowerPC and x86, as written by Fox [40], Leroy [63] and Sarkar [30], respectively.

Some examples of manual proofs using inference rules, proved from the definition of the Hoare triple, are presented towards the end of this chapter. Subsequent chapters will develop automatic proof tools based on these definitions and inference rules.

3.2 Interface to detailed processor models

Before describing the Hoare triple for machine code, a note is made on where the detailed instruction set architecture (ISA) models are from and how the interface to them is set up.

3.2.1 Processor models

The machine-language models used in this work are publicly available models that attempt to accurately capture the behaviour of the machine instructions running in user mode.

- Fox [40] has written the ARM model, which is a detailed specification of all ARM instructions in version 4 of the architecture. Fox proved the model correct with respect to a register-transfer level specification of an ARM processor, and has performed some tests of the model by executing, inside HOL4, ARM code implementing cryptographic algorithms; the results of these evaluations performed within HOL4 correspond to the 'correct' result vectors supplied by the company who wrote the code.

[2] Most modern operating systems will raise an exception in certain cases of unintended self-modification.

- Leroy [63] has developed a specification of a large subset of PowerPC instructions for the proof of a optimising C compiler in Coq. This PowerPC specification was translated manually into HOL4 and an instruction decoder was attached to the specification in order to transform Leroy's assembly level specification into a machine-code model. Leroy has tested the output of his compiler on real hardware, which gives us confidence that his Coq model of PowerPC assembly is reasonably accurate. However, the HOL4 translation has not yet been compared with real hardware executions.
- The x86 specification is a functional HOL4 version of Sarkar's x86 specification [30], developed originally in Twelf for use in applications of proof-carrying code [92]. The HOL4 version covers a subset of the 32-bit x86 architecture, comparable in size (number of modelled instructions) to Leroy's PowerPC specification. The HOL4 model of x86 has been tested extensively, one instruction at a time, against an Intel processor implementing a 32-bit x86 architecture.

None of the machine language specifications model exceptions, interrupts or page tables. Instead these models attempt to capture the behaviour of the respective platforms as observed by user-mode programs running inside an operating system which deals with hardware interfaces and context switching between different programs.

All the models, which this work is based on, are freely available with HOL4 [110].

3.2.2 Read and write interface

Processors have multiple operation modes and registers that are invisible to certain operation modes. Read- and write-functions were defined to access the "programmer's view" of the state. For ARM, these read/write functions are the following. Memory is accessed using 30-bit addresses. Registers and memory locations hold 32-bit values.

$$
\begin{array}{ll}
\mathsf{arm\_read\_reg} & : \mathsf{reg\_name} \rightarrow \mathsf{arm\_state} \rightarrow \mathsf{word}_{32} \\
\mathsf{arm\_read\_status} & : \mathsf{status\_name} \rightarrow \mathsf{arm\_state} \rightarrow \mathsf{boolean} \\
\mathsf{arm\_read\_mem} & : \mathsf{word}_{30} \rightarrow \mathsf{arm\_state} \rightarrow \mathsf{word}_{32}\ \mathsf{option} \\
\mathsf{arm\_read\_undefined} & : \mathsf{arm\_state} \rightarrow \mathsf{boolean} \\
 & \\
\mathsf{arm\_write\_reg} & : \mathsf{reg\_name} \rightarrow \mathsf{word}_{32} \rightarrow \mathsf{arm\_state} \rightarrow \mathsf{arm\_state} \\
\mathsf{arm\_write\_status} & : \mathsf{status\_name} \rightarrow \mathsf{boolean} \rightarrow \mathsf{arm\_state} \rightarrow \mathsf{arm\_state} \\
\mathsf{arm\_write\_mem} & : \mathsf{word}_{30} \rightarrow \mathsf{word}_{32} \rightarrow \mathsf{arm\_state} \rightarrow \mathsf{arm\_state} \\
\mathsf{arm\_write\_undefined} & : \mathsf{boolean} \rightarrow \mathsf{arm\_state} \rightarrow \mathsf{arm\_state}
\end{array}
$$

These access the visible part of an ARM state, i.e. arm_read_reg r *state* returns the value of register r as observed by the current operation mode in *state*. Shadow registers and coprocessor state are inaccessible by these functions.

Functions arm_read_undefined and arm_write_undefined access an artificial state component (from Fox's model) which indicates whether the current state is an accurate representation of the behaviour of the ARM processor, i.e. arm_read_undefined *state* is true if and only if *state* accurately models the state of the ARM hardware. This means that arm_read_undefined *state* will be false after execution of a malformed or illegal instruction.

3.2.3 Next-state evaluations

Each processor model defines a next-state function (of type state → state) describing one step of the execution of machine code. For each model, a tool was written which, given a concrete encoding of an instruction, produces a theorem stating how the state is modified by the instruction in terms of the read and write functions. For instance, the ARM interface derives the following theorem about arm_next_state when given instruction `E2878001` (this is the encoding of `add r8,r7,#1`, i.e. add one to register 7 and store result in register 8). Here $w[31—2]$ is the upper 30-bits of word w.

$$
\begin{array}{l}
(\mathsf{arm\_read\_mem}\ ((\mathsf{arm\_read\_reg}\ 15\ \mathit{state})[31—2])\ \mathit{state} = \texttt{E2878001}) \wedge \\
(\mathsf{arm\_read\_undefined}\ \mathit{state} = \mathsf{F})\ \Rightarrow \\
(\mathsf{arm\_next\_state}\ \mathit{state} = \\
\quad \mathsf{arm\_write\_reg}\ 15\ (\mathsf{arm\_read\_reg}\ 15\ \mathit{state} + 4) \\
\qquad (\mathsf{arm\_write\_reg}\ 8\ (\mathsf{arm\_read\_reg}\ 7\ \mathit{state} + 1)\ \mathit{state}))
\end{array}
$$

The theorem states that, if the memory holds `E2878001` at the address stored in the upper 30 bits (selected using [31—2]) of the register 15 (the program counter), then the program counter is incremented by one instruction length (4 bytes, thus "+ 4") and register 8 is updated to contain the value of register 7 plus one.

This specification is reasonably concise because `add r8,r7,#1` does not update the status bits. However, such low-level theorems tend to become unreadable in practice as many instructions read and write multiple resource at once.

3.2.4 Translating states to a set-based representation

In subsequent sections, a set-based representation for states is used; each state is *a set of state elements*. For example, a concrete ARM state where register three has value 2, register four has value 6 and the carry status bit (bit C) has value true, written T, could be:

$$\{\ \ldots\ , \mathsf{Reg}\ 3\ 2, \mathsf{Reg}\ 4\ 6,\ \ldots\ , \mathsf{Status\ C\ T},\ \ldots\ , \mathsf{Undef\ F},\ \ldots\ \}$$

Here Reg, Status and Undef are type constructors for a data-type arm_state_element – the type of an ARM state element.

$$
\begin{array}{lcl}
\mathsf{Reg} & : & \mathsf{reg\_names} \rightarrow \mathsf{word}_{32} \rightarrow \mathsf{arm\_state\_element} \\
\mathsf{Status} & : & \mathsf{status\_names} \rightarrow \mathsf{boolean} \rightarrow \mathsf{arm\_state\_element} \\
\mathsf{Memory} & : & \mathsf{word}_{30} \rightarrow \mathsf{word}_{32}\ \mathsf{option} \rightarrow \mathsf{arm\_state\_element} \\
\mathsf{Undef} & : & \mathsf{bool} \rightarrow \mathsf{arm\_state\_element}
\end{array}
$$

A translation into the set-based format is defined for each processor model. The translation is defined using the read functions from Section 3.2.2. For ARM, a translation arm2set was defined as follows. Here range $f = \{\ y \mid \exists x.\ f\ x = y\ \}$.

$$
\begin{array}{l}
\mathsf{arm2set}\ \mathit{state} = \\
\quad \mathsf{range}\ (\lambda r.\ \mathsf{Reg}\ r\ (\mathsf{arm\_read\_reg}\ r\ \mathit{state})) \cup \\
\quad \mathsf{range}\ (\lambda a.\ \mathsf{Memory}\ a\ (\mathsf{arm\_read\_mem}\ a\ \mathit{state})) \cup \\
\quad \mathsf{range}\ (\lambda s.\ \mathsf{Status}\ s\ (\mathsf{arm\_read\_status}\ s\ \mathit{state})) \cup \\
\quad \{\ \mathsf{Undef}\ (\mathsf{arm\_read\_undefined}\ \mathit{state})\ \}
\end{array}
$$

3.3 Set-based separating conjunction

The machine-code Hoare triple, which is defined in the next section, is based on a separating conjunction. This separating conjunction is unconventional in that it splits sets of state elements instead of the previous chapter's conventional split between two functions representing states as partial mappings from names to values [105]. The conventional separating conjunction is ill-suited for machine code as processor states are essentially multiple different mappings (mappings from register names to register values, memory locations to memory values, status-bit names to bit values etc.). The collection of mappings would be different for each processor.

The set-based separating conjunction treats all resources uniformly. It splits a set (of state elements) into two sets: $p * q$ is true for set s if s can be split into two disjoint sets u and v such that p is true for u and q is true for v.

$$(p * q)\ s\ =\ \exists u\ v.\ p\ u \wedge q\ v \wedge (u \cup v = s) \wedge (u \cap v = \{\})$$

The separating conjunction $*$ is associative and commutative. Its unit is emp and angled brackets $\langle \ldots \rangle$ will be used for carrying pure boolean assertions $(\forall p\ c\ s.\ (p * \langle c \rangle)\ s = p\ s \wedge c)$:

$$\begin{array}{rcl} \mathsf{emp}\ s & = & (s = \{\}) \\ \langle b \rangle\ s & = & (s = \{\}) \wedge b \end{array}$$

This separating conjunction will mostly be used together with the translation functions such as $\mathsf{arm2set}$ from above. An example will illustrate its use, but first a few definitions. Let $\mathsf{mem}\ a\ w$ assert that memory location a contains word w and let $\mathsf{reg}\ r\ v$ assert that register r holds value v.

$$\begin{array}{rcl} (\mathsf{mem}\ a\ w)\ s & = & (s = \{\mathsf{Mem}\ a\ (\mathsf{some}\ w)\}) \\ (\mathsf{reg}\ r\ v)\ s & = & (s = \{\mathsf{Reg}\ a\ w\}) \end{array}$$

(Assertions $\mathsf{reg}\ 0\ v$, $\mathsf{reg}\ 1\ w$ etc. will most often be written as just $\mathsf{r0}\ v$ and $\mathsf{r1}\ w$ etc.)

These assertions have their intended meaning when used with $\mathsf{arm2set}$:

$$\begin{array}{l} \forall p\ t.\ (\mathsf{mem}\ a\ w * p)\ (\mathsf{arm2set}\ t) \Rightarrow (\mathsf{arm\_read\_mem}\ a\ t = w) \\ \forall p\ t.\ (\mathsf{reg}\ r\ v * p)\ (\mathsf{arm2set}\ t) \Rightarrow (\mathsf{arm\_read\_reg}\ r\ t = v) \end{array}$$

The separating conjunction separates assertions:

$$\forall p\ t.\ (\mathsf{mem}\ a\ w_1 * \mathsf{mem}\ b\ w_2 * \mathsf{reg}\ r\ w_3 * \mathsf{reg}\ d\ w_4 * p)\ (\mathsf{arm2set}\ t)\ \Rightarrow\ a \neq b\ \wedge\ r \neq d$$

The fact that $*$ separates between any resource assertions, of the same kind, is important for the frame rule of the machine-code Hoare triple, which will be defined in Section 3.4.

3.3.1 Resource assertions

The separating conjunction from above is used together with resource assertions, two of which (reg and mem) were presented above. Two similar assertions for ARM are sts and undef.

$$\begin{array}{rcl} (\mathsf{sts}\ b\ v)\ s & = & (s = \{\mathsf{Status}\ b\ v\}) \\ (\mathsf{undef}\ u)\ s & = & (s = \{\mathsf{Undef}\ u\}) \end{array}$$

Two other assertions: code c asserts for each $(a, i) \in c$ that the memory contains word i at address a; similarly, memory f asserts for each word-aligned address $a \in$ domain f (an address is word aligned if $a \,\&\, 3 = 0$) that memory location a contains $f\ a$. Here [31—2] is a function which selects the upper 30 bits of a 32-bit word (for the ARM model, the memory is a mapping from 30-bit words to 32-bit word options).

$$\begin{array}{rcl} (\mathsf{code}\ c)\ s & = & (s = \{\ \mathsf{Mem}\ (a[31\text{—}2])\ (\mathsf{some}\ i) \mid (a, i) \in c\ \}) \\ (\mathsf{memory}\ f)\ s & = & (s = \{\ \mathsf{Mem}\ (a[31\text{—}2])\ (\mathsf{some}\ (f\ a)) \mid a \in \mathsf{domain}\ f \wedge a \,\&\, 3 = 0\ \}) \end{array}$$

Throughout, m will frequently be used to abbreviate memory.

3.4 Machine-code Hoare triple

The set-based separating conjunction, from above, is used to define a machine-code version of the total-correctness Hoare triple SepTotalSpec from the previous chapter. In the previous chapter, programs were given meaning by a relation called eval. The machine-code Hoare triple will, in contrast, consider applications of some next-state function *next*. Let $\mathsf{run}(n, s)$ be a function which applies the *next* function n times to s.

$$\begin{array}{rcl} \mathsf{run}(0, s) & = & s \\ \mathsf{run}(n{+}1, s) & = & \mathsf{run}(n, next(s)) \end{array}$$

The definition of the machine-code Hoare triple for ARM is presented before giving the general definition in Section 3.7. The ARM instantiation of the machine-code Hoare triple $\{p\}\ c\ \{q\}$ states that any state s which satisfies p separately from code c and some frame r (written $p * \mathsf{code}\ c * r$), will reach (after some k applications of the next-state function) a state which satisfies q separately from the code c and frame r (written $q * \mathsf{code}\ c * r$).

$$\begin{array}{rcl} \{p\}\ c\ \{q\} & = & \forall s\ r.\ (p * \mathsf{code}\ c * r)\ (\mathsf{arm2set}(s)) \Rightarrow \\ & & \quad \exists k.\ (q * \mathsf{code}\ c * r)\ (\mathsf{arm2set}(\mathsf{run}(k, s))) \end{array}$$

As an example, the ARM instruction `add r8,r7,#1` (encoded as `E2878001`), from Section 3.2.3, has the following specification.

$$\{\ \mathsf{r7}\ x * \mathsf{r8}\ y * \mathsf{pc}\ p\ \}\ \ p : \texttt{E2878001}\ \ \{\ \mathsf{r7}\ x * \mathsf{r8}\ (x{+}1) * \mathsf{pc}\ (p{+}4)\ \}$$

Here and throughout pc p states the program counter (for ARM, register 15) holds aligned address p and that the state is not undefined. The definition of the pc assertion is for ARM: $\mathsf{pc}\ p = \mathsf{r15}\ p * \langle p \,\&\, 3 = 0 \rangle * \mathsf{undef\ F}$. Concrete code sets $\{(p, i), (q, j), \ldots\}$ are written as "$p : i,\ q : j,\ \ldots$" in order to avoid confusion with the curly brackets of the Hoare triple.

Figure 3.1 expand the Hoare triple above in order to show what such triples mean in terms of the basic read and write functions. The last line of the above expansion relating arm2set *state* to arm2set *state′* is very important. It essentially states that nothing other than registers 7, 8 and 15 observable through the read functions changed. This fact that nothing outside of the foot-print of the specification was affected, comes from the universally quantified frame r in the definition of the machine-code Hoare triple $\{p\}\ c\ \{q\}$.

$$
\begin{array}{l}
\{\,\mathsf{r7}\ x * \mathsf{r8}\ y * \mathsf{pc}\ p\,\}\ \ p : \texttt{E2878001}\ \ \{\,\mathsf{r7}\ x * \mathsf{r8}\ (x{+}1) * \mathsf{pc}\ (p{+}4)\,\} \\
= \\
\forall \mathit{state}.\quad (\mathsf{arm\_read\_reg}\ 7\ \mathit{state} = x) \wedge \\
\qquad\qquad (\mathsf{arm\_read\_reg}\ 8\ \mathit{state} = y) \wedge \\
\qquad\qquad (\mathsf{arm\_read\_reg}\ 15\ \mathit{state} = p) \wedge (p\,\&\,3 = 0) \wedge \\
\qquad\qquad (\mathsf{arm\_read\_undefined}\ 15\ \mathit{state} = \mathsf{F}) \wedge \\
\qquad\qquad (\mathsf{arm\_read\_mem}\ p\ \mathit{state} = \texttt{E2878001}) \Rightarrow \\
\qquad\qquad \exists n\ \mathit{state}'.\quad (\mathit{state}' = \mathit{run}(n, \mathit{state})) \wedge \\
\qquad\qquad\qquad\qquad (\mathsf{arm\_read\_reg}\ 7\ \mathit{state}' = x) \wedge \\
\qquad\qquad\qquad\qquad (\mathsf{arm\_read\_reg}\ 8\ \mathit{state}' = x{+}1) \wedge \\
\qquad\qquad\qquad\qquad (\mathsf{arm\_read\_reg}\ 15\ \mathit{state}' = p{+}4) \wedge ((p{+}4)\,\&\,3 = 0) \wedge \\
\qquad\qquad\qquad\qquad (\mathsf{arm\_read\_undefined}\ 15\ \mathit{state}' = \mathsf{F}) \wedge \\
\qquad\qquad\qquad\qquad (\mathsf{arm\_read\_mem}\ p\ \mathit{state}' = \texttt{E2878001}) \wedge \\
\qquad\qquad\qquad\qquad (\mathsf{arm2set}\ \mathit{state} - \mathit{Frame} = \mathsf{arm2set}\ \mathit{state}' - \mathit{Frame})
\end{array}
$$

where $\mathit{Frame} = \mathsf{range}\ (\lambda w.\ \mathsf{Reg}\ 7\ w) \cup \mathsf{range}\ (\lambda w.\ \mathsf{Reg}\ 8\ w) \cup \mathsf{range}\ (\lambda w.\ \mathsf{Reg}\ 15\ w)$

Figure 3.1: A machine-code Hoare triple expanded.

Hoare triples for instructions that read or write memory will include safety preconditions that require word alignment for word-sized accesses. For example, the following Hoare-triple theorem describes the ARM instruction for swapping the content of register 6 with a memory location pointed to by register 5. The address stored in register 5 must be word-aligned, i.e. $x\,\&\,3 = 0$. Here $m[x \mapsto y] = \lambda z.\ \mathsf{if}\ z = x\ \mathsf{then}\ y\ \mathsf{else}\ m(z)$.

$$
\begin{array}{l}
\{\,\mathsf{r5}\ x * \mathsf{r6}\ y * \mathsf{m}\ m * \mathsf{pc}\ p * \langle x\,\&\,3 = 0 \wedge x \in \mathsf{domain}\ m\rangle\,\} \\
\ p : \texttt{E1056096}\ \ \texttt{[swp r6,r6,[r5]]} \\
\{\,\mathsf{r5}\ x * \mathsf{r6}\ (m(x)) * \mathsf{m}\ (m[x \mapsto y]) * \mathsf{pc}\ (p{+}4)\,\}
\end{array}
$$

3.5 Proof rules

This section presents proof rules (theorems of higher-order logic) that have been proved from the definitions of the machine-code Hoare triple and separating conjunction given above. These rules have been proved for any machine instantiation of the Hoare triple (details in Section 3.7), although they are written simply $\{p\}\ c\ \{q\}$.

Frame: $\{p\}\ c\ \{q\}\ \Rightarrow\ \forall r.\ \{p * r\}\ c\ \{q * r\}$

The frame rule allows any assertions to be added to the pre- and postconditions, e.g. one can add $\mathsf{r3}\ z$ (register 3 has value z) to the specification of `add r8,r7,#1` from above, thus:

$$\{\,\mathsf{r3}\ z * \mathsf{r7}\ x * \mathsf{r8}\ y * \mathsf{pc}\ p\,\}\ \ p : \texttt{E2878001}\ \ \{\,\mathsf{r3}\ z * \mathsf{r7}\ x * \mathsf{r8}\ (x{+}1) * \mathsf{pc}\ (p{+}4)\,\}$$

This specification tells us that the value of register 3 is unaffected by `add r8,r7,#1`. The frame rule is most often used before an application of the composition rule.

Composition: $\{p\}\ c_1\ \{q\} \wedge \{q\}\ c_2\ \{r\}\ \Rightarrow\ \{p\}\ c_1 \cup c_2\ \{r\}$

The composition rule composes two specifications and takes the union of the two code sets (the code sets may overlap, as will happen when proving loops). Consider the following specification for `sub r7,r3,r8` (subtract `r8` from `r3` and store in `r7`, encoded as `E0437008`):

$$\{\,\mathsf{r3}\ z * \mathsf{r7}\ x * \mathsf{r8}\ y * \mathsf{pc}\ p\,\}\ \ p : \texttt{E0437008}\ \ \{\,\mathsf{r3}\ z * \mathsf{r7}\ (z{-}y) * \mathsf{r8}\ y * \mathsf{pc}\ (p{+}4)\,\}$$

The specifications for `add r8,r7,#1` and `sub r7,r3,r8` from above, can be composed to produce the following specification:

$$\{\,\mathsf{r3}\ z{*}\mathsf{r7}\ x{*}\mathsf{r8}\ y{*}\mathsf{pc}\ p\,\}\ \ p : \texttt{E28FF556},\ p{+}4 : \texttt{E0437008}\ \ \{\,\mathsf{r3}\ z{*}\mathsf{r7}\ (z{-}x{+}1){*}\mathsf{r8}\ (x{+}1){*}\mathsf{pc}\ (p{+}8)\,\}$$

Postcondition weakening: $\{p\}\ c\ \{q\} \wedge (\forall s.\ q\ s \Rightarrow r\ s)\ \Rightarrow\ \{p\}\ c\ \{r\}$

Postcondition weakening is a standard rule for discarding information, e.g. a specification which promises to assign 783 to register one, $\{p\}\ c\ \{q{*}\mathsf{r1}\ 783\}$, can be turned into one which promises to assign some non-zero value to register one, $\{p\}\ c\ \{\exists x.\ q * \mathsf{r1}\ x * \langle x \neq 0\rangle\}$.

Precondition strengthening: $\{p\}\ c\ \{q\} \wedge (\forall s.\ r\ s \Rightarrow p\ s)\ \Rightarrow\ \{r\}\ c\ \{q\}$

Precondition strengthening is the opposite of postcondition weakening: given an abstract precondition, $\{\exists x.\ p * \mathsf{r1}\ x * \langle x \neq 0\rangle\}\ c\ \{q\}$, one can use precondition strengthening to specialise the precondition, e.g. to a concrete instance $\{p{*}\mathsf{r1}\ 500\}\ c\ \{q\}$ or another abstract instance $\{\exists x.\ p * \mathsf{r1}\ x * \langle 64 \leq x\rangle\}\ c\ \{q\}$.

Precondition exists: $\{\exists x.\ p\ x\}\ c\ \{q\}\ =\ \forall x.\ \{p\ x\}\ c\ \{q\}$

Existential quantifiers in the precondition are equivalent to universal quantifiers outside of the specification, i.e. if the precondition requires that some value x exists such that the precondition holds then the specification holds for any such x:

$$\{\exists x.\ p * \mathsf{r1}\ x * \langle x \neq 0\rangle\}\ c\ \{q\}\ =\ \forall x.\ \{p * \mathsf{r1}\ x * \langle x \neq 0\rangle\}\ c\ \{q\}$$

Move pure condition: $\{p * \langle b\rangle\}\ c\ \{q\}\ =\ (b \Rightarrow \{p\}\ c\ \{q\})$

Pure conditions (written in angled brackets $\langle \ldots \rangle$, Section 3.3) can be moved into or out of the precondition of a Hoare triple, e.g. the example from above can be further developed:

$$\{\exists x.\ p * \mathsf{r1}\ x * \langle x \neq 0\rangle\}\ c\ \{q\}\ =\ \forall x.\ x \neq 0 \Rightarrow \{p * \mathsf{r1}\ x\}\ c\ \{q\}$$

Code extension: $\{p\}\ c\ \{q\}\ \Rightarrow\ \forall e.\ \{p\}\ c \cup e\ \{q\}$

This rule illustrates that the machine-code Hoare triple treats the program code differently

from the standard Hoare triple defined in the previous chapter. Here $\{p\}\ c\ \{q\}$ states that code c is sufficient to transform states satisfying p into states satisfying q. Thus any extension e to set c will also be sufficient $\{p\}\ c \cup e\ \{q\}$. This may seem unintuitive at first as one can include seemingly irrelevant instructions into specification, e.g. the specification of `add r8,r7,#1` from above can be extended to include the `sub` instruction `E0437008`, even thought the pre- and postconditions only describe the effect of the first instruction.

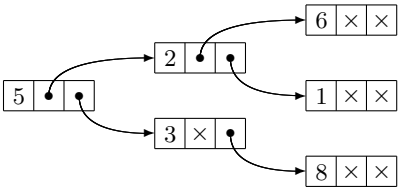
$$\{\,\mathsf{r7}\ x * \mathsf{r8}\ y * \mathsf{pc}\ p\,\}\ \ p : \texttt{E28FF556},\ p{+}4 : \texttt{E0437008}\ \{\,\mathsf{r7}\ x * \mathsf{r8}\ (x{+}1) * \mathsf{pc}\ (p{+}4)\,\}$$

The rule for code extension is a consequence of the frame rule, and is used to fill in code that is actually not used, e.g. instructions that are skipped by a branch instruction.

3.6 Verification example: recursive procedures

This section shows how the proof rules in Section 3.5 can be used manually to derive specifications for machine-code programs. The next chapter will automate proofs based on the machine-code Hoare triple.

The example developed in this section shows how to formalise and prove a specification of a recursive procedure, which calculates the sum of the numbers stored at the nodes of a binary tree. For example, the program adds $5 + 2 + 6 + 1 + 3 + 8 = 25$ to an accumulator, given the following tree.



3.6.1 Specification

Binary trees consist of branch nodes $\mathsf{Node}(x, l, r)$ that end in Leaf nodes. Each branch holds a 32-bit value x and two subtrees l and r. The sum of such a tree is defined as follows:

$$\begin{aligned} \mathsf{sum}\ (\mathsf{Leaf}) &= 0 \\ \mathsf{sum}\ (\mathsf{Node}(x, l, r)) &= x + \mathsf{sum}\ l + \mathsf{sum}\ r \end{aligned}$$

Let $\mathsf{btree}\ (x, t, m)$ state that a binary tree t is stored in memory m with its root at address x. A nil address, address 0, indicates a leaf node. Branch nodes are located at non-zero word-aligned addresses $(a \neq 0) \wedge (a\ \&\ 3 = 0)$ and consist of three consecutive 32-bit words: a data word $(a{+}0)$, and an address for the root of each subtree ($a{+}4$ and $a{+}8$).

$$\begin{aligned} \mathsf{btree}\ (a, \mathsf{Leaf}, m) &= (a = 0) \\ \mathsf{btree}\ (a, \mathsf{Node}(x, l, r), m) &= (a \neq 0) \wedge (a\ \&\ 3 = 0) \wedge \\ &\quad (m(a{+}0) = x) \wedge \mathsf{btree}\ (m(a{+}4), l, m) \wedge \mathsf{btree}\ (m(a{+}8), r, m) \end{aligned}$$

Let tree state that a binary tree is in memory:

$$\mathsf{tree}\ (x,t,m)\ =\ \mathsf{memory}\ m * \langle \mathsf{btree}\ (x,t,m)\rangle$$

One can prove, as shown in Figure 3.2, that a load instruction (`ldr r2,[r1]`, encoded as `E5912000`) followed by an add instruction (`add r0,r0,r2`, encoded as `E0800002`) add the value of a node pointed to by register 1 to an accumulator, register 0. Here the notation '$p\ \_$' from separation logic is defined by $\forall p.\ p\ \_ = \exists x.\ p\ x$.

$$\{\ \mathsf{r0}\ z * \mathsf{r1}\ x * \mathsf{r2}\ \_ * \mathsf{tree}\ (x, \mathsf{Node}(n,l,r), m) * \mathsf{pc}\ p\ \}$$
$$p : \texttt{E5912000},\ p{+}4 : \texttt{E0800002}$$
$$\{\ \mathsf{r0}\ (z{+}n) * \mathsf{r1}\ x * \mathsf{r2}\ \_ * \mathsf{tree}\ (x, \mathsf{Node}(n,l,r), m) * \mathsf{pc}\ (p{+}8)\ \}$$

The verified ARM code (`binsum`) for calculating the sum of a binary tree is listed in Figure 3.3. The `bl` instructions (branch-and-link) execute the recursive procedure calls, and the stack is used for storing local variables. The Hoare triple requires any resources that may have changed during the execution to be mentioned, thus it is necessary to specify how much stack space was used. Let $\mathsf{stack}\ (a,n)$ assert that the next n words of stack space have some value. On ARM the stack pointer is register 13 and the stack grows towards smaller addresses.

$$\begin{aligned}
\mathsf{space}\ (a,0) &= \mathsf{emp}\\
\mathsf{space}\ (a,n{+}1) &= \mathsf{mem}\ a\ _ * \mathsf{space}\ (a{-}4,n)\\
\mathsf{stack}\ (a,n) &= \mathsf{r13}\ a\ * \mathsf{space}\ (a{-}4,n)
\end{aligned}$$

The precondition of `binsum` states that code requires $2 \times \mathsf{depth}\ t$ words of stack space.

$$\mathsf{pre}\ (x,z,t,a,m) = \mathsf{r0}\ z * \mathsf{r1}\ x * \mathsf{stack}\ (a, 2 \times \mathsf{depth}\ t) * \mathsf{tree}\ (x,t,m)$$

The postcondition states that it returns the same amount of stack space, but does not guarantee anything about its content. The sum of tree t is added to register 0.

$$\mathsf{post}\ (x,z,t,a,m) = \mathsf{r0}\ (z{+}sum(t)) * \mathsf{r1}\ x * \mathsf{stack}\ (a, 2 \times \mathsf{depth}\ t) * \mathsf{tree}\ (x,t,m)$$

`binsum` has a specification of a procedure: it expects a word-aligned return address to be passed in the link-register (register 14). On exit it jumps to this return address:

$$\{\mathsf{pre}\ (x,z,t,a,m) * \mathsf{r14}\ lr * \mathsf{pc}\ p * \langle lr\ \&\ 3 = 0\rangle\}\ p : \texttt{binsum}\ \{\mathsf{post}\ (x,z,t,a,m) * \mathsf{r14}\ \_ * \mathsf{pc}\ lr\}$$

3.6.2 Proof sketch

The proof of this specification is most naturally done by induction on the structure of a binary tree. The base case is trivial as the second instruction returns control. For the step case one can assume the above specification for the subtrees of t. Suppose t' is the left subtree of t, then one can use the specification of the first `bl binsum` instruction

$$\{\ \mathsf{r14}\ x * \mathsf{pc}\ p\ \}\ \ p : \texttt{EBFFFFF7}\ \ \{\ \mathsf{r14}\ (p{+}4) * \mathsf{pc}\ (p{-}28)\ \}$$

Specification for load instruction and add instruction:

1. $\{\, \mathsf{r1}\ x * \mathsf{r2}\ y * \mathsf{memory}\ m * \mathsf{pc}\ p * \langle x \,\&\, 3 = 0 \rangle \,\}$
   p : `E5912000 [ ldr r2,[r1] ]`
   $\{\, \mathsf{r1}\ x * \mathsf{r2}\ m(x) * \mathsf{memory}\ m * \mathsf{pc}\ (p{+}4) \,\}$
2. $\{\, \mathsf{r0}\ x * \mathsf{r2}\ y * \mathsf{pc}\ p \,\}$
   p : `E0800002 [ add r0,r0,r2 ]`
   $\{\, \mathsf{r0}\ (x{+}y) * \mathsf{r2}\ y * \mathsf{pc}\ (p{+}4) \,\}$

Frame applied to specifications 1. and 2.

3. $\{\, \mathsf{r0}\ z * \mathsf{r1}\ x * \mathsf{r2}\ y * \mathsf{memory}\ m * \mathsf{pc}\ p * \langle x \,\&\, 3 = 0 \rangle \,\}$
   p : `E5912000 [ ldr r2,[r1] ]`
   $\{\, \mathsf{r0}\ z * \mathsf{r1}\ x * \mathsf{r2}\ m(x) * \mathsf{memory}\ m * \mathsf{pc}\ (p{+}4) \,\}$
4. $\{\, \mathsf{r0}\ z * \mathsf{r1}\ x * \mathsf{r2}\ m(x) * \mathsf{memory}\ m * \mathsf{pc}\ (p{+}4) \,\}$
   $p{+}4$: `E0800002 [ add r0,r0,r2 ]`
   $\{\, \mathsf{r0}\ (z{+}m(x)) * \mathsf{r1}\ x * \mathsf{r2}\ m(x) * \mathsf{memory}\ m * \mathsf{pc}\ (p{+}8) \,\}$

Specifications 3. and 4. composed:

5. $\{\, \mathsf{r0}\ z * \mathsf{r1}\ x * \mathsf{r2}\ y * \mathsf{memory}\ m * \mathsf{pc}\ p * \langle x \,\&\, 3 = 0 \rangle \,\}$
   p : `E5912000`, $p{+}4$: `E0800002 [ ldr r2,[r1] ; add r0,r0,r2 ]`
   $\{\, \mathsf{r0}\ (z{+}m(x)) * \mathsf{r1}\ x * \mathsf{r2}\ m(x) * \mathsf{memory}\ m * \mathsf{pc}\ (p{+}8) \,\}$

Assume *tree* with a branching node:

6. $\mathsf{btree}\ (x, \mathsf{Node}(n, l, r), m) \Rightarrow$
   $\{\, \mathsf{r0}\ z * \mathsf{r1}\ x * \mathsf{r2}\ y * \mathsf{memory}\ m * \mathsf{pc}\ p * \langle x \,\&\, 3 = 0 \rangle \,\}$
   p : `E5912000`, $p{+}4$: `E0800002 [ ldr r2,[r1] ; add r0,r0,r2 ]`
   $\{\, \mathsf{r0}\ (z{+}m(x)) * \mathsf{r1}\ x * \mathsf{r2}\ m(x) * \mathsf{memory}\ m * \mathsf{pc}\ (p{+}8) \,\}$

Weakening of postcondition to introduce $z{+}n$ and tree into the postcondition:

7. $\mathsf{btree}\ (x, \mathsf{Node}(n, l, r), m) \Rightarrow$
   $\{\, \mathsf{r0}\ z * \mathsf{r1}\ x * \mathsf{r2}\ y * \mathsf{memory}\ m * \mathsf{pc}\ p * \langle x \,\&\, 3 = 0 \rangle \,\}$
   p : `E5912000`, $p{+}4$: `E0800002 [ ldr r2,[r1] ; add r0,r0,r2 ]`
   $\{\, \mathsf{r0}\ (z{+}n) * \mathsf{r1}\ x * \mathsf{r2}\ \_ * \mathsf{tree}\ (x, \mathsf{Node}(n, l, r), m) * \mathsf{pc}\ (p{+}8) \,\}$

Moving assumption into precondition and introducing existential quantifier:

8. $\{\, \exists y.\ \mathsf{r0}\ z * \mathsf{r1}\ x * \mathsf{r2}\ y * \mathsf{memory}\ m * \mathsf{pc}\ p * \langle x \,\&\, 3 = 0 \rangle * \langle \mathsf{btree}\ (x, \mathsf{Node}(n, l, r), m) \rangle \,\}$
   p : `E5912000`, $p{+}4$: `E0800002 [ ldr r2,[r1] ; add r0,r0,r2 ]`
   $\{\, \mathsf{r0}\ (z{+}n) * \mathsf{r1}\ x * \mathsf{r2}\ \_ * \mathsf{tree}\ (x, \mathsf{Node}(n, l, r), m) * \mathsf{pc}\ (p{+}8) \,\}$

Strengthening precondition:

9. $\{\, \mathsf{r0}\ z * \mathsf{r1}\ x * \mathsf{r2}\ \_ * \mathsf{tree}\ (x, \mathsf{Node}(n, l, r), m) * \mathsf{pc}\ p \,\}$
   p : `E5912000`, $p{+}4$: `E0800002 [ ldr r2,[r1] ; add r0,r0,r2 ]`
   $\{\, \mathsf{r0}\ (z{+}n) * \mathsf{r1}\ x * \mathsf{r2}\ \_ * \mathsf{tree}\ (x, \mathsf{Node}(n, l, r), m) * \mathsf{pc}\ (p{+}8) \,\}$

Figure 3.2: Proving that load followed by add updates accumulator register 0.

```
 0:  binsum:  cmp r1,#0            ; compare root address (r1) with 0
 4:           moveq r15,r14        ; return, if nil
 8:           str r1,[r13,#-4]!    ; push root address, r13 := r13 - 4
12:           str r14,[r13,#-4]!   ; push return address, r13 := r13 - 4
16:           ldr r14,[r1]         ; temp := node value
20:           add r0,r0,r14        ; r0 := r0 + temp
24:           ldr r1,[r1,#4]       ; r1 := address of left subtree
28:           bl binsum            ; recursive call adds sum of left subtree to r0
32:           ldr r1,[r13,#4]      ; r1 := original r1
36:           ldr r1,[r1,#8]       ; r1 := address of right subtree
40:           bl binsum            ; recursive call adds sum of right subtree to r0
44:           ldr r15,[r13],#8     ; pop two and return
```

Figure 3.3: `binsum` sums the values at the nodes of a binary tree.

to get a specification of the effect of executing `binsum` for subtree t'. Instantiate p and weaken the postcondition to get the following. The result $(p+32)\,\&\,3 = 0$ comes from the definition of pc.

$$\{\,\mathsf{r14}\ x * \mathsf{pc}\ (p{+}28)\,\}\ \ p{+}28 : \texttt{EBFFFFF7}\ \ \{\,\mathsf{r14}\ (p{+}32) * \mathsf{pc}\ p * \langle (p{+}32)\,\&\,3 = 0\rangle\,\}$$

An application of frame and then composition produces:

$$\{\,\mathsf{pre}\ (x, z, t', a, m) * \mathsf{r14}\ lr * \mathsf{pc}\ (p{+}28)\,\}\ \ p : \texttt{binsum}\ \ \{\,\mathsf{post}\ (x, z, t', a, m) * \mathsf{r14}\ \_ * \mathsf{pc}\ (p{+}32)\,\}$$

This is a specification of the first recursive call (control travels from p+28 to p+32). However, the statement depends on the presence of the entire `binsum` code (of which the call instruction is a part and therefore disappears by set-union of the composition rule). The remainder of the proof is in spirit very similar to that listed in Figure 3.2, now that one has straightforward specifications for each instruction of `binsum`.

A variant of `binsum`, which replaces the last recursive call with a tail-call, is easily constructed by replacing the last two instructions with `ldr r14,[r13],#8` followed by `b binsum`. The verification proof is different only in very minor details.

In HOL4, the above example can either be proved using forward reasoning or goal-oriented backward reasoning. In forward reasoning, the user combines proved specifications to produce new proved specifications. For backward reasoning, the user states the theorem to be proved as a goal and then successively reduces the goal until all subgoals are proved. The more popular method of goal-oriented backward reasoning tends to require longer proof scripts as users must write intermediate assertions. Although forward reasoning is very manual to apply and the proof scripts often become unreadable, forward proofs did not require the user to write intermediate assertions. Therefore, the author preferred forward reasoning for these examples.

3.7 General definition of Hoare triple

Examples have so far only concerned ARM. However, the machine-code Hoare triple is defined more generally by parameterising it on the next-state function, the instruction function and the translation function. Let $\mathsf{funpow}(next, k, s)$ apply $next$ k-times to s, and let $\mathsf{C}\ inst\ code\ state = (state = \bigcup\{\ inst(addr, cmd) \mid (addr, cmd) \in code\ \})$

$$\begin{aligned}\mathsf{CodeSpec}\ (next, inst, trans)\ p\ c\ q\ =\ \forall s\ r.\ (p * \mathsf{C}\ inst\ c * r)\ (trans(s))\ \Rightarrow \\ \exists k.\ (q * \mathsf{C}\ inst\ c * r)\ (trans(\mathsf{funpow}(next, k, s)))\end{aligned}$$

The next-state functions are taken directly from each model. The translation functions are straight-forward adaptations of the processor specific read/write functions of each interface. The interesting part is the different instruction functions $inst$ and memory assertions memory, given for PowerPC and x86 below. For ARM, $next = \mathsf{arm\_next}$, $trans = \mathsf{arm2set}$, and $inst = \lambda(a, c).\{\mathsf{Mem}\ (a[31—2])\ (\mathsf{some}\ c)\}$.

The proof rules listed in Section 3.5 were proved for any instantiations of $(next, inst, trans)$. For example, the frame rule and code extension look as follows.

$$\begin{aligned}&\forall m\ p\ c\ q.\ \ \mathsf{CodeSpec}\ m\ p\ c\ q\ \Rightarrow\ \forall r.\ \mathsf{CodeSpec}\ m\ (p * r)\ c\ (q * r) \\ &\forall m\ p\ c\ q.\ \ \mathsf{CodeSpec}\ m\ p\ c\ q\ \Rightarrow\ \forall e.\ \mathsf{CodeSpec}\ m\ p\ (c \cup e)\ q\end{aligned}$$

3.7.1 PowerPC instantiation

The instruction functions for PowerPC and x86 are slightly more complicated than that for ARM due to the fact that their set representation (and the underlying model) considers the memory as byte-addressed, i.e. a 32-bit word consists of four bytes. The instruction function and memory assertion for PowerPC (which is a big-endian architecture) is as follows:

$$\begin{aligned}\mathsf{ppc\_word}\ (p, w)\ &=\ \{\ \mathsf{pMem}\ (p{+}0)\ (\mathsf{some}\ (w[31{-}24])), \\ &\qquad \mathsf{pMem}\ (p{+}1)\ (\mathsf{some}\ (w[23{-}16])), \\ &\qquad \mathsf{pMem}\ (p{+}2)\ (\mathsf{some}\ (w[15{-}08])), \\ &\qquad \mathsf{pMem}\ (p{+}3)\ (\mathsf{some}\ (w[07{-}00]))\ \} \\ \mathsf{ppc\_inst}\ (a, c)\ &=\ \mathsf{ppc\_word}\ (a, c) \\ \mathsf{ppc\_memory}\ m\ s\ &=\ (s = \bigcup\{\ \mathsf{ppc\_word}\ (a, m(a)) \mid a \in \mathsf{domain}\ m\ \wedge a\,\&\,3 = 0\ \})\end{aligned}$$

3.7.2 x86 instantiation

For x86 the instruction function is defined recursively, as x86 instructions are lists of bytes:

$$\begin{aligned}\mathsf{x86\_inst}\ (p, [])\ &=\ \{\} \\ \mathsf{x86\_inst}\ (p, c{::}cs)\ &=\ \{\ \mathsf{xMem}\ p\ (\mathsf{some}\ c)\ \} \cup \mathsf{x86\_inst}\ (p{+}1, cs)\end{aligned}$$

x86's memory assertion takes into account that the architecture is little-endian:

$$\mathsf{x86\_word}\ (p, w)\ =\ \{\ \mathsf{xMem}\ (p{+}0)\ (\mathsf{some}\ (w[07{-}00])),$$

$$\begin{array}{rcl} & & \mathsf{xMem}\ (p{+}1)\ (\mathsf{some}\ (w[15{-}08])), \\ & & \mathsf{xMem}\ (p{+}2)\ (\mathsf{some}\ (w[23{-}16])), \\ & & \mathsf{xMem}\ (p{+}3)\ (\mathsf{some}\ (w[31{-}24]))\ \} \\ \mathsf{x86\_memory}\ m\ s & = & (s = \bigcup \{\ \mathsf{x86\_word}\ (a, m(a)) \mid a \in \mathsf{domain}\ m \wedge a\,\&\,3 = 0\ \}) \end{array}$$

3.8 Discussion of related work

Arbib and Alagic [2] appear to have been the first to extend Hoare logic to handle unstructured control flow of assembly programs. Inspired by proof-carrying code [92], Saabas and Uustalo [108] and Tan and Appel [112] define Hoare triples which allow multiple pre- and postconditions. Saabas and Uustalo argue that multiple pre- and postconditions are necessary for 'natural' Hoare-logic-like reasoning at low levels of abstraction. This chapter defines Hoare triples with only one pre- and one postcondition, but can still reason 'naturally' about multiple-entry and multiple-exit machine code, since updates to the program counter are made explicit. For example, a branch on the value of status bit Z ($\mathsf{sz}\ z$) is described by a theorem with an if-statement in the postcondition:

$$\{\ \mathsf{sz}\ z * \mathsf{pc}\ p\ \}\ \ p : \texttt{FBFFFF1A}\ \ \{\ \mathsf{sz}\ z * \mathsf{pc}\ (\mathsf{if}\ z\ \mathsf{then}\ p{+}4\ \mathsf{else}\ p{-}12)\ \}$$

Shao's group at the University of Yale has constructed many different Hoare logics to handle various aspects of low-level code, including embedded code pointers [95], context switching [96], memory management [76], self-modifying code [18] and hardware interrupts [36]. Each Hoare logic is applied to only a single machine language (idealised MIPS or x86). In contrast, this chapter presented a general-purpose Hoare logic geared to fit on top of multiple models of commercial machine languages, while still possibly being a strong enough framework to handle some of the features for which they have defined specialised programming logics. For example, support for embedded code pointers requires being able to nest Hoare triples and jump to code pointed to in data, both of which can be done with the Hoare triples presented in this chapter:

$$\begin{array}{rl} & \{q_1 * \mathsf{pc}\ p_1\}\ c_1\ \{q_2 * \mathsf{pc}\ p_2 * \langle\{q_2 * \mathsf{pc}\ p_2\}\ c_2\ \{q_3 * \mathsf{pc}\ p_3\}\rangle\} \\ \Rightarrow & \{q_1 * \mathsf{pc}\ p_1\}\ c_1 \cup c_2\ \{q_3 * \mathsf{pc}\ p_3\} \end{array}$$

Similarly, reasoning about self-modifying code may also work, since code is just data separated by a separating conjunction, as can be seen by the following theorem.[3] Here $\emptyset$ means empty set, i.e. in this case that the set of instructions is empty.

$$\forall p\ c\ q.\quad \{p\}\ c\ \{q\}\ =\ \{p * \mathsf{code}\ c\}\ \emptyset\ \{q * \mathsf{code}\ c\}$$

Remember that the code assertion is an instance of the normal memory assertion memory or m, Section 3.3.1.

[3] However, the underlying instruction-set specifications ought to be updated to include some notion of instruction cache before verification of self-modifying code is considered.

Chapter 4

Decompilation into logic

This chapter describes a novel method for machine-code verification: machine code is decompiled into tail-recursive functions and then verification proofs are performed in the native language of a theorem prover. Unlike established methods, decompilation allows proof reuse even between different instruction architectures. Fully automatic reverse engineering of machine code is achieved by combining the machine-code Hoare triple of Chapter 3 with the idea of transforming programs into recursive functions from Chapter 2.

4.1 Introduction

Machine-code programs operate over a complex state at a very low level of abstraction. Proving properties of such programs is, as a result, a labour intensive task. This chapter tackles the challenge of making machine-code verification tractable:

A: without introducing simplifying assumptions, and

B: not requiring expert knowledge of the underlying detailed machine models, while still

C: allowing reuse of proofs between different architectures.

Current approaches struggle to address challenge *C*, as they either involve direct reasoning about the next-state function [16], or are based on annotating the code with assertions [50, 70]. Annotating the code with assertions inevitably ties the proof to the specific code and machine model as assertions are mixed with the code and depend on machine-specific resource names.

This chapter describes a new method which adds a thin layer of abstraction to the verification process in order to make verification proofs tractable and reusable. A fully automatic decompiler is presented, which translates machine code, via automatic deduction, into tail-recursive functions defined in the language of a theorem prover. The automation is a result

of combining ideas of transforming programs into recursive functions from Chapter 2 with the machine-code Hoare triple of Chapter 3.

Given a sequence of machine-code instructions, the decompiler derives a tail-recursive function and proves a theorem stating that the function accurately describes the effect of the given machine code (this addresses challenge A). The user can concentrate on proving properties of the generated function, so irrelevant details of the underlying machine language specification are hidden (challenge B). Properties proved about the generated function are, via an automatically derived theorem, related to the execution of the original machine code. The function describes the executed low-level algorithm and is likely to be similar (illustrated in Section 4.2.3) to another function describing the same algorithm implemented in a different machine language. This can facilitate reuse of verification proofs (challenge C).

Notation. Program specifications are written here as *Hoare triples* $\{p\}\, c\, \{q\}$ with informal meaning: if p holds for the current state then execution of code c will leave the process in a state satisfying q; the formal definition was given in the previous chapter, Section 3.4.

4.2 Example

This section shows how decompilation aids verification. Subsequent sections describe the decompilation algorithm.

4.2.1 Running the automation

Consider the following ARM machine code (with corresponding assembly code shown to the right) for calculating the length of a linked list. The code sets register 0 to zero; it then compares register 1 (the list pointer) with zero (nil). The last three instructions execute conditionally based on the result of this comparison: if register 1 is not zero, then the last three instructions increment register 0, load register 1 from memory and jump back to the compare instruction, otherwise the last three instructions do nothing.

```
 0:  E3A00000          mov r0, #0
 4:  E3510000       L: cmp r1, #0
 8:  12800001          addne r0, r0, #1
12:  15911000          ldrne r1, [r1]
16:  1AFFFFFB          bne L
```

Given the above list of hexadecimal numbers, our decompiler produces a function f describing the effect of the code.

$$
\begin{aligned}
\mathsf{f}(r_0, r_1, m) &= \mathsf{let}\ r_0 = 0\ \mathsf{in}\ \mathsf{g}(r_0, r_1, m) \\
\mathsf{g}(r_0, r_1, m) &= \mathsf{if}\ r_1 = 0\ \mathsf{then}\ (r_0, r_1, m)\ \mathsf{else} \\
&\qquad \mathsf{let}\ r_0 = r_0{+}1\ \mathsf{in} \\
&\qquad \mathsf{let}\ r_1 = m(r_1)\ \mathsf{in} \\
&\qquad\quad \mathsf{g}(r_0, r_1, m)
\end{aligned}
$$

The decompiler also automatically proves the following theorem, stated as a Hoare triple, relating the execution of the ARM code with the function f (and an automatically generated precondition $\mathsf{f}_{\mathsf{pre}}$, given in Section 4.3.5). The Hoare-triple specification (defined in Section 3.4) can be read informally as follows: given a state where register 0, register 1, and a part of memory is described by (r_0, r_1, m), the program counter is p and precondition $\mathsf{f}_{\mathsf{pre}}$ holds, then executing the code will leave the processor in a state where register 0, register 1, a part of memory is described by $\mathsf{f}\ (r_0, r_1, m)$ and the program counter is $p + 20$. Here, and throughout, $(k_1, k_2, ..., k_n)$ is $(x_1, x_2, ..., x_n)$ abbreviates $k_1\ x_1 * k_2\ x_2 * ... * k_n\ x_n$.

$$\begin{array}{l} \{\ (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ (r_0, r_1, m) * \mathsf{pc}\ p * \langle \mathsf{f}_{\mathsf{pre}}(r_0, r_1, m)\rangle\ \} \\ p : \texttt{E3A00000}, \texttt{E3510000}, \texttt{12800001}, \texttt{15911000}, \texttt{1AFFFFFB} \\ \{\ (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ (\mathsf{f}\ (r_0, r_1, m)) * \mathsf{pc}\ (p + 20)\ \} \end{array} \tag{4.1}$$

The precondition $\mathsf{f}_{\mathsf{pre}}$ collects the side-conditions which must hold for f to execute properly. Each time the above load instruction is encountered, register 1 must contain a word-aligned address in the domain of m:

$$\begin{array}{rcl} \mathsf{f}_{\mathsf{pre}}(r_0, r_1, m) & = & \mathsf{let}\ r_0 = 0\ \mathsf{in}\ \mathsf{g}_{\mathsf{pre}}(r_0, r_1, m) \\ \mathsf{g}_{\mathsf{pre}}(r_0, r_1, m) & = & \mathsf{if}\ r_1 = 0\ \mathsf{then}\ \mathsf{T}\ \mathsf{else} \\ & & \quad \mathsf{let}\ r_0 = r_0{+}1\ \mathsf{in} \\ & & \quad \mathsf{let}\ cond = \mathsf{aligned}\ r_1 \wedge r_1 \in \mathsf{domain}\ m\ \mathsf{in} \\ & & \quad \mathsf{let}\ r_1 = m(r_1)\ \mathsf{in} \\ & & \quad\quad \mathsf{g}_{\mathsf{pre}}(r_0, r_1, m) \wedge cond \end{array}$$

4.2.2 Verifying the code

The statement "the memory holds a linked list" needs to be formalised in order to verify that the code above computes the length of a linked list. Let $\mathsf{list}\ (l, a, m)$ be a recursively-defined predicate which states that an abstract list of 32-bit words l, e.g. $l = [4, 5] = 4{::}5{::}\mathsf{nil}$ (list cons is written here as '::'), is represented by a linked list in memory m with its head at address a. Each element of the list is represented by a word for the next pointer $m(a)$ and a word for the data $m(a{+}4)$. The words are positioned 4 bytes apart, hence "+4".

$$\begin{array}{rcl} \mathsf{list}\ (\mathsf{nil}, a, m) & = & a = 0 \\ \mathsf{list}\ (x{::}l, a, m) & = & \exists a'.\ m(a) = a' \wedge m(a{+}4) = x \wedge a \neq 0 \wedge \\ & & \quad \mathsf{list}\ (l, a', m) \wedge \mathsf{aligned}\ a \end{array}$$

Let $\mathsf{length}\ l$ be the length of an abstract list l, e.g. $\mathsf{length}\ (4{::}5{::}\mathsf{nil}) = 2$. It is now easy (15 lines of HOL4) to prove, by induction on the abstract list l, that the function f, from above, calculates the length of a linked list and also that list implies the precondition $\mathsf{f}_{\mathsf{pre}}$.

$$\forall x\, l\, a\, m. \quad \mathsf{list}\ (l, a, m) \quad \Rightarrow \quad \mathsf{f}(x, a, m) = (\mathsf{length}\ l, 0, m) \tag{4.2}$$

$$\forall x\, l\, a\, m. \quad \mathsf{list}\ (l, a, m) \quad \Rightarrow \quad \mathsf{f}_{\mathsf{pre}}(x, a, m) \tag{4.3}$$

Lemmas (4.2) and (4.3) can be used to rewrite and strengthen the automatically proved Hoare-triple certificate theorem (4.1) to state that the ARM code calculates "(length $l, 0, m$)".

$$\{\ (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ (r_0, r_1, m) * \mathsf{pc}\ p * \langle \mathsf{list}\ (l, r_1, m)\rangle\ \}$$
$$p : \texttt{E3A00000}, \texttt{E3510000}, \texttt{12800001}, \texttt{15911000}, \texttt{1AFFFFFB}$$
$$\{\ (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ (\mathsf{length}\ l, 0, m) * \mathsf{pc}\ (p + 20)\ \}$$

In this manner, the user need only prove properties of the abstract function f in order to verify properties of the ARM machine code.

4.2.3 Reusing the proof

An interesting aspect of the decompilation approach is that it can facilitates reuse of proofs, even between different architectures. To illustrate this consider the following x86 code, which uses standard x86 tricks for assigning zero to a register using `xor` (bitwise exlusive or) and testing for equality with zero using `test` (zero compared with bitwise and),

```
 0:  31C0            xor eax, eax
 2:  85F6      L1:   test esi, esi
 4:  7405            jz L2
 6:  40              inc eax
 7:  8B36            mov esi, [esi]
 9:  EBF7            jmp L1
               L2:
```

and also the following PowerPC code for calculating the length of a linked-list.

```
 0:  38A00000        addi 5,0,0
 4:  2C140000  L1:   cmpwi 20,0
 8:  40820010        bc 4,2,L2
12:  82940000        lwz 20,0(20)
16:  38A50001        addi 5,5,1
20:  4BFFFFF0        b L1
               L2:
```

Since the functional behaviour of all three code examples is essentially the same, the functions describing their behaviour are almost identical. Function fx is extracted for the x86 code and function fp is the same for PowerPC. Here '$\otimes$' denotes bitwise xor and '&' means bitwise and.

$$\begin{aligned}
\mathsf{fx}(eax, esi, m) &= \mathsf{let}\ eax = eax \otimes eax\ \mathsf{in}\ \mathsf{gx}(eax, esi, m)\\
\mathsf{gx}(eax, esi, m) &= \mathsf{if}\ esi\ \&\ esi = 0\ \mathsf{then}\ (eax, esi, m)\ \mathsf{else}\\
&\qquad \mathsf{let}\ eax = eax{+}1\ \mathsf{in}\\
&\qquad \mathsf{let}\ esi = m(esi)\ \mathsf{in}\\
&\qquad\quad \mathsf{gx}(eax, esi, m)
\end{aligned}$$

$$\begin{array}{rcl}
\mathsf{fp}(r_5, r_{20}, m) & = & \mathsf{let}\ r_5 = 0\ \mathsf{in}\ \mathsf{gp}(r_5, r_{20}, m) \\
\mathsf{gp}(r_5, r_{20}, m) & = & \mathsf{if}\ r_{20} = 0\ \mathsf{then}\ (r_5, r_{20}, m)\ \mathsf{else} \\
 & & \quad \mathsf{let}\ r_{20} = m(r_{20})\ \mathsf{in} \\
 & & \quad \mathsf{let}\ r_5 = r_5{+}1\ \mathsf{in} \\
 & & \quad\quad \mathsf{gp}(r_5, r_{20}, m)
\end{array}$$

Minor differences, such as register names, conditional execution (ARM), variable instruction length (x86), and some instruction reorderings (PowerPC example has load before increment), disappear in the functional description of the behaviour of the code. As a result the extracted functions can be proved equal by a short proof without induction, in this case a three-line HOL4 proof, using facts $w \otimes w = 0$ and $w \,\&\, w = w$.

$$\mathsf{f} = \mathsf{fx} = \mathsf{fp} \quad \text{and} \quad \mathsf{f}_{\mathsf{pre}} = \mathsf{fx}_{\mathsf{pre}} = \mathsf{fp}_{\mathsf{pre}}$$

Thus, any result proved for f and $\mathsf{f}_{\mathsf{pre}}$ also describes the x86 and PowerPC implementations. By rewriting and strengthening the certificate theorems, using (4.2) and (4.3), one immediately obtains the same specification for the x86 machine code:

$$\{\ (\mathsf{eax}, \mathsf{esi}, \mathsf{m})\ \mathsf{is}\ (eax, esi, m) * \mathsf{eip}\ p * \langle \mathsf{list}\ (l, esi, m) \rangle\ \}$$
$$p : \texttt{31C0}, \texttt{85F6}, \texttt{7405}, \texttt{40}, \texttt{8B36}, \texttt{EBF7}$$
$$\{\ (\mathsf{eax}, \mathsf{esi}, \mathsf{m})\ \mathsf{is}\ (\mathsf{length}\ l, 0, m) * \mathsf{eip}\ (p + 11)\ \}$$

and similarly for the PowerPC code:

$$\{\ (\mathsf{r5}, \mathsf{r20}, \mathsf{m})\ \mathsf{is}\ (r_5, r_{20}, m) * \mathsf{pc}\ p * \langle \mathsf{list}\ (l, r_{20}, m) \rangle\ \}$$
$$p : \texttt{38A00000}, \texttt{2C140000}, \texttt{40820010}, \texttt{82940000}, \texttt{38A50001}, \texttt{4BFFFFF0}$$
$$\{\ (\mathsf{r5}, \mathsf{r20}, \mathsf{m})\ \mathsf{is}\ (\mathsf{length}\ l, 0, m) * \mathsf{pc}\ (p + 24)\ \}$$

The decompiler automates the machine-specific proofs and delivers a recursive function describing the code. The generated functions are sufficiently abstract to be reusable while at the same time have a strong connection with the original machine code, enabling properties of the function to carry over to properties proved of the code.

4.2.4 Larger examples

The decompilation technique presented here has been applied to a number of verification examples. The most significant are two copying collectors, variants of the Cheney garbage collector, presented in the next chapter. The largest examples consist of around a hundred machine instructions.

4.3 Decompilation algorithm

The algorithm for decompilation is essentially an adaption of the *translation to recursive functions* technique described in Chapter 2. The challenges of its adaption to machine

code include: handling the more general control-flow, suppressing superfluous status bit information, and managing the many side conditions machine code reasoning requires.

The decompilation algorithm can be broken down into the following steps:

1. calculate the behaviour of each individual instruction;
2. prove a Hoare-triple, $\{...\}\, c\, \{...\}$, theorem for each instruction;
3. discover the control flow by analysing the Hoare triple theorems;
4. split the code according to the control-flow graph;
5. for each code segment:
   a) derive a theorem for one pass through the code,
   b) generate a function describing the code,
   c) apply a special loop rule;
6. compose the top-level specifications and repeat from step 5 until all of the code is described by one Hoare triple.

The following subsections explain these steps when applied to the linked-list example from Section 4.2. Later subsections describe support for procedure calls as well as non-nested loops. Restrictions of the approach are outlined in Section 4.4.

4.3.1 Behaviour of instructions

As a first step, each instruction's effect on the underlying machine-language model is evaluated. The next-state function is unrolled one step using symbolic values. As an example: the ARM instruction which assigns 0 to register 0, i.e. `mov r0, #0` encoded as `E3A00000`, evaluates to the following theorem:

$$
\begin{array}{l}
(\mathsf{arm\_read\_reg}\ 15\ \mathit{state} = p)\ \wedge \\
(\mathsf{arm\_read\_mem}\ (p[31\text{—}2])\ \mathit{state} = \mathtt{0xE3A00000})\ \wedge \\
(\mathsf{arm\_read\_undefined}\ \mathit{state} = \mathsf{F})\ \Rightarrow \\
(\mathsf{arm\_next\_state}\ \mathit{state} = \mathsf{arm\_write\_reg}\ 15\ (p+4)\ (\mathsf{arm\_write\_reg}\ 0\ 0\ \mathit{state}))
\end{array}
$$

(Remarks on the implementation of the decompiler are deferred until Section 4.4.)

4.3.2 Instruction specifications

As a second step, Hoare triple theorems are derived for each instruction in the program. The following are derived for the ARM code of the linked-list example. Conditionally executed instructions get two specifications: one for the case when the instruction is executed and one for the case when it is not.

The move instruction assigns zero to register 0:

$$\{\,\mathsf{r0}\ r_0 * \mathsf{pc}\ p\,\}$$
$$p : \texttt{E3A00000 [mov r0, \#0]}$$
$$\{\,\mathsf{r0}\ 0 * \mathsf{pc}\ (p{+}4)\,\}$$

Compare updates the status bits. Here sn, sz, sc, sv assert the value of bits N, Z, C and V.

$$\{\,\mathsf{r1}\ r_1 * \mathsf{pc}\ (p{+}4) * \mathsf{sn}\ n * \mathsf{sz}\ z * \mathsf{sc}\ c * \mathsf{sv}\ v\,\}$$
$$p{+}4 : \texttt{E3510000 [cmp r1, \#0]}$$
$$\{\,\mathsf{r1}\ r_1 * \mathsf{pc}\ (p{+}8) * \mathsf{sn}\ (2^{31} < r_1) * \mathsf{sz}\ (r_1 = 0) * \mathsf{sc}\ \mathsf{T} * \mathsf{sv}\ \mathsf{F}\,\}$$

The add instruction increments register 0 if status bit Z is false.

$$\{\,\mathsf{r0}\ r_0 * \mathsf{pc}\ (p{+}8) * \mathsf{sz}\ z * \langle \neg z \rangle\,\}$$
$$p{+}8 : \texttt{12800001 [addne r0, r0, \#1]}$$
$$\{\,\mathsf{r0}\ (r_0{+}1) * \mathsf{pc}\ (p{+}12) * \mathsf{sz}\ z\,\}$$

$$\{\,\mathsf{pc}\ (p{+}8) * \mathsf{sz}\ z * \langle z \rangle\,\}$$
$$p{+}8 : \texttt{12800001 [addne r0, r0, \#1]}$$
$$\{\,\mathsf{pc}\ (p{+}12) * \mathsf{sz}\ z\,\}$$

The load instruction loads a value into register 1 if status bit Z is false.

$$\{\,\mathsf{r1}\ r_1 * \mathsf{m}\ m * \mathsf{pc}\ (p{+}12) * \mathsf{sz}\ z * \langle \neg z \wedge r_1 \in \mathsf{domain}\ m \wedge \mathsf{aligned}\ r_1 \rangle\,\}$$
$$p{+}12 : \texttt{15911000 [ldrne r1, [r1]]}$$
$$\{\,\mathsf{r1}\ m(r_1) * \mathsf{m}\ m * \mathsf{pc}\ (p{+}16) * \mathsf{sz}\ z\,\}$$

$$\{\,\mathsf{pc}\ (p{+}12) * \mathsf{sz}\ z * \langle z \rangle\,\}$$
$$p{+}12 : \texttt{15911000 [ldrne r1, [r1]]}$$
$$\{\,\mathsf{pc}\ (p{+}16) * \mathsf{sz}\ z\,\}$$

The branch instruction jumps backwards in the code if status bit Z is false.

$$\{\,\mathsf{pc}\ (p{+}16) * \mathsf{sz}\ z * \langle \neg z \rangle\,\}$$
$$p{+}16 : \texttt{1AFFFFFB [bne L]}$$
$$\{\,\mathsf{pc}\ (p{+}4) * \mathsf{sz}\ z\,\}$$

$$\{\,\mathsf{pc}\ (p{+}16) * \mathsf{sz}\ z * \langle z \rangle\,\}$$
$$p{+}16 : \texttt{1AFFFFFB [bne L]}$$
$$\{\,\mathsf{pc}\ (p{+}20) * \mathsf{sz}\ z\,\}$$

Post-processing. A minor reformulation is applied to the theorems above in order for the decompiler to easily retain structure when composing theorems: each Hoare triple is reformulated into a list of updates and side-conditions that imply the Hoare triple. For example, the theorem describing the load instruction from above becomes the following.

Here $p12@r_1$ is a new automatically generated name which encodes that r_1 gets this value at location $p + 12$.

$$\neg z \wedge r_1 \in \mathsf{domain}\ m \wedge \mathsf{aligned}\ r_1 \ \Rightarrow$$
$$(p12@r_1 = m(r_1)) \ \Rightarrow$$
$$\{\, \mathsf{r1}\ r_1 * \mathsf{m}\ m * \mathsf{pc}\ (p{+}12) * \mathsf{sz}\ z \,\}$$
$$p{+}12 : \texttt{15911000 [ldrne r1, [r1]]}$$
$$\{\, \mathsf{r1}\ p12@r_1 * \mathsf{m}\ m * \mathsf{pc}\ (p{+}16) * \mathsf{sz}\ z \,\}$$

4.3.3 Control-flow discovery

As a third step, a summary of the control flow is collected: a simple program extracts the values from the pc-assertions in the Hoare triple theorems. The Hoare triple theorems from above produce the following control-flow summary:

$$0 \to 4,\ 4 \to 8,\ 8 \to 12,\ 12 \to 16,\ 16 \to 20,\ 16 \to 4$$

A heuristic then searches for loops in this graph; for the graph depicted above, it finds that instructions $4, 8, 12, 16$ constitute a loop.

4.3.4 One-pass theorem

Once loops have been detected in the control-flow graph, decompilation starts by proving a theorem for the inner-most (properly-nested) loop. Hoare triple theorems for individual instructions are composed and cases are merged to produce a single theorem describing one pass through the inner-most loop.

For the running example, this one-pass theorem is the following. Here the values of status bits have been hidden, using $\mathsf{s} = \exists n\ z\ c\ v.\ \mathsf{sn}\ n * \mathsf{sz}\ z * \mathsf{sc}\ c * \mathsf{sv}\ v$, and 'new' variables are introduced to keep track of final values.

$$\begin{array}{l}
(\mathsf{if}\ r_1 = 0\ \mathsf{then} \\
\quad (new@p = p{+}20) \wedge (new@r_0 = r_0) \wedge (new@r_1 = r_1) \wedge (new@m = m) \\
\ \mathsf{else} \\
\quad (p8@r_0 = r_0{+}1) \wedge \\
\quad (r_1 \in \mathsf{domain}\ m \wedge \mathsf{aligned}\ r_1) \wedge \\
\quad (p12@r_1 = m(r_1)) \wedge \\
\quad (new@p = p{+}4) \wedge (new@r_0 = p8@r_0) \wedge (new@r_1 = p12@r_1) \wedge (new@m = m)) \ \Rightarrow \\
\{\, \mathsf{r0}\ r_0 * \mathsf{r1}\ r_1 * \mathsf{m}\ m * \mathsf{pc}\ (p{+}4) * \mathsf{s} \,\} \\
p{+}4 : \texttt{E3510000}, \texttt{12800001}, \texttt{15911000}, \texttt{1AFFFFFB} \\
\{\, \mathsf{r0}\ new@r_0 * \mathsf{r1}\ new@r_1 * \mathsf{m}\ new@m * \mathsf{pc}\ new@p * \mathsf{s} \,\}
\end{array}$$

Post-processing. A minor reformulation is applied to the one-pass theorems: intermediate-value variables, such as $p8@r_0$ and $p12@r_1$, are turned into let-expressions. Theorems of the following form are applied for this transformation.

$$\begin{array}{lrcl}
\forall p\ q. & (\forall x.\ p\ x \Rightarrow q) & = & (\exists x.\ p\ x) \Rightarrow q \\
\forall p\ q. & (\exists x.\ q \wedge p\ x) & = & q \wedge (\exists x.\ p\ x) \\
\forall g\ p\ r. & (\exists x.\ \mathsf{if}\ g\ \mathsf{then}\ p\ x\ \mathsf{else}\ r\ x) & = & \mathsf{if}\ g\ \mathsf{then}\ (\exists x.\ p\ x)\ \mathsf{else}\ (\exists x.\ r\ x) \\
\forall p\ y. & (\exists x.\ (x = y) \wedge p\ x) & = & (\mathsf{let}\ x = y\ \mathsf{in}\ p\ x)
\end{array}$$

Logical variables are also renamed to remove prefixes '*p8@*' and '*p12@*'. The left-hand side of the one-pass theorem is the following after post-processing.

$$\begin{array}{l}
\mathsf{if}\ r_1 = 0\ \mathsf{then} \\
\quad (new@p = p{+}20) \wedge (new@r_0 = r_0) \wedge (new@r_1 = r_1) \wedge (new@m = m) \\
\mathsf{else} \\
\quad \mathsf{let}\ r_0 = r_0{+}1\ \mathsf{in} \\
\quad \mathsf{let}\ cond = \mathsf{aligned}\ r_1 \wedge r_1 \in \mathsf{domain}\ m\ \mathsf{in} \\
\quad \mathsf{let}\ r_1 = m(r_1)\ \mathsf{in} \\
\qquad ((new@p = p{+}4) \wedge (new@r_0 = r_0) \wedge (new@r_1 = r_1) \wedge (new@m = m) \wedge cond)
\end{array}$$

4.3.5 Proving the certificate theorem

Next decompilation will instantiate a loop rule, which introduces a tail-recursive function. All functions returned by the decompiler are instantiations of tailrec:

$$\mathsf{tailrec}\ x \quad = \quad \mathsf{if}\ G\ x\ \mathsf{then}\ \mathsf{tailrec}\ (F\ x)\ \mathsf{else}\ (D\ x) \tag{4.4}$$

(This function can be defined using while, from Section 2.5.4, as $\mathsf{tailrec}\ x = D\ (\mathsf{while}\ G\ F\ x)$ and therefore does not require a termination proof.)

For the running example, function g which describes the loop, Section 4.2.1, is defined as tailrec with parameters G, F and D instantiated as follows, based on the assumption in the one-pass theorem.

$$\begin{array}{rcl}
G & = & \lambda(r_0, r_1, m).\ \mathsf{if}\ r_1 = 0\ \mathsf{then}\ \mathsf{F}\ \mathsf{else} \\
& & \quad \mathsf{let}\ r_0 = r_0{+}1\ \mathsf{in} \\
& & \quad \mathsf{let}\ r_1 = m(r_1)\ \mathsf{in}\ \mathsf{T}
\end{array}$$

$$\begin{array}{rcl}
F = D & = & \lambda(r_0, r_1, m).\ \mathsf{if}\ r_1 = 0\ \mathsf{then}\ (r_0, r_1, m)\ \mathsf{else} \\
& & \quad \mathsf{let}\ r_0 = r_0{+}1\ \mathsf{in} \\
& & \quad \mathsf{let}\ r_1 = m(r_1)\ \mathsf{in}\ (r_0, r_1, m)
\end{array}$$

The user will not see the lengthy definition of g but instead receive the following equation, which is automatically proved as a corollary of theorem (4.4).

$$\begin{array}{rcl}
\mathsf{g}(r_0, r_1, m) & = & \mathsf{if}\ r_1 = 0\ \mathsf{then}\ (r_0, r_1, m)\ \mathsf{else} \\
& & \quad \mathsf{let}\ r_0 = r_0{+}1\ \mathsf{in} \\
& & \quad \mathsf{let}\ r_1 = m(r_1)\ \mathsf{in} \\
& & \quad\ \mathsf{g}(r_0, r_1, m)
\end{array}$$

Side-conditions and the fact that tailrec terminates are recorded by tailrec_pre, a machine-code version of the termination assumption terminates from Section 2.6.1. Here Q is a side-condition predicate that must be true on each iteration of the tailrec loop.

$$\begin{aligned}\mathsf{tailrec\_pre}\ x \quad = \quad & (\exists n.\ \neg G\ (\mathsf{funpow}\ F\ n\ x)) \wedge \\ & (\forall k.\ (\forall m.\ m < k \Rightarrow G\ (\mathsf{funpow}\ F\ m\ x)) \Rightarrow Q\ (\mathsf{funpow}\ F\ k\ x))\end{aligned}$$

The following theorem shows that tailrec_pre can be unrolled like a rewrite rule:

$$\mathsf{tailrec\_pre}\ x \quad = \quad Q\ x \wedge (G\ x \Rightarrow \mathsf{tailrec\_pre}\ (F\ x)) \tag{4.5}$$

The precondition $\mathsf{g_{pre}}$ is defined to be tailrec_pre with Q recording the necessary side condition for the load instruction:

$$\begin{aligned}Q \quad = \quad \lambda(r_0, r_1, m).\ & \mathsf{if}\ r_1 = 0\ \mathsf{then}\ \mathsf{T}\ \mathsf{else} \\ & \mathsf{let}\ r_0 = r_0{+}1\ \mathsf{in} \\ & \mathsf{let}\ \mathit{cond} = \mathsf{aligned}\ r_1 \wedge r_1 \in \mathsf{domain}\ m\ \mathsf{in} \\ & \mathsf{let}\ r_1 = m(r_1)\ \mathsf{in} \\ & \quad \mathit{cond}\end{aligned}$$

Again, the user will not see a lengthy definition but instead receives an automatically proved equation. The user gets the following equation which is proved as a corollary of theorem (4.5).

$$\begin{aligned}\mathsf{g_{pre}}(r_0, r_1, m) \quad = \quad & \mathsf{if}\ r_1 = 0\ \mathsf{then}\ \mathsf{T}\ \mathsf{else} \\ & \mathsf{let}\ r_0 = r_0{+}1\ \mathsf{in} \\ & \mathsf{let}\ \mathit{cond} = \mathsf{aligned}\ r_1 \wedge r_1 \in \mathsf{domain}\ m\ \mathsf{in} \\ & \mathsf{let}\ r_1 = m(r_1)\ \mathsf{in} \\ & \quad \mathsf{g_{pre}}(r_0, r_1, m) \wedge \mathit{cond}\end{aligned}$$

The precondition is stated in a way which makes it structurally very similar to g in order to enable the user to unroll g and $\mathsf{g_{pre}}$ simultaneously in verification proofs.

Next, decompilation instantiates a loop rule which will relate the code with the newly defined functions g and $\mathsf{g_{pre}}$. The loop rule is a consequence of the following induction scheme, which is proved from the definition of tailrec_pre.

$$\begin{aligned}\forall\varphi. \quad & (\forall x.\ Q\ x \wedge G\ x \wedge \varphi\ (F\ x) \Rightarrow \varphi\ x) \wedge \\ & (\forall x.\ Q\ x \wedge \neg G\ x \Rightarrow \varphi\ x) \\ & \Rightarrow (\forall x.\ \mathsf{tailrec\_pre}\ x \Rightarrow \varphi\ x)\end{aligned}$$

The decompiler's loop rule allows introduction of a tail-recursive function in case F describes the update of resources r when G is true, and D describes the update to r' when G is false.

$$\begin{aligned}\forall r\ r'\ c. \quad & (\forall x.\ Q\ x \wedge G\ x \Rightarrow \{r(x)\}\ c\ \{r(F\ x)\}) \wedge \\ & (\forall x.\ Q\ x \wedge \neg G\ x \Rightarrow \{r(x)\}\ c\ \{r'(D\ x)\}) \\ & \Rightarrow (\forall x.\ \mathsf{tailrec\_pre}\ x \Rightarrow \{r(x)\}\ c\ \{r'(\mathsf{tailrec}\ x)\})\end{aligned}$$

(This loop rule is proved by instantiating φ with $\lambda x.\ \{r(x)\}\, c\, \{r'(\mathsf{tailrec}\ x)\}$ and then applying the composition rule and the facts $c \cup c = c$ and $G\ x \Rightarrow (\mathsf{tailrec}\ (F\ x) = \mathsf{tailrec}\ x)$.)

Resource assertions r and r' are instantiated as follows to make the premises compatible with the one-pass theorem from the previous section.

$$\begin{aligned} r &= \lambda(r_0, r_1, m).\ (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ (r_0, r_1, m) * \mathsf{pc}\ (p{+}4) * \mathsf{s} \\ r' &= \lambda(r_0, r_1, m).\ (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ (r_0, r_1, m) * \mathsf{pc}\ (p{+}20) * \mathsf{s} \end{aligned}$$

With these instantiations the conclusion of the loop rule produces the desired result:

$$\begin{array}{l} \{\ (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ (r_0, r_1, m) * \mathsf{pc}\ (p+4) * \mathsf{s} * \langle \mathsf{g_{pre}}(r_0, r_1, m)\rangle\ \} \\ p{+}4 : \mathtt{E3510000}, \mathtt{12800001}, \mathtt{15911000}, \mathtt{1AFFFFFB} \\ \{\ (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ \mathsf{g}(r_0, r_1, m) * \mathsf{pc}\ (p+20) * \mathsf{s}\ \} \end{array}$$

The premises of the loop rule are trivial consequences of the one-pass theorem from the previous section.

4.3.6 Recursive decompilation

Once the innermost loops have been processed, decompilation continues by then considering the innermost loop enclosing the previously proved loops. This process is repeated until all loops and other surrounding code is described by one function and one Hoare-triple theorem. In other words, decompilation progresses in a bottom-up manner one loop at a time. (Decompilation could instead equally well be performed in a bottom-up manner one basic block at a time.)

For the linked-list example, the first pass through the decompiler produces a function g describing the execution of the (only) loop. During the second run, the Hoare-triple certificate for g is used as if it were a single instruction. The one-pass theorem for the second run of decompilation is the following after post-processing.

$$\begin{array}{l} (\mathsf{let}\ r_0 = 0\ \mathsf{in} \\ \mathsf{let}\ \mathit{cond} = \mathsf{g_{pre}}(r_0, r_1, m)\ \mathsf{in} \\ \mathsf{let}\ (r_0, r_1, m) = \mathsf{g}(r_0, r_1, m)\ \mathsf{in} \\ \quad ((\mathit{new}@p = p{+}20) \wedge (\mathit{new}@r_0 = r_0) \wedge (\mathit{new}@r_1 = r_1) \wedge (\mathit{new}@m = m) \wedge \mathit{cond}))\ \Rightarrow \\ \{\ \mathsf{r0}\ r_0 * \mathsf{r1}\ r_1 * \mathsf{m}\ m * \mathsf{pc}\ p * \mathsf{s}\ \} \\ p : \mathtt{E3A00000}, \mathtt{E3510000}, \mathtt{12800001}, \mathtt{15911000}, \mathtt{1AFFFFFB} \\ \{\ \mathsf{r0}\ \mathit{new}@r_0 * \mathsf{r1}\ \mathit{new}@r_1 * \mathsf{m}\ \mathit{new}@m * \mathsf{pc}\ \mathit{new}@p * \mathsf{s}\ \} \end{array}$$

Function f and a precondition $\mathsf{f_{pre}}$ are generated just as g and $\mathsf{g_{pre}}$ were generated above.

$$\begin{aligned} \mathsf{f}(r_0, r_1, m) &= \mathsf{let}\ r_0 = 0\ \mathsf{in}\ \mathsf{g}(r_0, r_1, m) \\ \mathsf{f_{pre}}(r_0, r_1, m) &= \mathsf{let}\ r_0 = 0\ \mathsf{in}\ \mathsf{g_{pre}}(r_0, r_1, m) \end{aligned}$$

The loop rule is applied, with $G = \lambda x.\ \mathsf{F}$, to produce the certificate Hoare-triple theorem.

$$\{\ (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ (r_0, r_1, m) * \mathsf{pc}\ p * \mathsf{s} * \langle \mathsf{f}_{\mathsf{pre}}(r_0, r_1, m)\rangle\ \}$$
$$p : \texttt{E3A00000}, \texttt{E3510000}, \texttt{12800001}, \texttt{15911000}, \texttt{1AFFFFFB}$$
$$\{\ (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ (\mathsf{f}\ (r_0, r_1, m)) * \mathsf{pc}\ (p + 20) * \mathsf{s}\ \}$$

4.3.7 Non-nested loops

The examples above have considered machine-code programs that start executing at the top of the code and exit at the end of the code, with all intermediate loops properly nested. More general forms of control flow are handled by treating the program counter as 'any other resource', i.e. including it into one of the value the generated function keeps track of, e.g. "(..., pc) is f(...)". In such cases, the position q of the code needs to be passed into the generated function f. As an example, when the following non-nested loops are processed

```
 0:  010080E2     L:  add  r0,r0,#1
 4:  010010E3     M:  tst  r0,#1
 8:  FCFFFF1A         bne  L          ;; might jump to L
12:  020050E2         subs r0,r0,#2
16:  FBFFFF1A         bne  M          ;; might jump to M
```

the generated function compares the program counter p with the position of the code q:

$$\begin{array}{l}\mathsf{f}(r_0, p, q)\ = \\ \quad \mathsf{if}\ p = q\ \mathsf{then}\ (\mathsf{let}\ r_0 = r_0 + 1\ \mathsf{in}\ \mathsf{f}(r_0, q{+}4, q)) \\ \quad \mathsf{else}\ \mathsf{if}\ r_0\ \&\ 1 \neq 0\ \mathsf{then}\ \mathsf{f}(r_0, q, q) \\ \quad \mathsf{else}\ \mathsf{let}\ r_0 = r_0 - 2\ \mathsf{in} \\ \qquad \mathsf{if}\ r_0 = 0\ \mathsf{then}\ (r_0, q{+}20)\ \mathsf{else}\ \mathsf{f}(r_0, q{+}4, q)\end{array}$$

The resulting theorem is:

$$p \in \{q, q{+}4\} \Rightarrow \{\ (\mathsf{r0}, \mathsf{pc})\ \mathsf{is}\ (r_0, p)\ \}\ \ q : ...\ \ \{\ (\mathsf{r0}, \mathsf{pc})\ \mathsf{is}\ \mathsf{f}(r_0, p, q)\ \}$$

The decompiler instantiates q with p before returning this theorem.

$$\{\ (\mathsf{r0}, \mathsf{pc})\ \mathsf{is}\ (r_0, p)\ \}\ \ p : ...\ \ \{\ (\mathsf{r0}, \mathsf{pc})\ \mathsf{is}\ \mathsf{f}(r_0, p, p)\ \}$$

4.3.8 Procedure calls

Procedure calls are, in machine code, implemented using branch-and-link instructions. These branch instructions perform a normal branch and at the same time save a return address, e.g. on ARM the branch-and-link instruction stores the return address in register 14. The following specification describes an ARM branch-and-link which makes a 48-byte long jump.

$$\{\mathsf{r14}\ x * \mathsf{pc}\ p\} \quad p : \texttt{090000EB} \quad \{\mathsf{r14}\ (p{+}4) * \mathsf{pc}\ (p{+}48)\}$$

One can prove a specification for the effect of a procedure call given a specification for the procedure which is to be called. Given the following specification for the procedure's code, i.e. code that is described by some function t,

$$\begin{array}{l}\{\ (\mathsf{pc}, \mathsf{r14}, \mathsf{res})\ \mathsf{is}\ (p, r_{14}, x) * \mathsf{t}_{\mathsf{pre}}(p, r_{14}, x)\ \} \\ \ p : \mathit{procedure\_code} \\ \{\ (\mathsf{pc}, \mathsf{r14}, \mathsf{res})\ \mathsf{is}\ \mathsf{t}(p, r_{14}, x)\ \}\end{array}$$

one can compose the the branch-and-link instruction with the procedure's specification,

$$\begin{array}{l}\{\ (\mathsf{pc}, \mathsf{r14}, \mathsf{res})\ \mathsf{is}\ (p, r_{14}, x) * \mathsf{t}_{\mathsf{pre}}(p{+}48, p{+}4, x)\ \} \\ \ p : \mathtt{090000EB} \cup p{+}48 : \mathit{procedure\_code} \\ \{\ (\mathsf{pc}, \mathsf{r14}, \mathsf{res})\ \mathsf{is}\ \mathsf{t}(p{+}48, p{+}4, x)\ \}\end{array}$$

and then strengthen the precondition to assume that control returns to the address passed as the return address, in this case $p{+}4$, let $\mathsf{fst}(x, y, z, ...) = x$,

$$\mathsf{fst}\ \mathsf{t}(p{+}48, p{+}4, x) = p{+}4$$

This composition followed by precondition strengthening proves a specification which has the shape of a specification for a normal instruction: control enters at pc p and exits at pc $(p{+}4)$. Let $\mathsf{snd}\ (x, y, z, ...) = (y, z, ...)$.

$$\begin{array}{l}\{\ \mathsf{pc}\ p * (\mathsf{r14}, \mathsf{res})\ \mathsf{is}\ (r_{14}, x) * (\mathsf{t}_{\mathsf{pre}}(p{+}48, p{+}4, x) \wedge \mathsf{fst}\ \mathsf{t}(p{+}48, p{+}4, x) = p{+}4)\ \} \\ \ p : \mathtt{090000EB} \cup p{+}48 : \mathit{procedure\_code} \\ \{\ \mathsf{pc}\ (p{+}4) * (\mathsf{r14}, \mathsf{res})\ \mathsf{is}\ (\mathsf{snd}\ \mathsf{t}(p{+}48, p{+}4, x))\ \}\end{array}$$

Thus specifications for procedure calls can be derived from the specification of the called procedure. The function generated by the decompiler includes a reference to the function t generated for the procedures body.

$$\mathsf{let}\ (r_{14}, x) = \mathsf{snd}\ \mathsf{t}(p{+}48, p{+}4, x)\ \mathsf{in}\ ...$$

Procedural recursion can be handled using the technique described in the section above.

4.3.9 Support for user-defined resource assertions

Notice that the operations of the decompiler do not depend on the particular properties of the basic resource assertions (r0, r1, m etc.). As a result, specifications involving completely different, user-defined, assertions can be fed into the decompiler for use instead of automatically proved instruction specifications.

As an example, consider this Hoare triple describing the alloc routine of one of the garbage collectors that were verified using decompilation (see Section 4.2.4). Here heap is a predicate stating that a garbage collected heap is present in memory. The allocation function's precondition states that the number of reachable elements in the abstract heap h from roots $v_1, v_2, v_3, v_4, v_5, v_6$ must be less than the heap limit l. The postcondition states that the abstract heap modelling function h is updated with a new element fresh h, which points at

a cons cell containing (v_1, v_2). The address of the new element is stored in the place of root variable v_1. Details of this specification are presented in the next Chapter.

$$\begin{array}{l}\mathsf{cardinality}\ (\mathit{reachables}\ [v_1, v_2, v_3, v_4, v_5, v_6]\ (\mathit{set}\ h)) < l\ \Rightarrow \\ \{\ \mathsf{heap}\ (a, l)\ (v_1, v_2, v_3, v_4, v_5, v_6, h) * \mathsf{s} * \mathsf{pc}\ p\ \} \\ p : \texttt{E50A3018}, \texttt{E50A4014}, ..., \texttt{AAA8F004} \\ \{\ \mathsf{heap}\ (a, l)\ (\mathsf{fresh}\ h, v_2, v_3, v_4, v_5, v_6, h[\mathsf{fresh}\ h \mapsto (v_1, v_2, 0)]) * \mathsf{s} * \mathsf{pc}\ (p + 348)\ \}\end{array}$$

When such specifications are given as input to the decompiler (using a special keyword '`insert` ...'), the decompiler can look up and use this specification. The resulting generated function contains the roots $v_1, v_2, v_3, v_4, v_5, v_6$ and heap h as variables:

$$\mathsf{let}\ (v_1, h) = (\mathsf{fresh}\ h,\ h[\mathsf{fresh}\ h \mapsto (v_1, v_2)])\ \mathsf{in}\ ...$$

the theorem contains the heap-assertion, and the generated precondition will keep track of a sufficient condition under which the heap limit is not exceeded.

4.3.10 Memory separation

The examples presented so far have only used a single memory assertion $\mathsf{m}\ m$ at a time. However, it is often useful to separate memory into logical segments, e.g. one for the stack s and one for the heap h:

$$\{\mathsf{m}\ s * \mathsf{m}\ h * ...\} \quad p : ... \quad \{...\}$$

Notice that this specification implicitly assumes that s and h describe disjoint parts of memory, since '$*$' makes assertions 'consume' memory (Section 3.4). The functions produced by the decompiler will then use two memory modelling functions s and h, and most importantly an update to the stack s will not affect the heap h, and vice versa. This feature is used heavily in some of the collector proofs in order to avoid some proof obligations that arise from possible pointer aliasing between the stack and the heap, i.e. one can avoid repeatedly proving that the stack and the heap do not overlap.

Memory separation can easily be implemented by modifying the output from the routines that derive Hoare triple specifications (Section 4.3.2). A heuristic is fed in to that routine which renames memory modelling functions depending on the registers that access them, e.g. our default heuristic renames memory modelling functions to s, if the stack pointer is used, while all other accesses are to memory called m.

4.4 Implementation, scalability and restrictions

The previous section described the decompilation algorithm and provided examples to guide intuition. This section will comment on the implementation, its scalability, and restrictions of the decompilation algorithm.

The decompiler is fully automatic and passes no dangling proof obligations to the user. It is implemented on top of HOL4 as a 2,200-line ML program. Of these 2,200 lines, approximately 500 lines are architecture specific. A large part of the decompiler was made architecture independent by operating only using proof rules that can be applied to Hoare-triple theorems of any architecture. Sections 3.7 and 3.5 give examples of such proof rules.

Early implementations of decompilation suffered from a lack of robustness. This lack of robustness was nearly always a result of some powerful proof tool, e.g. HOL4's standard simplifier `SIMP_RULE`, either not doing enough or, more often, simplifying expressions too aggressively. The decompiler has since then been completely rewritten in terms of primitive proof tools, such as the basic rewriter `REWRITE_RULE` and modus ponens `MATCH_MP`. The most critical parts, e.g. the post-processing of one-step theorems of Section 4.3.4, were written largely using handcrafted term conversions. Conversions are ML functions which given a term t produce a theorem $\vdash t = t'$, e.g. HOL4's `EVAL` function is a conversion which given term $2 + 3$ produces theorem $\vdash 2 + 3 = 5$.

With the latest decompiler implementation based on only primitive HOL4 proof tools, users can expect nearly any user-mode machine code to decompile successfully. An exception to this rule is code which uses complex control-flow structures that confuse the heuristic for control-flow detection. The heuristic for control-flow discovery works well for code where all branches are made to offsets of the current program counter. It considers branch-and-link instructions to be procedure calls and any instruction moving an address into the program counter from a register or stack location is assumed to perform a procedure return. As a result, this simple heuristic is easily confused by computed branches and calls to code pointers.

It should also be mentioned that the decompilation algorithm, as presented here, is only applicable to programs that have deterministic behaviour, otherwise the code is not a *function* of its inputs and the decompiler could not produce a function describing the code.

4.5 Discussion of related work

Different techniques for program verification with respect to accurate models of machine code are discussed below.

Symbolic simulation is a technique applicable to machine code modelled by an operational semantics. The approach is based on executing the next-state function on states where registers and memory location have been assigned symbolic values (logical variables, e.g. x, y). The result is a new state where resources hold expressions (e.g. $x+y$), for which the verifier is to prove properties. This method is emphasised and successfully applied by the ACL2 community [15, 78], e.g. Boyer and Yu used symbolic simulation in pioneering work on verification of the GNU string library compiled by GCC for the Motorola MC68020 [16]. Similar techniques were used by Liu and Moore in proofs of Java bytecode programs with respect to an extensive model of the Java Virtual Machine (JVM) [67]. Symbolic simulation has the disadvantage that the expressions produced by simulation, directly on top of the operation semantics, can become fiendishly complex and the technique does not directly support loops.

Using a programming logic directly on top of the definition of the semantics of the machine code is an approach which lends itself well to reasoning about loops and control flow. Shao's group at Yale [20] have used programming logics (inspired by separation logic and rely-guarantee) to verify (slightly idealised) assembly programs. Foundational proof-carrying code (FPCC) [102] and Typed Assembly Language (TAL) [111, 23, 21] also belong to this category. However, FPCC and TAL aim to check relatively weak safety properties – while the techniques presented here are concerned with proving complete functional correctness.

Using *verification condition generators (VCGs)* one annotates the code with assertions for which the VCGs calculates verification conditions that imply consistency of the assertions with respect to some programming logic. The integrity of the VCG is a concern, as practical VCGs tend to be complex [38, 60]. However, Homeier and Martin showed that VCGs can be verified [52] and Matthews et al. has showed that off-the-shelf theorem provers can be used in a way which gives the benefits of a VCG without actually constructing a full VCG [70]. Hardin et al. have applied the technique described by Matthews et al. to machine code of Rockwell Collins AAMP7G [50]. The main disadvantage of annotating the code with assertions is that the assertions become tied to the specific machine language and/or the particular definition of the semantics and, thus, do not provide the appropriate abstractions required for proof reuse.

Decompilation automatically reverse-engineers an abstraction of machine code. Decompilation is most often used to reverse compilation from a language such as C [104], but can, as was shown here, be used to produce abstractions in higher-order logic – a language much more amenable to formal reasoning than C. There is generally little work in this area, but work by Filliâtre [37] and Katsumata and Ohori [54] is related to ours. Filliâtre shows how imperative loops can, in type theory, be turned into recursive functions for purposes of verification. Unlike our approach his requires the code to be annotated with invariants and does not apply the method to low-level languages. Katsumata and Ohori have developed a decompiler, from a small subset of idealised Java bytecode to recursive functions, based on ideas from type theory. The decompiler implementing Katsumata and Ohori methodology has not been verified. It seems that their decompiler would need to be verified or made proof-producing, if its output were to be used in verification.

Chapter 5

Verified memory allocator and garbage collector

This chapter shows how an allocator, with a built-in Cheney collector, can be verified using decompilation and how the verification proof can be reused to verify machine code for different architectures implementing the same algorithm. The proof improves on published work on verification of Cheney collectors by being 70% shorter, directly reusable on several architectures and handling the 'out-of-memory' case properly.

5.1 Introduction

Memory is seen through machine code as a flat array-like data structure.[1] However, most programs use data structures such as trees, linked-lists and hash tables, which are commonly constructed in a segment of memory called *the heap*. Programs request space on the heap by calling a *memory allocator*, a procedure which reserves an unused part of the heap and returns a pointer to that part. Memory allocators are conventionally components of highly optimised software libraries that other programs assume are functionally correct.

This chapter shows how the decompiler from Chapter 4 can be used to verify the correctness of a memory allocator. The allocator considered here has a built-in *garbage collector*, a routine which traverses the heap and removes data that is no longer used. The collector ensures that calls to the memory allocator will succeed as long as the size of the currently used data on the heap does not exceed the capacity of the heap.

The collection routine implemented and verified here was invented by Cheney [22]. A Cheney collector divides the heap into two disjoint halves of equal size, only one of which is used for data at any time. When the collector is called, it copies all reachable data objects (data still in use) over into the other half of the heap, leaving behind all unreachable data objects

[1] True for sequential single processor systems; multi-processor systems may differ [68].

(unused data). These implementations of the Cheney algorithm are known as stop-the-world copying collectors and have the advantage that they only traverse reachable data objects. For verification they have the added interesting property that addresses are modified by the collector; essentially, addresses are renamed according to some bijective renaming function.

This chapter shows how a memory allocator can be implemented in ARM machine code (82 instructions, 328 bytes) and verified with the aid of decompilation, and how the verification proof can be reused to prove the correctness of x86 code (107 instructions, 306 bytes) and PowerPC code (90 instructions, 360 bytes) implementing the same algorithm.

The contribution is a reusable verification proof which is shorter by 70% than published proofs and the first to be used to map down to implementations in three different machine languages. Unlike other proofs [13, 76], this proof handles the 'out-of-memory' case properly.

5.2 Layers of abstraction

The construction and proof of the verified allocator is a top-down development using the following levels of abstraction. Throughout, allocated blocks will only contain two addresses since two will be sufficient for the case study of Chapter 7.

Level 4. At the top level, the heap is modelled as a finite partial function h, mapping natural numbers (addresses) to objects (n, m, d) where n and m are addresses and d is some data. At this level allocation is an update to the function h: allocation makes h map an unused address $(\mathsf{fresh}\ h) \notin \mathsf{domain}\ h$ to a new element (r_1, r_2, d) where r_1 and r_2 are two root addresses. Collection is unobservable at this level of abstraction.

Level 3. At the next level down, the heap is modelled as a total function from natural numbers (addresses) to a data-type with constructors Data, Ref and Emp:

$\mathsf{Data}\ (m, n, d)$ – an allocated block: n and m are addresses and d is data

$\mathsf{Ref}\ n$ – a special block used in intermediate stages of a collection cycle

Emp – non-existent memory block or "don't care"

At this level, allocation, which includes a Cheney collection routine, is implemented by a functional program cheney_alloc (presented in the next section).

Level 2. At the penultimate level of abstraction, the implementation is the decompilation of the ARM machine code (level below). Here memory is a function from word-aligned 32-bit words (machine addresses) to 32-bit words (content of memory).

Level 1. At the lowest level, we have the real ARM, x86 and PowerPC machine code.

Proving a relationship between levels 3 and 4 verifies the Cheney algorithm and its use in allocation. The connection between 2 and 3 maps the verified algorithm down to implementation specific types. The otherwise labour intensive verification of the level 2 implementation against machine-code implementations is automated by the decompiler from Chapter 4.

5.3 High-level implementation

As a first step, we implement our Cheney allocator at a high-level of abstraction (level 3) using the type constructors Data, Ref and Emp (from above). We define the following functions for dealing with the new data-types:

$$\begin{aligned} \mathsf{isRef}\ x &= \exists i.\ (x = \mathsf{Ref}\ i) \\ \mathsf{getRef}\ (\mathsf{Ref}\ x) &= x \\ \mathsf{getData}\ (\mathsf{Data}\ y) &= y \end{aligned}$$

We use function update $\mapsto$, as defined by:

$$m[a \mapsto b] = \lambda c.\ (\mathsf{if}\ a = c\ \mathsf{then}\ b\ \mathsf{else}\ m\ c)$$

Informally, a Cheney collector makes progress by moving heap blocks (one at a time) over to the new half of the heap. Each copied data block (Data) is replaced in the "from heap" with a reference cell (Ref) pointing to the address (in the "to heap") to which the Data block has been moved. The null pointer is natural number 0.

We implement moving of one data block as $\mathsf{move}\ (x, j, m)$. Here x is the address to be moved, j is the index of the next unused address in the "to heap" and m is the memory:

$$\begin{array}{l} \mathsf{move}\ (x, j, m) = \\ \quad \mathsf{if}\ x = 0\ \mathsf{then}\ (x, j, m)\ \mathsf{else} \\ \quad\quad \mathsf{if}\ \mathsf{isRef}\ (m\ x)\ \mathsf{then} \\ \quad\quad\quad (\mathsf{getRef}\ (m\ x), j, m) \\ \quad\quad \mathsf{else} \\ \quad\quad\quad \mathsf{let}\ m = m[j \mapsto m\ x]\ \mathsf{in} \\ \quad\quad\quad \mathsf{let}\ m = m[x \mapsto \mathsf{Ref}\ j]\ \mathsf{in} \\ \quad\quad\quad\quad (j, j + 1, m) \end{array}$$

As part of initialisation, the Cheney collector moves all data elements addressed by root variables (a list rs of natural numbers) into the "to heap" using move:

$$\begin{array}{l} \mathsf{move\_roots}\ ([], j, m) = ([], j, m) \\ \mathsf{move\_roots}\ (r{::}rs, j, m) = \\ \quad \mathsf{let}\ (r, j, m) = \mathsf{move}\ (r, j, m)\ \mathsf{in} \\ \quad \mathsf{let}\ (rs, j, m) = \mathsf{move\_roots}\ (rs, j, m)\ \mathsf{in} \\ \quad\quad (r{::}rs, j, m) \end{array}$$

After moving data objects addressed by roots into the heap, a Cheney collector will successively move over all data blocks which are referenced from the data cells in the "to heap". Index i, the address up to which all data elements have been processed, is incremented until it meets index j, which is the address of the first empty space in the "to heap". Index j is

occasionally bumped forward by move.

$$
\begin{array}{l}
\mathsf{cheney\_loop}\ (i, j, e, m)\ = \\
\quad \mathsf{if}\ i = j\ \mathsf{then}\ (i, m)\ \mathsf{else} \\
\quad\quad \mathsf{let}\ (x, y, d) = \mathsf{getData}\ (m\ i)\ \mathsf{in} \\
\quad\quad \mathsf{let}\ (x, j, m) = \mathsf{move}\ (x, j, m)\ \mathsf{in} \\
\quad\quad \mathsf{let}\ (y, j, m) = \mathsf{move}\ (y, j, m)\ \mathsf{in} \\
\quad\quad \mathsf{let}\ m = m[i \mapsto \mathsf{Data}\ (x, y, d)]\ \mathsf{in} \\
\quad\quad\quad \mathsf{cheney\_loop}\ (i + 1, j, e, m)
\end{array}
$$

The full Cheney collector is implemented by cheney_collector, which is defined below. The collector starts by inverting u (which keeps track of which heap half is used); it then calculates the start of the new heap and stores the result in i; root elements are copied over into the heap using move_roots (which also initialises index j); the main loop cheney_loop copies over all other reachable data elements; and finally, the memory outside of the new heap is erased (overwritten with Emp elements) using cut:

$$
\mathsf{cut}\ (i, j)\ m\ =\ \lambda k.\ \ \mathsf{if}\ i \leq k \wedge k < j\ \mathsf{then}\ m\ j\ \mathsf{else}\ \mathsf{Emp}
$$

$$
\begin{array}{l}
\mathsf{cheney\_collector}\ (i, e, rs, l, u, m)\ = \\
\quad \mathsf{let}\ u = \neg\, u\ \mathsf{in} \\
\quad \mathsf{let}\ i = (\mathsf{if}\ u\ \mathsf{then}\ 1 + l\ \mathsf{else}\ 1)\ \mathsf{in} \\
\quad \mathsf{let}\ (rs, j, m) = \mathsf{move\_roots}\ (rs, i, m)\ \mathsf{in} \\
\quad \mathsf{let}\ (j, m) = \mathsf{cheney\_loop}\ (i, j, i + l, m)\ \mathsf{in} \\
\quad \mathsf{let}\ m = \mathsf{cut}\ (i, i + l)\ m\ \mathsf{in} \\
\quad\quad (j, i + l, rs, l, u, m)
\end{array}
$$

The Cheney collector takes as input and produces as output a state where the first elements i and e are addresses; i is the address of the next free slot for a data object and e is the address of the first block outside of the heap ($i = e$ means that there is no space left).

The allocator will only call the collector if the current heap is completely full:

$$
\begin{array}{l}
\mathsf{cheney\_alloc\_gc}\ (i, e, rs, l, u, m)\ = \\
\quad \mathsf{if}\ i < e\ \mathsf{then}\ (i, e, rs, l, u, m)\ \mathsf{else}\ \mathsf{cheney\_collector}\ (i, e, rs, l, u, m)
\end{array}
$$

Allocation performs a collection, if necessary, and then inserts a new element into memory, if there is enough space. A successful allocation writes the pointer to the new element into rs. An unsuccessful allocation writes 0 into rs.

$$
\begin{array}{l}
\mathsf{cheney\_alloc\_aux}\ (i, e, r_1{::}r_2{::}rs, l, u, m)\ d\ = \\
\quad \mathsf{if}\ i = e\ \mathsf{then}\ (i, e, 0{::}r_2{::}rs, l, u, m)\ \mathsf{else} \\
\quad\quad \mathsf{let}\ m = m[i \mapsto \mathsf{Data}\ (r_1, r_2, d)]\ \mathsf{in} \\
\quad\quad\quad (i + 1, e, i{::}r_2{::}rs, l, u, m)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{cheney\_alloc}\ (i, e, rs, l, u, m)\ d\ = \\
\quad \mathsf{cheney\_alloc\_aux}\ (\mathsf{cheney\_alloc\_gc}\ (i, e, rs, l, u, m))\ d
\end{array}
$$

5.4 High-level specification and proof

The specification and verification of the Cheney algorithm is described next.

5.4.1 Well-formed states

The state tuple (i, e, r, l, u, m) at abstraction level 3 must satisfy ok_state before and after a call to cheney_alloc. The ok_state predicate requires index i, the address of the next empty slot, to lie in between the start and the end address of the current heap, a and e respectively. All nonzero roots (r is a list of root addresses) must point to a data element (address in set s). Locations that do not contain data elements are empty (Emp). Locations that contain data elements must only contain pointers to other data elements or zero (members of set $\{0\} \cup s$).

$$\mathsf{range}\ (i, j) = \{\ k \mid i \leq k \wedge k < j\ \}$$

$$\begin{array}{l}
\mathsf{ok_state}\ (i, e, r, l, u, m) = \\
\quad \mathsf{let}\ a = (\mathsf{if}\ u\ \mathsf{then}\ 1 + l\ \mathsf{else}\ 1)\ \mathsf{in} \\
\quad \mathsf{let}\ s = \mathsf{range}\ (a, i)\ \mathsf{in} \\
\qquad a \leq i \wedge i \leq e \wedge (e = a + l) \wedge \\
\qquad (\forall k.\ \mathsf{mem}\ k\ r \wedge k \neq 0 \Rightarrow k \in s) \wedge \\
\qquad (\forall k.\ k \notin s \Rightarrow (m\ k = \mathsf{Emp})) \wedge \\
\qquad (\forall k.\ k \in s \Rightarrow \exists x\ y\ d.\ (m\ k = \mathsf{Data}\ (x, y, d)) \wedge \{x, y\} \subseteq \{0\} \cup s)
\end{array}$$

At abstraction level 4, the state is a tuple (r, h) with r a list of root addresses and h a finite partial mapping from natural numbers to tuples (x, y, d). The heap h must define an element for each nonzero root and pointer in the image of h, and zero must not be in the domain of h.

$$\begin{array}{l}
\mathsf{ok_abs}\ (r, h) = \\
\quad 0 \notin \mathsf{domain}\ h\ \wedge\ (\forall k.\ \mathsf{mem}\ k\ r \wedge k \neq 0\ \Rightarrow\ k \in \mathsf{domain}\ h) \wedge \\
\quad (\forall x\ y\ z\ d.\ h\ x = (y, z, d)\ \Rightarrow\ \{y, z\} \subseteq \{0\} \cup \mathsf{domain}\ h)
\end{array}$$

5.4.2 Set-based representation

In the specification below we relate level 3 and level 4 states by converting them into a common set-based representation of the heap, e.g. the following three element set

$$\{\,(1, 2, 5, 3567), (2, 5, 1, 7000), (5, 0, 1, 23)\,\}$$

is a heap containing blocks at addresses 1, 2 and 5. The format of each block is

$$(\,\text{location}\,,\ \text{next pointer 1}\,,\ \text{next pointer 2}\,,\ \text{data}\,)$$

The level 3 memory is abstracted into the set-based representation using a function called abstract which applies an address-translation function b to each address:

$$\mathsf{abstract}\ (b, m)\quad =\quad \{\ (b\ x, b\ y, b\ z, d) \mid m\ x = \mathsf{Data}\ (y, z, d)\ \}$$

The heap from level 4 is translated into the set-representation using a function called *set*:

$$\mathsf{set}\ h \quad = \quad \{\ (x, y, z, d) \mid x \in \mathsf{domain}\ h \wedge h\ x = (y, z, d)\ \}$$

5.4.3 Specification of the Cheney collector

The specification of the Cheney collector makes use of the concept of a reachable node, i.e. a node to which there exists some path through next-pointers from some root r in rs:

$$[] +\!\!+ ys = ys$$
$$(x{::}xs) +\!\!+ ys = x{::}(xs +\!\!+ ys)$$

$$\mathsf{path}\ [x]\ s = \mathsf{T}$$
$$\mathsf{path}\ \ (x{::}y{::}ys)\ s = \mathsf{path}\ \ (y{::}ys)\ s \wedge \exists z\ d.\ (x, y, z, d) \in s \vee (x, z, y, d) \in s$$

$$\mathsf{reachable}\ r\ s\ i = (r = i) \vee \exists p.\ \mathsf{path}\ ([r] +\!\!+ p +\!\!+ [i])\ s$$
$$\mathsf{reachables}\ rs\ s = \{\ (a, x, y, d) \mid (a, x, y, d) \in s \wedge \exists r.\ \mathsf{mem}\ r\ rs \wedge \mathsf{reachable}\ r\ s\ a\ \}$$

The specification for cheney_collector states that the collector removes all unreachable elements. The collector renames all addresses according to some function b, which is its own inverse $b \circ b = (\lambda x.x) = \mathsf{id}$.

$$\begin{array}{l}(\mathsf{cheney\_collector}\ (i, e, r, l, u, m) = (i', e', r', l', u', m')) \wedge \\ \mathsf{ok\_state}\ (i, e, r, l, u, m)\ \Rightarrow \\ \mathsf{ok\_state}\ (i', e', r', l', u', m') \wedge (l = l') \wedge \\ \exists b.\ (b \circ b = \mathsf{id}) \wedge (b\ 0 = 0) \wedge (\mathsf{map}\ b\ r = r') \wedge \\ \quad (\mathsf{abstract}\ (b, m') = \mathsf{reachables}\ r\ (\mathit{abstract}(\mathsf{id}, m)))\end{array}$$

5.4.4 Specification of the memory allocator

The specification of allocation is stated in terms of the heap h at abstraction level 4. We define ch_inv, below, to relate h from level 4 with the memory m at level 3. Predicate ch_inv states that there exist some bijection b which relates memory m of level 3 with heap h at level 4, all elements reachable from root nodes r in h must exist in m, and similarly, all data blocks present in m must exist in h:

$$\begin{array}{l}\mathsf{ch\_inv}\ (r, h, l')\ (i, e, c, l, u, m) = \\ \quad \mathsf{ok\_state}\ (i, e, c, l, u, m) \wedge \mathsf{ok\_abs}\ (r, h) \wedge (l = l') \wedge \\ \quad \exists b.\ \mathsf{bijection}\ b \wedge (b(0) = 0) \wedge (\mathsf{map}\ b\ c = r) \wedge \\ \qquad \mathsf{reachables}\ r\ (\mathsf{set}\ h) \subseteq \mathsf{abstract}\ (b, m) \subseteq \mathsf{set}\ h\end{array}$$

The specification for allocation creates a fresh address fresh h. Here fresh is a function which selects some unused non-zero natural number which is not in the domain of h (the domain of h is finite by definition), i.e.

$$\forall h\ i.\ \ (\mathsf{fresh}\ h = i)\ \Rightarrow\ (0 \neq i) \wedge i \notin \mathsf{domain}\ h$$

The specification of allocation can now be expressed concisely. The following specification states: if cheney_alloc turns state s into s' and the cardinality of the set of reachable nodes

in heap h is less than the heap capacity then s' is related to h updated to map fresh h to a new element (r_1, r_2, d).

$$\begin{aligned}
&(\mathsf{cheney_alloc}\ s\ d = s') \wedge \\
&\mathsf{cardinality}\ (\mathsf{reachables}\ [r_1, r_2, r_3, r_4, \ldots, r_n]\ (\mathsf{set}\ h)) < l \wedge \\
&\mathsf{ch_inv}\ ([r_1, r_2, r_3, r_4, \ldots, r_n], h, l)\ s\ \Rightarrow \\
&\mathsf{ch_inv}\ ([\mathsf{fresh}\ h, r_2, r_3, r_4, \ldots, r_n], h[\mathsf{fresh}\ h \mapsto (r_1, r_2, d)], l)\ s'
\end{aligned}$$

We also have a specification for the case when cheney_alloc runs out of memory, the allocator returns the null pointer rather than a pointer to a new data block:

$$\begin{aligned}
&(\mathsf{cheney_alloc}\ s\ d = s') \wedge \\
&\mathsf{cardinality}\ (\mathsf{reachables}\ [r_1, r_2, r_3, r_4, \ldots, r_n]\ (\mathsf{set}\ h)) \geq l \wedge \\
&\mathsf{ch_inv}\ ([r_1, r_2, r_3, r_4, \ldots, r_n], h, l)\ s\ \Rightarrow \\
&\mathsf{ch_inv}\ ([0, r_2, r_3, r_4, \ldots, r_n], h, l)\ s'
\end{aligned}$$

ch_inv was engineered to make collection cycles unobservable at abstraction level 4.

5.4.5 Verification proof

Finding the invariant for cheney_loop is the main challenge in proving the specifications above. It took approximately one week to find an invariant which was sufficiently strong to imply the necessary specifications. The definition of the invariant is listed in Figure 5.1. An overview of its main points is presented here:

Line 1 constrains the boundary variables: $0 < b \leq i \leq j \leq e \leq f$. Here b points to the start of the "to heap", i points at the next data element to be processed, j points at the first empty slot in the "to heap", e points at the end of the "to heap", and f is some pointer beyond which memory consists of only empty "don't care" elements.

Lines 2–8 state restrictions on what type of object can be where in memory, and also where each type of object can point, e.g. line 6,

$$\texttt{d1 (cut (i,j) m)} \subseteq \{0\} \cup \texttt{dr0 (icut (b,e) m)}$$

states that every data element in the range i to j must point to either zero or some non-empty cell outside of the "to heap"', i.e. outside of the range b to e.

Lines 9, A and B state, respectively, that there are at most $e-j$ data elements outside of the range b to e, that all elements in the range b to i are reachable from root addresses, and that pointers in reference cells are preserved.

Lines C–E state that the pointers in the reference cells define a function v by which addresses have been renamed; v is its own inverse (and thus v is a bijection).

Invariant cheney_inv was used in proving the following theorem by induction over $e-i$, which measures progress of index i towards index e, the end of the current heap segment.

$$\begin{aligned}
&\mathsf{cheney_inv}\ (b, i, j, e, f, m, m_0, m_1, r) \wedge \\
&(\mathsf{cheney_loop}\ (i, j, e, m) = (i', m'))\ \Rightarrow \\
&\mathsf{cheney_inv}\ (b, i', i', e, f, m', m_0, m_1, r) \wedge j \leq i'
\end{aligned}$$

```
     range (i,j) = { k | i ≤ k ∧ k < j }
     irange (i,j) = { k | ¬(i ≤ k ∧ k < j) }

     cut (i,j) m = λk. (if range (i,j) k then m k else EMP)
     icut (i,j) m = λk. (if irange (i,j) k then m k else EMP)

     d0 m = { k | ∃x y z. m k = DATA (x,y,z) }
     d1 m = { x | ∃k y z. (m k = DATA (x,y,z)) ∨ (m k = DATA (y,x,z)) }
     r0 m = { k | ∃a. m k = REF a }
     r1 m = { a | ∃k. m k = REF a }
     dr0 m = d0 m ∪ r0 m
     dr1 m = d1 m ∪ r1 m

     addr k n EMP = (n = k)
     addr k n (REF i) = (n = i)
     addr k n (DATA x) = (n = k)

     abs m = { (a,n,n',d) |
               ∃k k'. (m a = DATA (k,k',d)) ∧ addr k n (m k) ∧ addr k' n' (m k') }

     basic_abs m = { (a,n,n',d) | m a = DATA (n,n',d) }

     apply h s = { (a,n,n',d) | (h a,h n,h n',d) ∈ s }

     cheney_inv (b,i,j,e,f,m,m0,m1,r) =
1      0 < b ∧ b ≤ i ∧ i ≤ j ∧ j ≤ e ∧ e ≤ f ∧
2      (∀k. k ∈ range (j,e) ∨ k = 0 ∨ f < k ⇒ (m k = EMP)) ∧
3      (d0 (cut (b,j) m) = range (b,j)) ∧ (range (b,j) = r1 m) ∧
4      d1 (cut (b,i) m) ⊆ {0} ∪ range (b,j) ∧
5      range (i,j) ⊆ r ∪ d1 (cut (b,i) m) ∧
6      d1 (cut (i,j) m) ⊆ {0} ∪ dr0 (icut (b,e) m) ∧
7      d1 (icut (b,e) m) ⊆ {0} ∪ dr0 (icut (b,e) m) ∧
8      (∀i j k. (m i = REF k) ∧ (m j = REF k) ⇒ (i = j)) ∧
9      CARDINALITY (d0 (icut (b,e) m)) ≤ e - j ∧ FINITE (d0 (icut (b,e) m)) ∧
A      range (b,i) ⊆ { x | ∃t. t ∈ r ∧ reachable t (basic_abs (cut (b,i) m)) x } ∧
B      (∀k i. (m0 k = REF i) ⇒ (m k = REF i)) ∧
C      ∃v. (v ∘ v = λx.x) ∧ (abs m1 = apply v (abs m)) ∧
D          (∀k. ¬isREF (m k) ∧ k ∉ range (b,j) ⇒ (v k = k)) ∧
E          (∀k i. (m k = REF i) ⇒ (v k = i))
```

Figure 5.1: The definition of cheney_inv in HOL4.

5.5 Low-level implementation

For the low-level assembly code implementations, we need to decide how to represent Data, Ref and Emp cells in real memory. Two different representation have been implemented:

α. one where each heap element consists of two addresses and one word of data; and

β. another representation where heap elements are two 32-bit words that are either pointers or data – a pointer is distinguished from data by inspection of the two least significant bits of the 32-bit word (word w is a pointer if w is word aligned, i.e. $w \,\&\, 3 = 0$).

A verified implementation of representation β is used for the LISP interpreter of Chapter 7.

For clarity of presentation, this chapter concerns an implementation and proof based on representation α, in which the implementation of Data, Ref and Emp are outlined below. (These informal descriptions are made precise in Section 5.7.)

'Data (m, n, d)' is represented in memory as three consecutive 32-bit words: two word-aligned pointers m and n (zero is the null pointer) followed by a word d storing data;

'Ref n' also consists of three words but only the first carries information: the first word contains pointer n with the least significant bit set to 1 (making it a misaligned pointer and thus distinguishable from Data cells where pointers are word-aligned);

'Emp' elements have no representation in the real memory.

Based on these design decisions, assembly implementations of cheney_alloc were written. As an example, move from Section 5.3 was implemented in one place for ARM as:

```
E3550000     cmp r5,#0           ;;  test whether r5 is nil
0A000009     beq L1              ;;  if yes, exit by jumping to L1
E5957000     ldr r7,[r5]         ;;  load value from address r5 into r7
E3170001     tst r7,#1           ;;  test whether r5 points at REF element
04847004     streq r7,[r4],#4    ;;  if no then:
05958004     ldreq r8,[r5,#4]    ;;    copy the three words of a DATA element
05957008     ldreq r7,[r5,#8]    ;;    at address r5 to location given by r4,
04848004     streq r8,[r4],#4    ;;    increment r4 by 12 (length of three
04847004     streq r7,[r4],#4    ;;    32-bit words), then store new
0244700B     subeq r7,r4,#11     ;;    REF element at address r5
05857000     streq r7,[r5]       ;;  end if
E2475001     sub r5,r7,#1        ;;  calculate address of new DATA element
        L1: ...                  ;;  (start of next routine)
```

In another place, move was implemented by the same code except that `r5` was replaced by `r6` in order to make use of registers more effectively. Note that such renamings are trivially equivalent after decompilation. The above ARM code uses conditional execution for some of the instructions (those ending in '`eq`').

5.6 Low-level specification and proof

The hand-written ARM code was decompiled into a functional implementation. The machine code above for move is decompiled into a function we call arm_move:

```
arm_move (r4,r5,r7,r8,f) =
  if r5 = 0w then
    (r4,r5,r7,r8,f)
  else
    let r7 = f r5 in
      if r7 && 1w = 0w then
        let f = f[r4 ↦ r7] in
        let r4 = r4 + 4w in
        let r8 = f (r5 + 4w) in
        let r7 = f (r5 + 8w) in
        let f = f[r4 ↦ r8] in
        let r4 = r4 + 4w in
        let f = f[r4 ↦ r7] in
        let r4 = r4 + 4w in
        let r7 = r4 - 11w in
        let f = f[r5 ↦ r7] in
        let r5 = r7 - 1w in
          (r4,r5,r7,r8,f)
      else
        let r5 = r7 - 1w in
          (r4,r5,r7,r8,f)
```

The entire 70-line ARM implementation for cheney_alloc is decompiled into a function called arm_cheney_alloc. The decompiler automates the proof which relates arm_cheney_alloc with the ARM code, i.e. it proves the following theorem:

$$\begin{array}{l}\mathsf{arm\_cheney\_alloc\_pre}\ (r_3,r_4,r_5,r_6,r_7,r_8,r_9,f)\ \Rightarrow\\ \{\ \mathsf{r3}\ r_3 * \mathsf{r4}\ r_4 * \mathsf{r5}\ r_5 * \mathsf{r6}\ r_6 * \mathsf{r7}\ r_7 * \mathsf{r8}\ r_8 * \mathsf{r9}\ r_9 * \mathsf{memory}\ f * \mathsf{s} * \mathsf{pc}\ p\ \}\\ \quad p : \texttt{E50A3018}, \texttt{E50A4014}, \ldots, \texttt{AAA8F004}\\ \{\ \mathsf{let}\ (r_3,r_4,r_5,r_6,r_7,r_8,r_9,f) = \mathsf{arm\_cheney\_alloc}\ (r_3,r_4,r_5,r_6,r_7,r_8,r_9,f)\ \mathsf{in}\\ \quad \mathsf{r3}\ r_3 * \mathsf{r4}\ r_4 * \mathsf{r5}\ r_5 * \mathsf{r6}\ r_6 * \mathsf{r7}\ r_7 * \mathsf{r8}\ r_8 * \mathsf{r9}\ r_9 * \mathsf{memory}\ f * \mathsf{s} * \mathsf{pc}\ (p{+}348)\ \}\end{array}$$

The decompiler automates lengthy proofs that deal with the complex definition of ARM's next-state function. It presents the user with a function which is large but much more manageable than the next-state function of the ARM model. The decompiler shields the user from details of the ISA.

5.7 Relating low-level and high-level specifications

Section 5.4 presented a formal connection between abstraction levels 3 and 4, and Section 5.6 presented a connection between abstraction levels 1 and 2. In order to get a usable high-

level specification for allocation, we formalise a connection between levels 2 and 3 (informally described in Section 5.5), and then construct a specification relating levels 1 and 4.

5.7.1 Coupling invariant

Section 5.5 outlined the design decisions made when writing machine code implementation of cheney_alloc. We formalise the connection between the two abstraction levels here.

Level 3, at which cheney_alloc is defined, treats addresses as natural numbers. Addresses are translated into 32-bit words by multiplying by 12 and then adding a base address a. Here and throughout n2w converts a natural number into a 32-bit machine word.

$$\mathsf{ref\_addr}\ a\ n\ =\ a + 12 \times (\mathsf{n2w}\ n) :\mathsf{word32}$$

The correspondence between memory at level 2 (function f, base address a) and level 3 (function m) is captured by the predicate ref_memory m (a, f).

$$\begin{aligned}
\mathsf{ref_mem}\ i\ \mathsf{Emp}\ (a,f) &= \mathsf{T} \\
\mathsf{ref_mem}\ i\ (\mathsf{Ref}\ j)\ (a,f) &= (f\ (\mathsf{ref_addr}\ a\ i) = \mathsf{ref_addr}\ a\ j + 1) \\
\mathsf{ref_mem}\ i\ (\mathsf{Data}\ (x,y,d))\ (a,f) &= (f\ (\mathsf{ref_addr}\ a\ i + 0) = \mathsf{ref_addr}\ a\ x)\ \wedge \\
&\quad\ (f\ (\mathsf{ref_addr}\ a\ i + 4) = \mathsf{ref_addr}\ a\ y)\ \wedge \\
&\quad\ (f\ (\mathsf{ref_addr}\ a\ i + 8) = d) \\
\mathsf{ref_memory}\ m\ (a,f) &= (a\ \&\ 3 = 0)\ \wedge\ \forall i.\ \mathsf{ref_mem}\ i\ (m\ i)\ (a,f)
\end{aligned}$$

The full coupling invariant ch_word has the lengthy definition given below. The definition uses w2n which converts an (unsigned) machine word into a natural number.[2]

$$\begin{aligned}
&\mathsf{ch_word}\ (i,e,rs,l,u,m)\ (r_1,r_2,r_3,r_4,r_5,r_6,a,f)\ = \\
&\ \exists x_1\ x_2\ x_3\ x_4\ x_5\ x_6. \\
&\quad (rs = [x1,x2,x3,x4,x5,x6])\ \wedge\ \mathsf{ok_state}\ (i,e,rs,l,u,m)\ \wedge\ \mathsf{ref_memory}\ m\ (a,f)\ \wedge \\
&\quad 32\ \leq\ \mathsf{w2n}\ a\ \wedge\ \mathsf{w2n}\ a + 2 \times 12 \times l + 12 < 2^{32}\ \wedge \\
&\quad (r_1 = \mathsf{ref_addr}\ a\ x_1)\ \wedge\ (r_2 = \mathsf{ref_addr}\ a\ x_2)\ \wedge\ (r_3 = \mathsf{ref_addr}\ a\ x_3)\ \wedge \\
&\quad (r_4 = \mathsf{ref_addr}\ a\ x_4)\ \wedge\ (r_5 = \mathsf{ref_addr}\ a\ x_5)\ \wedge\ (r_6 = \mathsf{ref_addr}\ a\ x_6)\ \wedge \\
&\quad (f\ a = \mathsf{ref_addr}\ a\ i)\ \wedge\ (f\ (a+4) = \mathsf{ref_addr}\ a\ e)\ \wedge \\
&\quad (f\ (a-28) = (\mathsf{if}\ u\ \mathsf{then}\ 0\ \mathsf{else}\ 1))\ \wedge\ (f\ (a-32) = 12 \times \mathsf{n2w}\ l)\ \wedge \\
&\quad (\mathsf{domain}\ f = \{\ a + 4 \times \mathsf{n2w}\ i \mid i \leq 4 * l + 2\ \} \cup \{\ a - 4 * \mathsf{n2w}\ i \mid i \leq 8\ \})
\end{aligned}$$

This coupling invariant is maintained by the ARM implementation of cheney_alloc and is also strong enough to imply the automatically generated side condition arm_cheney_alloc_pre.

$$\begin{aligned}
&\forall s\ t.\ \mathsf{ch_word}\ s\ t\ \Rightarrow\ \mathsf{ch_word}\ (\mathsf{cheney_alloc}\ s\ 0)\ (\mathsf{arm_cheney_alloc}\ t) \\
&\forall s\ t.\ \mathsf{ch_word}\ s\ t\ \Rightarrow\ \mathsf{arm_cheney_alloc_pre}\ t
\end{aligned}$$

[2] n2w and w2n are related: $\forall w.\ \mathsf{n2w}\ (\mathsf{w2n}\ w) = w$; $\forall n.\ \mathsf{w2n}\ (\mathsf{n2w}\ n) = n \bmod 2^{\alpha}$, for words of length α.

5.7.2 Overall specification for allocation

We compose ch_inv and ch_word to define a relation ch_rel between level 4 and level 2:

$$\mathsf{ch\_rel}\ s\ t\ =\ \exists u.\ \mathsf{ch\_inv}\ s\ u\ \wedge\ \mathsf{ch\_word}\ u\ t$$

The specification of allocation for the case when there is enough space on the heap ($<$), and the case when there is not enough space ($\geq$):

$$\begin{array}{l}\mathsf{cardinality}\ (\mathsf{reachables}\ [v_1,v_2,v_3,v_4,v_5,v_6]\ (\mathsf{set}\ h)) < l\ \Rightarrow\\ \mathsf{ch\_rel}\ ([v_1,v_2,v_3,v_4,v_5,v_6],h,l)\ s\ \Rightarrow\\ \mathsf{ch\_rel}\ ([\mathsf{fresh}\ h,v_2,v_3,v_4,v_5,v_6],h[\mathsf{fresh}\ h\mapsto(v_1,v_2,0)],l)\ (\mathsf{arm\_cheney\_alloc}\ s)\end{array}$$

$$\begin{array}{l}\mathsf{cardinality}\ (\mathsf{reachables}\ [v_1,v_2,v_3,v_4,v_5,v_6]\ (\mathsf{set}\ h)) \geq l\ \Rightarrow\\ \mathsf{ch\_rel}\ ([v_1,v_2,v_3,v_4,v_5,v_6],h,l)\ s\ \Rightarrow\\ \mathsf{ch\_rel}\ ([0,v_2,v_3,v_4,v_5,v_6],h,l)\ (\mathsf{arm\_cheney\_alloc}\ s)\end{array}$$

Using the decompiler-derived theorem (Section 5.6) about the ARM code, one can prove a machine code specification for allocation. Let $\mathsf{heap}\ (a,l)\ (v_1,v_2,v_3,v_4,v_5,v_6,h)$ state that abstract heap h with roots v_1 – v_6, capacity l is based around address a:

$$\begin{array}{l}\mathsf{heap}\ (a,l)\ (v_1,v_2,v_3,v_4,v_5,v_6,h)\ =\\ \quad\exists r_3\ r_4\ r_5\ r_6\ r_7\ r_8\ f.\\ \quad\ \mathsf{r3}\ r_3 * \mathsf{r4}\ r_4 * \mathsf{r5}\ r_5 * \mathsf{r6}\ r_6 * \mathsf{r7}\ r_7 * \mathsf{r8}\ r_8 * \mathsf{r9}\ r_9 * \mathsf{memory}\ f *\\ \quad\ \langle\mathsf{ch\_rel}\ ([v_1;v_2;v_3;v_4;v_5;v_6],h,l)\ (r_3,r_4,r_5,r_6,r_7,r_8,a,f)\rangle\end{array}$$

Our final specification for successful allocation, "allocation assigns fresh h to v_1 and updates heap h with a new element $(v_1,v_2,0)$", is stated using heap:

$$\begin{array}{l}\mathsf{cardinality}\ (\mathsf{reachables}\ [v_1,v_2,v_3,v_4,v_5,v_6]\ (set\ h)) < l\ \Rightarrow\\ \{\ \mathsf{heap}\ (a,l)\ (v_1,v_2,v_3,v_4,v_5,v_6,h) * \mathsf{s} * \mathsf{pc}\ p\ \}\\ \quad p : \texttt{E50A3018}, \texttt{E50A4014}, ..., \texttt{AAA8F004}\\ \{\ \mathsf{let}\ (v_1,h) = (\mathsf{fresh}\ h, h[\mathsf{fresh}\ h\mapsto(v_1,v_2,0)])\ \mathsf{in}\\ \quad\mathsf{heap}\ (a,l)\ (v_1,v_2,v_3,v_4,v_5,v_6,h) * \mathsf{s} * \mathsf{pc}\ (p+348)\ \}\end{array}$$

The specification for an unsuccessful allocation states that the ARM code assigns 0 to $v1$ if the heap is full, even after a collection cycle:

$$\begin{array}{l}\mathsf{cardinality}\ (\mathsf{reachables}\ [v_1,v_2,v_3,v_4,v_5,v_6]\ (set\ h)) \geq l\ \Rightarrow\\ \{\ \mathsf{heap}\ (a,l)\ (v_1,v_2,v_3,v_4,v_5,v_6,h) * \mathsf{s} * \mathsf{pc}\ p\ \}\\ \quad p : \texttt{E50A3018}, \texttt{E50A4014}, ..., \texttt{AAA8F004}\\ \{\ \mathsf{let}\ v_1 = 0\ \mathsf{in}\\ \quad\mathsf{heap}\ (a,l)\ (v_1,v_2,v_3,v_4,v_5,v_6,h) * \mathsf{s} * \mathsf{pc}\ (p+348)\ \}\end{array}$$

5.7.3 Vacuously true?

The specification above for memory allocation relies on multiple less than straight-forward definitions: heap, ch_rel, ch_inv, ch_word etc. This raises the questions: are there instantiations of the parameters which make the precondition true? And is the precondition accidentally equivalent to false making the specification vacuously true?

We argue that the specifications are not vacuously true by showing that we have code which can setup a heap satisfying heap from essentially no precondition at all. Given a continuous segment of memory f that is large enough to store a heap of size l at address a, the initialisation code sets up a heap with all roots set to zero and the function empty (a function with an empty domain domain empty $= \{\}$) describing the content of the heap.

$$
\begin{array}{l}
32 \leq \mathsf{w2n}\ a \wedge \mathsf{w2n}\ a + 2 \times 12 \times l + 12 < 2^{32} \wedge \\
\mathsf{domain}\ f = \{\ a + 4 \times \mathsf{n2w}\ i \mid i \leq 4 \times l + 2\ \} \cup \{\ a - 4 \times \mathsf{n2w}\ i \mid i \leq 8\ \} \Rightarrow \\
\{\ \mathsf{r3}\ r_3 * \mathsf{r4}\ r_4 * \mathsf{r5}\ r_5 * \mathsf{r6}\ r_6 * \mathsf{r7}\ r_7 * \mathsf{r8}\ r_8 * \mathsf{r9}\ a * \mathsf{memory}\ f * \mathsf{s} * \mathsf{pc}\ p\ \} \\
\quad p : \texttt{E50A3018}, \texttt{E50A4014}, ..., \texttt{AAA8F004} \\
\{\ \mathsf{heap}\ (a, l)\ (0, 0, 0, 0, 0, 0, \mathsf{empty}) * \mathsf{s} * \mathsf{pc}\ (p + 88)\ \}
\end{array}
$$

5.8 Proof reuse: verification of x86 and PowerPC code

Based on the ARM code for cheney_alloc, similar x86 and PowerPC code was written. Some features from the ARM code, e.g. conditional execution, load-and-increment in one instruction, were not possible to port, but instead a few special features of the two other languages were used, e.g. the x86 code (below) for move reduces code size by executing "`test edx, edx`" instead of "`cmp edx, 0`".

```
85D2                test edx, edx
7426                je L1
8B2A                mov ebp,[edx]
F7C501000000        test ebp,1
7519                jne L12
8929                mov [ecx],ebp
8B7204              mov esi,[edx+4]
8B6A08              mov ebp,[edx+8]
897104              mov [ecx+4],esi
896908              mov [ecx+8],ebp
89CD                mov ebp,ecx
45                  inc ebp
81C10C000000        add ecx,12
892A                mov [edx],ebp
89EA          L12:  mov edx,ebp
4A                  dec edx
              L1:
```

The x86 and PowerPC implementations were decompiled into functions x86_cheney_alloc and ppc_cheney_alloc, respectively. These were proved equivalent to the ARM implementation via an easy proof, using only α and β-conversion and a few lemmas about word arithmetic.

x86_cheney_alloc = arm_cheney_alloc
x86_cheney_alloc_pre = arm_cheney_alloc_pre
ppc_cheney_alloc = arm_cheney_alloc
ppc_cheney_alloc_pre = arm_cheney_alloc_pre

New versions of heap were defined for each architecture; the definition of heap for x86:

$$
\begin{array}{l}
\mathsf{xheap}\ (a,l)\ (v_1,v_2,v_3,v_4,v_5,v_6,h)\ = \\
\quad \exists x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ f. \\
\qquad \mathsf{eax}\ x_1 * \mathsf{ecx}\ x_2 * \mathsf{edx}\ x_3 * \mathsf{ebx}\ x_4 * \mathsf{ebp}\ x_5 * \mathsf{esi}\ x_6 * \mathsf{edi}\ a * \mathsf{memory}\ f * \\
\qquad \langle \mathsf{ch\_rel}\ ([v_1;v_2;v_3;v_4;v_5;v_6],h,l)\ (x_1,x_2,x_3,x_4,x_5,x_6,a,f)\rangle
\end{array}
$$

The theorems proved automatically by the decompiler were then used in constructing a specification for allocation, e.g. the x86 code for allocation satisfies the following specification:

$$
\begin{array}{l}
\mathsf{cardinality}\ (\mathsf{reachables}\ [v_1,v_2,v_3,v_4,v_5,v_6]\ (set\ h)) < l\ \Rightarrow \\
\{\ \mathsf{xheap}\ (a,l)\ (v_1,v_2,v_3,v_4,v_5,v_6,h) * \mathsf{s} * \mathsf{pc}\ p\ \} \\
\quad p : \texttt{56AA02}, \texttt{145F}, \ldots, \texttt{1A84} \\
\{\ \mathsf{let}\ (v_1,h) = (\mathsf{fresh}\ h, h[\mathsf{fresh}\ h \mapsto (v_1,v_2,0)])\ \mathsf{in} \\
\quad \mathsf{xheap}\ (a,l)\ (v_1,v_2,v_3,v_4,v_5,v_6,h) * \mathsf{s} * \mathsf{pc}\ (p+306)\ \}
\end{array}
$$

5.9 Discussion of related work

There is a large body of literature on the topic of specification and verification of garbage collection routines, e.g. [48, 34, 107, 42]. However, few have proved collectors correct with respect to detailed models of realistic execution environments. Notable exceptions are work by Birkedal et al. [13] and McCreight et al. [76]. Birkedal et al. use a slightly altered version of separation logic to verify, on paper it seems, the correctness of a C-like program implementing the Cheney algorithm for a stop-the-world collector. McCreight et al. developed a general framework for collector proofs, in Coq, and verified MIPS-like code for several different collector algorithms, including a stop-the-world and an incremental Cheney collector. McCreight et al. proved their stop-the-world Cheney collector in 7,775 lines of Coq, while the proof presented here required only approximately 2,000 lines of HOL4 proofs. The allocators verified by McCreight et al. and Birkedal et al. enter an infinite loop in case the heap is full after a complete collection cycle. In contrast, the collectors verified here terminate and return a null pointer.

Benton's specification and verification of a memory allocator [7] is also related to the work presented here. Benton verified, using Coq, an implementation of an allocator in an invented assembly language. Instead of using conventional unary predicates for describing program properties, he uses quantified binary relations and states program properties in terms of contextual equivalence. This allows him show that his allocator transfers ownership of memory states to the client program. The allocator specification presented here does not provide such clean logical separation, instead the allocator will always 'own' the allocated memory and the client is forced to view the heap as an abstraction of the real memory. However, it remains unclear whether the cost of adding these extra features to the specifications is worth the trouble, since his proofs seem to have been frustratingly hard work, as he comments in a separate note [8]. The proof of his allocator and its client the factorial program is some 8,500 lines long even though his allocator operates over an infinite memory, which means that the allocator will never run out of space and therefore does not need a garbage collector.

Chapter 6

Proof-producing compilation

This chapter presents a compiler which maps functions from logic, via proof, down to multiple, carefully modelled, commercial machine languages. Unlike previously published work on compilation from higher-order logic, the compiler allows input functions to be partially specified, and also provides broad support for user-defined extensions. The compiler generates machine code using untrusted programs which apply multiple optimising transformations. Subsequently, the generated machine code is proved to implement the original source program using the decompiler from Chapter 4.

6.1 Introduction

Assembly code is very hard to write, understand and maintain. That is why high-level languages are used for programming. Developers write programs in understandable high-level languages and then use *compilers* to translate these machine-independent programs into executable low-level implementations. Compilers are often complex applications that apply multiple optimising transformations to their input in order to generate efficient code.

Compilers pose a problem for program verification: if a high-level program is proved correct, then the compiler's transformation must be trusted in order for the proof to carry over to a guarantee about the generated executable code.

In practice there is also another problem: most source languages (C, Java, Haskell etc.) do not have a formal semantics, and it is therefore hard to formally state and verify properties of programs written in these languages.

This chapter explores an approach to compilation aimed at supporting program verification. We describe a compiler which takes as input a mathematical function (expressed in higher-order logic), compiles the function to machine code (ARM, x86 or PowerPC) and also proves that the generated code executes the supplied function. For example, given function f as input

$$\mathsf{f}(r_1) \quad = \quad \mathsf{if}\ \ r_1 < 10\ \ \mathsf{then}\ \ r_1\ \ \mathsf{else}\ \ \mathsf{let}\ r_1 = r_1 - 10\ \mathsf{in}\ \mathsf{f}(r_1)$$

the compiler can generate ARM machine code

```
E351000A      L:  cmp r1,#10
2241100A          subcs r1,r1,#10
2AFFFFFC          bcs L
```

and automatically prove a theorem which certifies that the generated code executes f. The following ARM Hoare-triple theorem (Chapter 3) states, if register one r1 initially holds value r_1, then the code will leave register one with value $\mathsf{f}(r_1)$.

$$\{\mathsf{r1}\ r_1 * \mathsf{pc}\ p * \mathsf{s}\}\ \ p : \texttt{E351000A}, \texttt{2241100A}, \texttt{2AFFFFFC}\ \ \{\mathsf{r1}\ \mathsf{f}(r_1) * \mathsf{pc}\ (p{+}12) * \mathsf{s}\}$$

The fact that f is expressed as a function in higher-order logic means that it has a precise semantics and that one can prove properties about f, e.g. one can prove that $\mathsf{f}(x) = x \bmod 10$ (here mod is modulus over unsigned machine words). Properties proved for f carry over to guarantees about the generated machine code via the certificate proved by the compiler. For example, one can rewrite the theorem from above to state that the code calculates $r_1 \bmod 10$:

$$\{\mathsf{r1}\ r_1 * \mathsf{pc}\ p * \mathsf{s}\}\ \ p : \texttt{E351000A}, \texttt{2241100A}, \texttt{2AFFFFFC}\ \ \{\mathsf{r1}\ (r_1 \bmod 10) * \mathsf{pc}\ (p{+}12) * \mathsf{s}\}$$

Compilation from logic, with a certificate of correctness, has been explored before, as will be discussed in Section 6.5. The contributions that distinguish the work presented here are that the described compiler:

1. targets multiple, carefully modelled, commercial machine languages;
2. allows input functions to be partially specified (in contrast to [64, 65, 66]);
3. supports significant user-defined extensions to its input language (Section 6.3.1); and
4. can, without added complexity, handle some optimising transformations (Section 6.4).

The compiler uses a restrictive input language which can either be extended directly, as discussed in Section 6.3.1, or used as a back-end to a compiler with a more general input language, e.g. [65, 66].

6.2 Core functionality

The compiler presented in this chapter accepts tail-recursive first-order HOL functions as input. As output it returns machine code together with a correctness certificate, a theorem which states that the generated machine code executes the function given as input.

The decompiler from Chapter 4 is used for the main proofs as can be seen by the following outline of the implemented algorithm:

1. generate machine code for input function f with an unverified algorithm;
2. use decompiler to prove that a function f' describes the behaviour of the code;
3. automatically prove $\mathsf{f} = \mathsf{f}'$.

Although it is difficult to estimate how total this compiler is, experience suggests that the compiler is nearly total, since the code generator will never produce code with control-flow structures that the decompiler can misunderstand (complex control-flow is the decompiler's weakness), and all optimisations introduced at stage 1 are performed in a manner reversible at stage 3. When the compiler does fail, it nearly always does so at stage 3. Failures at stage 3 provide information on how to correct the ML code which implements stage 1.

6.2.1 Input language

The compiler's input language consists of let-expressions, if-statements and tail-recursion. The language restricts variable names to correspond to names of registers or stack locations.

The following grammar describes the input language. Let r range over register names, r_0, r_1, r_2, etc., and s over stack locations, s_1, s_2, s_3 etc., m over memory modelling functions (mappings from aligned 32-bit machine words to 32-bit machine words), f over function names, g over names of already compiled functions, and i_5, i_7 and i_{32} over unsigned words of size 5-, 7- and 32-bits, respectively. Bit-operators &, $\otimes$, !!, $\ll$, $\gg$ are bitwise-and, bitwise-xor, bitwise-or, left-shift, right-shift. Operators suffixed with '.' are signed-versions of those without the suffix.

$$
\begin{array}{rcl}
\mathit{input} & ::= & f(v, v, ..., v) = \mathit{rhs} \\
& | & f(v, v, ..., v) = \mathit{rhs} \ \wedge \ \mathit{input} \\
\\
\mathit{rhs} & ::= & \mathsf{let}\ r = \mathit{exp}\ \mathsf{in}\ \mathit{rhs} \\
& | & \mathsf{let}\ s = r\ \mathsf{in}\ \mathit{rhs} \\
& | & \mathsf{let}\ m = m[\,\mathit{address} \mapsto r\,]\ \mathsf{in}\ \mathit{rhs} \\
& | & \mathsf{let}\ (v, v, ..., v) = g(v, v, ..., v)\ \mathsf{in}\ \mathit{rhs} \\
& | & \mathsf{if}\ \mathit{guard}\ \mathsf{then}\ \mathit{rhs}\ \mathsf{else}\ \mathit{rhs} \\
& | & f(v, v, ..., v) \\
& | & (v, v, ..., v) \\
\\
\mathit{exp} & ::= & x \mid \neg\, x \mid s \mid x\ \mathit{binop}\ x \mid m\ \mathit{address} \mid x \ll i_5 \mid x \gg i_5 \mid x \gg. i_5 \\
\mathit{binop} & ::= & + \mid - \mid \times \mid \mathsf{div} \mid \& \mid \otimes \mid !! \\
\mathit{cmp} & ::= & < \mid \leq \mid > \mid \geq \mid <. \mid \leq. \mid >. \mid \geq. \mid = \\
\mathit{guard} & ::= & \neg\ \mathit{guard} \mid \mathit{guard} \wedge \mathit{guard} \mid \mathit{guard} \vee \mathit{guard} \mid x\ \mathit{cmp}\ x \mid x\ \&\ x = 0 \\
\mathit{address} & ::= & r \mid r + i_7 \mid r - i_7 \\
\\
x & ::= & r \mid i_{32} \\
v & ::= & r \mid s \mid m
\end{array}
$$

This input language was designed to be machine independent; programs constructed from this grammar can be compiled to any of the target languages: ARM, x86 and PowerPC. However the input language differs for each target in the number of registers available ($r_0...r_{12}$ for ARM, $r_0...r_6$ for x86 and $r_0...r_{31}$ for PowerPC) and restrictions on the use of multiplication. Further work will be to add a register allocator which will make the input language independent of the target language.

6.2.2 Code generation

The input language was defined to mimic the operations of machine instructions in order to ease code generation. Each let-expression usually produces a single instruction, e.g.

let $r_3 = r_3 + r_2$ in	generates ARM code	`add r3,r3,r2`
let $r_3 = r_3 + r_2$ in	generates x86 code	`add ebx,edx`
let $r_3 = r_3 + r_2$ in	generates PowerPC code	`add 3,3,2`

In some cases one let-expression is split into a few instructions, e.g.

let $r_3 = r_0 - r_2$ in	generates x86 code	`mov ebx,eax` `sub ebx,edx`
let $r_3 = 5000$ in	generates ARM code	`mov r3,#136` `or r3,r3,#19,lsl 8`

The code generator was programmed to use a few assembly tricks, e.g. on x86 certain instances of addition, which would normally require two instructions (`mov` followed by `add`), can be implemented in one instruction using the load-effective-address instruction `lea`:

let $r_3 = r_0 + r_2$ in	generates x86 code	`lea ebx,[eax+edx]`

A combination of compare and branch are used to implement if-statements, e.g.

if $r_3 = 45$ then ... else ...	generates ARM code	`cmp r3,#45` `bne ELSE1`

Recursive calls and function returns generate branch instructions.

The compiler generates a list of assembly instructions, which is translated into machine code using off-the-shelf assemblers: Netwide Assembler `nasm` for x86 and the GNU Assembler `gas` for ARM and PowerPC. Note that these external tools do not need to be trusted. If incorrect code is generated then the certification phase, which is to prove the correctness certificate, will fail.

6.2.3 Proving the correctness theorem

Given a function f, the compiler generates machine code c for executing f. Proving that the compiler's transformation of f into c is correct, i.e. that code c actually executes function

f, is done via the decompiler, as the following example illustrates. Suppose the following recursive function f is given as input to the compiler.

$$
\begin{array}{l}
\mathsf{f}(r_0, r_1, m) = \\
\quad \mathsf{if}\ r_0 = 0\ \mathsf{then}\ (r_0, r_1, m)\ \mathsf{else} \\
\quad\quad \mathsf{let}\ r_1 = m(r_1)\ \mathsf{in} \\
\quad\quad \mathsf{let}\ r_0 = r_0 - 1\ \mathsf{in} \\
\quad\quad\quad \mathsf{f}(r_0, r_1, m)
\end{array}
$$

If f is to be compiled to x86 code then the compiler generates:

```
0:  85C0        L1:  test eax, eax
2:  7405             jz L2
4:  8B09             mov ecx,[ecx]
6:  48               dec eax
7:  EBF7             jmp L1
                L2:
```

The decompiler is used to derive a function f' that describes the effect of the code.

$$
\begin{array}{l}
\mathsf{f}'(eax, ecx, m) = \\
\quad \mathsf{if}\ eax\ \&\ eax = 0\ \mathsf{then}\ (eax, ecx, m)\ \mathsf{else} \\
\quad\quad \mathsf{let}\ ecx = m(ecx)\ \mathsf{in} \\
\quad\quad \mathsf{let}\ eax = eax - 1\ \mathsf{in} \\
\quad\quad\quad \mathsf{f}'(eax, ecx, m)
\end{array}
$$

Decompilation also proves a theorem which states that f′ is executed by the x86 code:

$$
\begin{array}{l}
\mathsf{f}'_{\mathsf{pre}}(eax, ecx, m) \Rightarrow \\
\{\, (\mathsf{eax}, \mathsf{ecx}, \mathsf{m})\ \mathsf{is}\ (eax, ecx, m) * \mathsf{eip}\ p * \mathsf{s} \,\} \\
\ p : \texttt{85C074058B0948EBF7} \\
\{\, (\mathsf{eax}, \mathsf{ecx}, \mathsf{m})\ \mathsf{is}\ \mathsf{f}'(eax, ecx, m) * \mathsf{eip}\ (p{+}9) * \mathsf{s} \,\}
\end{array}
$$

Next the compiler proves f = f′. Both f and f′ are recursive functions; thus proving f = f′ would normally require an induction. The compiler can avoid an induction since both f and f′ are defined as instances of tailrec from Section 4.3.5:

$$\mathsf{tailrec}\ x = \mathsf{if}\ G\ x\ \mathsf{then}\ \mathsf{tailrec}\ (F\ x)\ \mathsf{else}\ (D\ x)$$

The compiler proves f = f′ by proving that the components of the tailrec instantiation are equal, i.e. for f and f′, as given above, the compiler only needs to prove the following. Here let-expressions have been expanded.

$$
\begin{array}{rcl}
\lambda(r_0, r_1, r_2).\ r_2 = 0 & = & \lambda(eax, ecx, ebx).\ ebx\ \&\ ebx = 0 \\
\lambda(r_0, r_1, r_2).\ (r_0, r_1, r_2) & = & \lambda(eax, ecx, ebx).\ (eax, ecx, ebx) \\
\lambda(r_0, r_1, r_2).\ (r_0, r_0{+}r_0{-}r_1, r_0{-}r_1) & = & \lambda(eax, ecx, ebx).\ (eax, eax{+}eax{-}ecx, ebx{-}ecx)
\end{array}
$$

The code generation phase is programmed in such a way that the above component proofs will always be proved by an expansion of let-expressions followed by rewriting with a handful of verified rewrite rules that undo assembly tricks, e.g. $\forall w.\ w \,\&\, w = w$.

The precondition $\mathsf{f'_{pre}}$ is not translated, instead $\mathsf{f_{pre}}$ is defined to be $\mathsf{f'_{pre}}$, which in this case is the following. Here let-expressions have been expanded.

$$\mathsf{f'_{pre}}(eax, ecx, ebx) \quad = \quad (ebx \,\&\, ebx \neq 0 \;\Rightarrow\; \mathsf{f'_{pre}}(eax, eax{+}eax{-}ecx, ebx{-}ecx))$$

The compiler proves the certificate of correctness by rewriting the output from the decompiler using theorems $\mathsf{f'} = \mathsf{f}$ and $\mathsf{f'_{pre}} = \mathsf{f_{pre}}$. The example results in the following theorem.

$$\{\ (\mathsf{eax}, \mathsf{ecx}, \mathsf{ebx})\ \mathsf{is}\ (eax, ecx, ebx) * \mathsf{eip}\ p * \langle \mathsf{f_{pre}}(eax, ecx, ebx) \rangle\ \}$$
$$\ldots\ x86\ code\ \ldots$$
$$\{\ (\mathsf{eax}, \mathsf{ecx}, \mathsf{ebx})\ \mathsf{is}\ \mathsf{f}(eax, ecx, ebx) * \mathsf{eip}\ (p{+}11)\ \}$$

6.3 Extensions, stacks and subroutines

The examples above outlined the algorithm of the compiler based on simple examples involving only registers. This section presents how the compiler supports user-defined extensions, stack operations and subroutine calls.

6.3.1 User-defined extensions

The compiler has a restrictive input language. User-defined extensions to this input language are thus vital in order to be able to make use of the features specific to each target language.

User-defined extensions to the input language are made possible by the proof method which derives a function $\mathsf{f'}$ describing the effect of the generated code: function $\mathsf{f'}$ is constructed by composing together Hoare triples describing parts of the generated code. By default, automatically derived Hoare triples for each individual machine instruction are used. However, the user can instead supply the proof method with alternative Hoare triples in order to build on previously proved theorems.

An example will illustrate how this observation works in practice. Given the following Hoare triple (proved in Section 6.1) which shows that ARM machine code has been proved to implement "r_1 is assigned $r_1 \bmod 10$",

$$\{\mathsf{r1}\ r_1 * \mathsf{pc}\ p * \mathsf{s}\}\ \ p : \texttt{E351000A}, \texttt{2241100A}, \texttt{2AFFFFFC}\ \ \{\mathsf{r1}\ (r_1 \bmod 10) * \mathsf{pc}\ (p{+}12) * \mathsf{s}\}$$

the code generator expands its input language for ARM with the following line:

$$rhs \quad ::= \quad \mathsf{let}\ r_1 = r_1 \bmod 10\ \mathsf{in}\ rhs$$

Now when a function f is to be compiled which uses this feature,

$$\begin{aligned} \mathsf{f}(r_1, r_2, r_3) \quad = \quad & \mathsf{let}\ r_1 = r_1 + r_2\ \mathsf{in} \\ & \mathsf{let}\ r_1 = r_1 + r_3\ \mathsf{in} \\ & \mathsf{let}\ r_1 = r_1\ \mathsf{mod}\ 10\ \mathsf{in} \\ & \quad r_1 \end{aligned}$$

the code generator implements "let $r_1 = r_1$ mod 10 in" using the machine code (tagged with `insert` below) found inside the Hoare triple.

```
                    E0811002            add r1,r1,r2
                    E0811003            add r1,r1,r3
( insert: mod10,    E351000A       L:   cmp r1,#10
                    2241100A            subcs r1,r1,#10
                    2AFFFFFC )          bcs L
```

The compiler would now normally derive f' by composing Hoare triples for the individual machine instructions, but in this case the compiler considers the tagged code as a 'single instruction' whose effect is described by the supplied Hoare triple. It composes the following Hoare triples, in order to derive a Hoare triple for the entire code.

$$\{\mathsf{r1}\ r_1 * \mathsf{r2}\ r_2 * \mathsf{pc}\ p\}\ \ p : \texttt{E0811002}\ \ \{\mathsf{r1}\ (r_1{+}r_2) * \mathsf{r2}\ r_2 * \mathsf{pc}\ (p{+}4)\}$$

$$\{\mathsf{r1}\ r_1 * \mathsf{r3}\ r_3 * \mathsf{pc}\ p\}\ \ p : \texttt{E0811003}\ \ \{\mathsf{r1}\ (r_1{+}r_3) * \mathsf{r3}\ r_3 * \mathsf{pc}\ (p{+}4)\}$$

$$\{\mathsf{r1}\ r_1 * \mathsf{pc}\ p * \mathsf{s}\}\ \ p : \texttt{E351000A}, \texttt{2241100A}, \texttt{2AFFFFFC}\ \ \{\mathsf{r1}\ (r_1 \bmod 10) * \mathsf{pc}\ (p{+}12) * \mathsf{s}\}$$

The resulting f' is trivially equal to f and thus the resulting Hoare triple states that the generated code actually executes f.

$$\begin{array}{l}\{\mathsf{r1}\ r_1 * \mathsf{r2}\ r_2 * \mathsf{r3}\ r_3 * \mathsf{pc}\ p * \mathsf{s}\}\\ p : \texttt{E0811002}, \texttt{E0811003}, \texttt{E351000A}, \texttt{2241100A}, \texttt{2AFFFFFC}\\ \{\mathsf{r1}\ \mathsf{f}(r_1, r_2, r_3) * \mathsf{r2}\ r_2 * \mathsf{r3}\ r_3 * \mathsf{pc}\ (p{+}20) * \mathsf{s}\}\end{array}$$

It is important to note that the Hoare triples supplied to the compiler need not concern registers or memory locations, instead more abstract Hoare triples can be supplied. For example in the next chapter, the compiler is given Hoare triples that show how basic operations over LISP s-expressions can be performed. The LISP operation car over a heap of s-expressions is implemented by ARM instruction `ldr r3,[r3]`, encoded as `E5933000`.

$$\begin{array}{l}(\exists x\ y.\ v_1 = \mathsf{Dot}\ x\ y)\ \Rightarrow\\ \{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p\ \}\\ p : \texttt{E5933000 [ldr r3,[r3]]}\\ \{\ \mathsf{lisp}\ (\mathsf{car}\ v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p+4)\ \}\end{array}$$

The above specification extends the code generator for with the assignments of car v_1 to s-expression variable v_1.

$$rhs \quad ::= \quad \mathsf{let}\ v_1 = \mathsf{car}\ v_1\ \mathsf{in}\ rhs$$

6.3.2 Stack usage

The stack can be used by assignments from and to variables s_0, s_1, s_2 etc., e.g. the following let-expressions correspond to machine code which loads register 1 from stack location 3

(three down from top of stack), adds 78 to register 1 and then stores the result in stack location 2.

$$\begin{array}{rcl} \mathsf{f}(r_1, s_2, s_3) & = & \mathsf{let}\ r_1 = s_3\ \mathsf{in} \\ & & \mathsf{let}\ r_1 = r_1 + 78\ \mathsf{in} \\ & & \mathsf{let}\ s_2 = r_1\ \mathsf{in} \\ & & \quad (r_1, s_2, s_3) \end{array}$$

Internally stack accesses are implemented by supplying the decompiler with specifications which specify stack locations using mem assertions (Section 3.3), e.g. the following is the specification used for reading the value of stack location 3 into register 1. Register 13 is the stack pointer.

$$\{\mathsf{r1}\ r_1 * \mathsf{r13}\ sp * \mathsf{mem}\ (sp{+}12)\ s_3 * \mathsf{pc}\ p * \mathsf{s}\}$$
$$p : \texttt{E59D100C [ldr r1,[r13,\#12]]}$$
$$\{\mathsf{r1}\ s_3 * \mathsf{r13}\ sp * \mathsf{mem}\ (sp{+}12)\ s_3 * \mathsf{pc}\ (p{+}4) * \mathsf{s}\}$$

The postcondition for the certification theorem proved for the above let-expressions:

$$\{\ (\mathsf{r1}, \mathsf{mem}\ (sp{+}8), \mathsf{mem}\ (sp{+}12))\ \mathsf{is}\ \mathsf{f}(r_1, s_2, s_3)\ *\ \mathsf{r13}\ sp\ *\ \mathsf{pc}\ (p{+}12)\ *\ \mathsf{s}\ \}$$

6.3.3 Subroutines and procedures

Subroutines can be in-lined and called as procedures. Each compilation adds a new let-expression into the input languages of the compiler. The added let-expressions describe the compiled code, i.e. they allow subsequent compilations to use the previously compiled code. For example, when the following function (which uses f from above) is compiled, the code for f will be in-lined.

$$\begin{array}{rcl} \mathsf{g}(r_1, r_2, s_2, s_3) & = & \mathsf{let}\ (r_1, s_2, s_3) = \mathsf{f}(r_1, s_2, s_3)\ \mathsf{in} \\ & & \mathsf{let}\ s_2 = r_1 + r_2\ \mathsf{in} \\ & & \quad (r_1, r_2, s_2, s_3) \end{array}$$

Note that each call to subroutine f must state the input and output exactly as "(r_1, s_2, s_3)", since the variable names are tied to real resource names. The code compiled for f expects the first argument in register 1, thus r_1.

If the compiler is asked to compile f as a procedure (essentially append a return instruction to the end of the code for f), then the stack variables need to change, as shown below, because the stack pointer is shifted by one position (return address stored on the stack). However, the definition of f is still the same as above, i.e. in terms of s_2 and s_3.

$$\begin{array}{rcl} \mathsf{g}(r_1, r_2, s_1, s_2) & = & \mathsf{let}\ (r_1, s_1, s_2) = \mathsf{f}(r_1, s_1, s_2)\ \mathsf{in} \\ & & \mathsf{let}\ s_2 = r_1 + r_2\ \mathsf{in} \\ & & \quad (r_1, r_2, s_1, s_2) \end{array}$$

6.4 Optimising transformations

Given a function f, the decompiler generates code, which it decompiles to produce function f' describing the behaviour of the generated code. The decompiler can perform any optimisations as long as it can at the end prove $\mathsf{f} = \mathsf{f}'$. In particular, certain instructions can

be reordered or removed, and the code's control flow can use special features of the target language.

6.4.1 Instruction reordering

Instruction reordering is a standard optimisation applied in order to avoid unnecessary pipeline stalls. The compiler presented here supports instruction reordering as is illustrated by the following example. Given a function f which stores r_1 into stack location s_5, then loads r_2 from stack location s_6, and finally adds r_1 and r_2.

$$\begin{array}{rcl} \mathsf{f}(r_1, r_2, s_5, s_6) & = & \mathsf{let}\ s_5 = r_1\ \mathsf{in} \\ & & \mathsf{let}\ r_2 = s_6\ \mathsf{in} \\ & & \mathsf{let}\ r_1 = r_1 + r_2\ \mathsf{in} \\ & & \quad (r_1, r_2, s_5, s_6) \end{array}$$

The code corresponding directly to f might cause a pipeline stall as the result of the load instruction ($\mathsf{let}\ r_2 = s_6\ \mathsf{in}$) may not be available on time for the add instruction ($\mathsf{let}\ r_1 = r_1 + r_2\ \mathsf{in}$). It is therefore beneficial to schedule the load instructions as early as possible; the generated code reduces the risk of a pipeline stall by placing the load instruction before the store instruction:

$$\begin{array}{rcl} \mathsf{f}'(r_1, r_2, s_5, s_6) & = & \mathsf{let}\ r_2 = s_6\ \mathsf{in} \\ & & \mathsf{let}\ s_5 = r_1\ \mathsf{in} \\ & & \mathsf{let}\ r_1 = r_1 + r_2\ \mathsf{in} \\ & & \quad (r_1, r_2, s_5, s_6) \end{array}$$

Valid reorderings of instructions are unnoticeable after expansion of let-expressions, thus the proof of $\mathsf{f} = \mathsf{f}'$ does not need to be smarter to handle this optimisation.

6.4.2 Removal of dead code

Live-variable analysis can be applied to the code in order to remove unused or *dead code*. In the following definition of f, the first let-expression is unnecessary.

$$\begin{array}{rcl} \mathsf{f}(r_1, r_2, s_5, s_6) & = & \mathsf{let}\ r_1 = s_5\ \mathsf{in} \\ & & \mathsf{let}\ r_2 = s_6\ \mathsf{in} \\ & & \mathsf{let}\ r_1 = r_2 + 8\ \mathsf{in} \\ & & \quad (r_1, r_2, s_5, s_6) \end{array}$$

The generated code ignores the first let-expression and produces a function f' which is, after expansion of let-expressions, identical to f.

6.4.3 Conditional execution

ARM machine code allows conditional execution of nearly all instructions in order to allow short forward jumps to be replaced by conditionally executed instructions (this reduces

branch overhead). The compiler produces conditionally-executed instruction blocks where short jumps would otherwise have been generated. The functions decompiled from conditionally executed instructions are indistinguishable from those decompiled from code with normal jumps (as can be seen in the examples of Section 4.2).

x86 supports conditional assignment using the conditional-move instruction `cmov`. For x86, the compiler replaces jumps across register-register moves by conditional-move instructions.

6.4.4 Shared tails

The compiler's input language supports if-statements that split control, but does not provide direct means for joining control-flow. For example, a program such as

```
(if r1 = 0 then r2 := 23 else r2 := 56); r1 := 4
```

is defined either directly as function f with duplicate tails

$$\begin{array}{rcl} \mathsf{f}(r_1, r_2) & = & \mathsf{if}\ r_1 = 0\ \mathsf{then\ let}\ r_2 = 23\ \mathsf{in\ let}\ r_1 = 4\ \mathsf{in}\ (r_1, r_2) \\ & & \quad \mathsf{else\ let}\ r_2 = 56\ \mathsf{in\ let}\ r_1 = 4\ \mathsf{in}\ (r_1, r_2) \end{array}$$

or as function g with auxiliary function g_2:

$$\begin{array}{rcl} \mathsf{g}(r_1, r_2) & = & \mathsf{let}\ (r_1, r_2) = \mathsf{g}_2(r_1, r_2)\ \mathsf{in\ let}\ r_1 = 4\ \mathsf{in}\ (r_1, r_2) \\ \mathsf{g}_2(r_1, r_2) & = & \mathsf{if}\ r_1 = 0\ \mathsf{then\ let}\ r_2 = 23\ \mathsf{in}\ (r_1, r_2) \\ & & \quad \mathsf{else\ let}\ r_2 = 56\ \mathsf{in}\ (r_1, r_2) \end{array}$$

Generating code naively for f would result in two instructions for $\mathsf{let}\ r_1 = 4\ \mathsf{in}$, one for each branch. The compiler implements an optimisation which detects 'duplicate tails' so that the code for f will be identical to that produced for g. The compiler generates the following ARM code for function g (using conditional execution to avoid inserting short jumps).

```
 0:  E3510000          cmp r1,#0
 4:  03A02017          moveq r2,#23
 8:  13A02038          movne r2,#56
12:  E3A01004          mov r1,#4
```

6.5 Discussion of related work

This chapter has presented how an extensible proof-producing compiler can be implemented using decompilation from Chapter 4. The implementation required only a light-weight certification phase (approximately 100 lines of ML code) to be programmed, but still proves full functional equivalence between the source functions and the generated machine code.

The compiler presented here was inspired by proof-producing compilers for synthesis of hardware by Iyoda et al. [46], and ARM-like assembly programs by Li et al. [64]. The

compilers mentioned above have input languages similar to the one presented here, although more recent improvements [66, 65] to the front-end of the software compiler relax some of the restrictions (tail-recursion, monomorphism). The hardware compiler operates using a number of refinement transformations, while newer instances of the software compiler are largely implemented using rewriting [65]. The most distinguishing feature of the compiler presented here is its support for user-defined extensions as well as the distinct separation between code generation and certification proof.

The method proposed in this chapter is to generate code in an unverified manner, but then afterwards automatically check the correctness of the generated code. Pnueli et al. [100] call this approach *translation validation* and have proposed a similar method before. Pnueli et al. showed how translation validation can be implemented for a compiler which maps synchronous multi-clock data-flow language SIGNAL to asynchronous C code. In the spirit of proof-carrying code, Necula [93] showed that translation validation can scale to conventional, moderately optimising compilers such as the GNU C compiler version 2. Pnueli et al. and Necula built their verifiers as standalone tools: Necula's tool consists of over 10,000 lines of trusted Objective CAML code. The compiler described here instead steers an off-the-shelf theorem prover (HOL4) to a proof. Thus our resulting certificate theorems are immediately usable in further automatic and interactive proofs within the framework of a programmable general-purpose theorem prover.

An alternative to producing a proof for each run is to prove the compiler correct once and for-all: a formal connection between source and target code is then achieved by simply instantiating the theorem describing the compiler's correctness. A recent, particularly impressive, milestone in compiler verification was achieved by Leroy [63] who proved that his optimising C compiler maps C programs to observationally equivalent PowerPC assembly code[1]. As part of this project, Tristan and Leroy [113] verified multiple translation validators. We chose not to verify our compiler/translation validator, since it seems that the user-defined extensions, such as those used in the next chapter, would have been much harder to implement in a verified compiler; verifying a compiler involves defining a deep embedding of the input language. The trusted computing base (TCB) of the compiler presented here is HOL4 and the specifications of the target machine languages.

Other recent work on compiler verifications include the following. Klein and Nipkow verified a compiler from a Java-like language to JVM-like byte-code, as part of project which formalised in Isabelle/HOL neat idealised versions of the Java language, the JVM, a compiler from Java to JVM and a byte-code verifier [58]. Chlipala has proved, in Coq, type preservation for a compiler from simply-typed lambda terms to an invented assembly language [24]. Chlipala chose a very simple source and target language but paid particular attention to nested variable scopes, first-class functions and dynamic allocation. Meyer and Wolff has showed how Isabelle/HOL can be used for compiling and optimising, in a trustworthy manner, the lazy function calls of 'MiniHaskell' into a strict language, which they call 'MiniML' [77]. And, Benton and Zarfaty have verified semantic type soundness of a compiler from a simple high-level language with heap allocated data to an idealised assem-

[1] The work presented here builds on Leroy's specification of PowerPC assembly code.

bly language. Benton and Zarfaty's work build on Benton's earlier work on reasoning about low-level code using binary relations as types [7], mentioned in Section 5.9.

Chapter 7

Verified LISP interpreters

This chapter shows how the techniques presented in earlier chapters can be used to construct verified applications. The compiler from Chapter 6 is used to construct correct machine-code implementations of a LISP interpreter. Unlike previous work on verification of LISP interpreters, this work maps proofs down to detailed models of commercial machine languages.

7.1 Introduction

Programs are written in a wide variety of programming languages, ranging from languages such as assembly and C, where pointers are explicit, to languages such as Java, Python, and Haskell, where data-structures are lists and trees. Programs are most often written in low-level languages when efficiency of the resulting code is a priority. High-level languages are used when speed of program development and ease of program maintenance are more important priorities.

This thesis has mainly concerned programs at the very lowest levels of abstraction, namely machine and assembly languages. A connection to high-level languages is made in this chapter, by showing how a verified implementation of functional language can be constructed using techniques presented in earlier chapters. A case study is presented which produced formally verified ARM, x86 and PowerPC machine code that parses, evaluates and prints LISP. This work seems to be the first to produce a formally verified end-to-end implementation of a functional programming language.

For a flavour of what has been implemented and proved consider an example: if the implementation is supplied with the following call to `pascal-triangle`,

```
(pascal-triangle '((1)) '6)
```

it parses the string, evaluates the expression and prints a string,

```
((1 6 15 20 15 6 1)
 (1 5 10 10 5 1)
 (1 4 6 4 1)
 (1 3 3 1)
 (1 2 1)
 (1 1)
 (1))
```

where `pascal-triangle` had been supplied to it as

```
(label pascal-triangle
  (lambda (rest n)
    (cond ((equal n '0) rest)
          ('t (pascal-triangle
                (cons (pascal-next '0 (car rest)) rest) (- n '1))))))
```

with auxiliary function:

```
(label pascal-next
  (lambda (p xs)
    (cond ((atom xs) (cons p 'nil))
          ('t (cons (+ p (car xs)) (pascal-next (car xs) (cdr xs)))))))
```

The theorem which was proved about the LISP implementation can be used to show e.g. that running `pascal-triangle` will terminate and print the first $n + 1$ rows of Pascal's triangle, without a premature exit due to lack of heap space. One can use our theorem to derive sufficient conditions on the inputs to guarantee that there will be enough heap space.

It is envisioned that this verified LISP interpreter will provide a platform on top of which formally verified software can be produced with much greater ease than at lower levels of abstraction, i.e. in languages where pointers are made explicit.

Why was LISP chosen? LISP was chosen since LISP has a neat definition of both syntax and semantics [74] and is still a very powerful language as one can see, for example, in the success of ACL2 [56].

7.2 Methodology

Instead of delving into the many detailed invariants developed for our proofs, this chapter will concentrate on describing the methodology that was used:

- ▷ First, machine code for various LISP primitives, such as car, cdr, cons, was written and verified (Section 7.3);
  - – The correctness of each code snippets is expressed as a machine-code Hoare triple from Chapter 3: $\{\, pre * \mathsf{pc}\ p \,\}\ p : code\ \{\, post * \mathsf{pc}\ (p + exit) \,\}$.

  - Primitives cons and equal were verified using decompilation from Chapter 4.

- ▷ Second, the verified LISP primitives were input into the proof-producing compiler, from Chapter 6, in such a way that the compiler can view the processors as a machine with six registers containing LISP s-expressions (Section 7.4);
  - The compiler from Chapter 6 maps tail-recursive functions, defined in the logic of HOL4, down to machine code and proves that the generated code executes the original HOL4 functions.
  - Theorems describing the LISP primitives were input into the compiler, which can use them as building blocks when deriving new code/proofs.
- ▷ Third, LISP evaluation was defined as a (partially-specified) tail-recursive function lisp_eval, and then compiled into machine code using the compiler mentioned above (Section 7.5).
  - LISP evaluation was defined as a tail-recursive function which only uses expressions/names for which the compiler has verified building blocks.
  - The function lisp_eval maintains a stack and a symbol-value list.
- ▷ Fourth, to gain confidence that lisp_eval implements 'LISP evaluation', a theorem was proved which states that lisp_eval implements a semantics of LISP 1.5 [74] (Section 7.6).
  - Gordon's relational semantics of pure LISP [45], which covers the core of McCarthy's original LISP 1.5 [74], was used here.
  - The semantics abstracts the stack and certain evaluation orders.
- ▷ Finally, the verified LISP interpreters were sandwiched between a verified parser and printer to produce string-to-string theorems describing the behaviour of the entire implementation (Section 7.7).
  - The parser and printer code, respectively, sets up and tears down an appropriate heap for s-expressions.

Section 7.8 gives some quantitative data on the effort and Section 7.9 discusses related work.

7.3 LISP primitives

LISP programs are expressed in and operate over s-expressions, expressions that are either a number, a symbol or a pair of s-expressions. In HOL, s-expressions are readily modelled abstractly using a data-type with constructors:

$$\begin{aligned} \mathsf{Num} &: \mathbb{N} \to \mathsf{SExp} \\ \mathsf{Sym} &: \mathsf{string} \to \mathsf{SExp} \\ \mathsf{Dot} &: \mathsf{SExp} \to \mathsf{SExp} \to \mathsf{SExp} \end{aligned}$$

LISP programs and s-expressions are conventionally written in an abbreviated string form. A few examples will illustrate the correspondence, which is given a formal definition in Section 7.7 (Figure 7.4).

`(car x)`	means	Dot (Sym `"car"`) (Dot (Sym `"x"`) (Sym `"nil"`))
`(1 2 3)`	means	Dot (Num 1) (Dot (Num 2) (Dot (Num 3) (Sym `"nil"`)))
`'f`	means	Dot (Sym `"quote"`) (Dot (Sym `"f"`) (Sym `"nil"`))
`(4 . 5)`	means	Dot (Num 4) (Num 5)

Some basic LISP primitives are defined over SExp as follows:

$$\begin{aligned}
\mathsf{car}\ (\mathsf{Dot}\ x\ y) &= x \\
\mathsf{cdr}\ (\mathsf{Dot}\ x\ y) &= y \\
\mathsf{cons}\ x\ y &= \mathsf{Dot}\ x\ y \\
\\
\mathsf{plus}\ (\mathsf{Num}\ m)\ (\mathsf{Num}\ n) &= \mathsf{Num}\ (m+n) \\
\mathsf{minus}\ (\mathsf{Num}\ m)\ (\mathsf{Num}\ n) &= \mathsf{Num}\ (m-n) \\
\mathsf{times}\ (\mathsf{Num}\ m)\ (\mathsf{Num}\ n) &= \mathsf{Num}\ (m \times n) \\
\mathsf{division}\ (\mathsf{Num}\ m)\ (\mathsf{Num}\ n) &= \mathsf{Num}\ (m\ \mathsf{div}\ n) \\
\mathsf{modulus}\ (\mathsf{Num}\ m)\ (\mathsf{Num}\ n) &= \mathsf{Num}\ (m\ \mathsf{mod}\ n) \\
\\
\mathsf{equal}\ x\ y &= \mathsf{if}\ x = y\ \mathsf{then}\ \mathsf{Sym}\ \texttt{"t"}\ \mathsf{else}\ \mathsf{Sym}\ \texttt{"nil"} \\
\mathsf{less}\ (\mathsf{Num}\ m)\ (\mathsf{Num}\ n) &= \mathsf{if}\ m < n\ \mathsf{then}\ \mathsf{Sym}\ \texttt{"t"}\ \mathsf{else}\ \mathsf{Sym}\ \texttt{"nil"}
\end{aligned}$$

In the definition of equal, the equality $x = y$ means structural equality.

7.3.1 Specification of primitive operations

Before writing and verifying the machine code implementing primitive LISP operations, a decision had to be made how to represent Num, Sym and Dot on a real machine. To keep memory usage to a minimum each Dot-pair is represented as a block of two pointers stored consecutively on the heap, each Num n is represented as a 32-bit word containing $4 \times n + 2$ (i.e. only natural numbers $0 \leq n < 2^{30}$ are representable), and each Sym s is represented as a 32-bit word containing $4 \times i + 3$, where i is the row number of symbol s in a symbol table which, in our implementation, is a linked-list kept outside of the garbage-collected heap.

Here '+2' and '+3' are used as tags to make sure that the garbage collector can distinguish Num and Sym values from proper pointers. Pointers to Dot-pairs are word-aligned, i.e. a mod $4 = 0$, a which can be tested in machine code by computing a & $3 = 0$, where & is bitwise-and.

This simple and small representation of SExp allows most LISP primitives from the previous section to be implemented in one or two machine instructions. For example, taking car of register 3 and storing the result in register 4 is implemented on ARM as a load instruction:

```
E5934000   ldr r4,[r3]    (* load into reg 4, memory at address reg 3 *)
```

Similarly, ARM code for performing LISP operation plus of register 3 and 4, and storing the result into register 3 is implemented by:

```
E0833004   add r3,r3,r4   (* reg 3 is assigned value reg 3 + reg 4 *)
E2433002   sub r3,r3,#2   (* reg 3 is assigned value reg 3 - 2 *)
```

The intuition here is: $(4 \times m + 2) + (4 \times n + 2) - 2 = 4 \times (m + n) + 2$.

The correctness of the above implementations of car and plus is expressed formally by the two ARM Hoare triples below. Here lisp $(v_1, v_2, v_3, v_4, v_5, v_6, l)$ is an assertion, defined below, which asserts that a heap with room for l Dot-pairs is located in memory and that s-expressions $v_1 ... v_6$ (each of type SExp) are stored in machine registers. This lisp assertion should be understood as lifting the level of abstraction to a level where specific machine instructions make the processor seem as if it has six[1] registers containing s-expressions, of type SExp.

$$
\begin{array}{l}
(\exists x\ y.\ \mathsf{Dot}\ x\ y = v_1)\ \Rightarrow \\
\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p\ \} \\
\ p : \texttt{E5934000 [ldr r4,[r3]]} \\
\{\ \mathsf{lisp}\ (v_1, \mathsf{car}\ v_1, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p+4)\ \}
\end{array}
$$

$$
\begin{array}{l}
(\exists m\ n.\ \mathsf{Num}\ m = v_1 \wedge \mathsf{Num}\ n = v_2 \wedge m{+}n < 2^{30})\ \Rightarrow \\
\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p\ \} \\
\ p : \texttt{E0833004 E2433002 [add r3,r3,r4; sub r3,r3,\#2]} \\
\{\ \mathsf{lisp}\ (\mathsf{plus}\ v_1\ v_2, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p+8)\ \}
\end{array}
$$

The new assertion is defined for ARM (lisp), x86 (lisp'), and PowerPC (lisp") as maintaining a relation lisp_inv between the abstract state $v_1 ... v_6$ (each of type SExp) and the concrete state $x_1 ... x_6$ (each of type 32-bit word). The details of lisp_inv (defined in Figure 7.1) are unimportant for this presentation.

$$
\begin{array}{l}
\mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l)\ = \\
\quad \exists t\ x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ m_1\ m_2\ m_3\ a.\ \ \mathsf{m}\ m_1 * \mathsf{m}\ m_2 * \mathsf{m}\ m_3 * \\
\quad\quad \mathsf{r2}\ t * \mathsf{r3}\ x_1 * \mathsf{r4}\ x_2 * \mathsf{r5}\ x_3 * \mathsf{r6}\ x_4 * \mathsf{r7}\ x_5 * \mathsf{r8}\ x_6 * \mathsf{r10}\ a * \\
\quad\quad \langle \mathsf{lisp\_inv}\ (v_1, v_2, v_3, v_4, v_5, v_6, l)\ (x_1, x_2, x_3, x_4, x_5, x_6, a, m_1, m_2, m_3) \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathsf{lisp'}\ (v_1, v_2, v_3, v_4, v_5, v_6, l)\ = \\
\quad \exists x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ m_1\ m_2\ m_3\ a.\ \ \mathsf{m}\ m_1 * \mathsf{m}\ m_2 * \mathsf{m}\ m_3 * \\
\quad\quad \mathsf{eax}\ x_1 * \mathsf{ecx}\ x_2 * \mathsf{edx}\ x_3 * \mathsf{ebx}\ x_4 * \mathsf{esi}\ x_5 * \mathsf{edi}\ x_6 * \mathsf{ebp}\ a * \\
\quad\quad \langle \mathsf{lisp\_inv}\ (v_1, v_2, v_3, v_4, v_5, v_6, l)\ (x_1, x_2, x_3, x_4, x_5, x_6, a, m_1, m_2, m_3) \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathsf{lisp''}\ (v_1, v_2, v_3, v_4, v_5, v_6, l)\ = \\
\quad \exists t\ x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ m_1\ m_2\ m_3\ a.\ \ \mathsf{m}\ m_1 * \mathsf{m}\ m_2 * \mathsf{m}\ m_3 * \\
\quad\quad \mathsf{r2}\ t * \mathsf{r3}\ x_1 * \mathsf{r4}\ x_2 * \mathsf{r5}\ x_3 * \mathsf{r6}\ x_4 * \mathsf{r7}\ x_5 * \mathsf{r8}\ x_6 * \mathsf{r10}\ a * \\
\quad\quad \langle \mathsf{lisp\_inv}\ (v_1, v_2, v_3, v_4, v_5, v_6, l)\ (x_1, x_2, x_3, x_4, x_5, x_6, a, m_1, m_2, m_3) \rangle
\end{array}
$$

The following examples will use only lisp defined for ARM.

[1] Number six was chosen since six is sufficient and suits the x86 implementation best.

```
ALIGNED a = (a && 3w = 0w)

string_mem "" (a,m,dm) = T
string_mem (STRING c s) (a,m,df) = a ∈ dm ∧
                                   (m a = n2w (ORD c)) ∧ string_mem s (a+1w,m,dm)

symbol_table [] x (a,dm,m,dg,g) = (m a = 0w) ∧ a ∈ dm ∧ (x = {})
symbol_table (s::xs) x (a,dm,m,dg,g) = (s ≠ "") ∧ ¬ MEM s xs ∧
  (m a = n2w (string_size s)) ∧ {a; a+4w} ⊆ dm ∧ ((a,s) ∈ x) ∧
    let a' = a + n2w (8 + (string_size s + 3) DIV 4 * 4) in
      a < a' ∧ (m (a+4w) = a') ∧ string_mem s (a+8w,g,dg) ∧
      symbol_table xs (x - {(a,s)}) (a',dm,m,dg,g)

builtin =
  ["nil"; "t"; "quote"; "+"; "-"; "*"; "div"; "mod"; "<"; "car"; "cdr";
   "cons"; "equal"; "cond"; "atomp"; "consp"; "numberp"; "symbolp"; "lambda"]

lisp_symbol_table sym (a,dm,m,dg,g) =
  ∃syms. symbol_table (builtin ++ syms) { (b,s) | (b-a,s) ∈ sym } (a,dm,m,dg,g)

lisp_x (Num k) (a,dm,m) sym = (a = n2w (k * 4 + 2)) ∧ k < 2 ** 30
lisp_x (Sym s) (a,dm,m) sym = ALIGNED (a - 3w) ∧ (a - 3w,s) ∈ sym
lisp_x (Dot x y) (a,dm,m) sym = lisp_x x (m a,dm,m) sym ∧ a ∈ dm ∧ ALIGNED a ∧
                                lisp_x y (m (a+4w),dm,m) sym

ref_set a f = {a + 4w * n2w i | i < 2 * f + 4} ∪ {a - 4w * n2w i | i ≤ 8}
ch_active_set (a,i,e) = { a + 8w * n2w j | i ≤ j ∧ j < e }
ok_data w d = if ALIGNED w then w ∈ d else ¬(ALIGNED (w - 1w))

lisp_inv (t1,t2,t3,t4,t5,t6,l) (w1,w2,w3,w4,w5,w6,a,(dm,m),sym,(dh,h),(dg,g)) =
  ∃i u.
    let v = if u then 1 + l else 1 in
    let d = ch_active_set (a,v,i) in
      32 ≤ w2n a ∧ w2n a + 2 * 8 * l + 20 < 2 ** 32 ∧ l ≠ 0 ∧
      (m a = a + n2w (8 * i)) ∧ ALIGNED a ∧ v ≤ i ∧ i ≤ v + l ∧
      (m (a + 4w) = a + n2w (8 * (v + l))) ∧
      (m (a - 28w) = if u then 0w else 1w) ∧
      (m (a - 32w) = n2w (8 * l)) ∧ (dm = ref_set a (l + l + 1)) ∧
      lisp_symbol_table sym (a + 16w * n2w l + 24w,dh,h,dg,g) ∧
      lisp_x t1 (w1,d,m) sym ∧ lisp_x t2 (w2,d,m) sym ∧ lisp_x t3 (w3,d,m) sym ∧
      lisp_x t4 (w4,d,m) sym ∧ lisp_x t5 (w5,d,m) sym ∧ lisp_x t6 (w6,d,m) sym ∧
      ∀w. w ∈ d ⇒ ok_data (m w) d ∧ ok_data (m (w + 4w)) d
```

Figure 7.1: The definition of the main invariant describing the LISP state.

The following lemma is used in the proof of the theorem about car LISP primitive, described in Section 7.3.1. This lemma can be read as saying that, if lisp_inv relates x_1 to Dot-pair v_1, then x_1 is a word-aligned address into memory segment m, and an assignment of car v_1 to v_2 corresponds to replacing x_2 with the value of memory m at address x_1, i.e. $m(x_1)$.

$$
\begin{array}{l}
(\exists x\ y.\ \mathsf{Dot}\ x\ y = v_1) \wedge \\
\mathsf{lisp\_inv}\ (v_1, v_2, v_3, v_4, v_5, v_6, l)\ (x_1, x_2, x_3, x_4, x_5, x_6, a, m, m_2, m_3)\ \Rightarrow \\
(x_1\,\&\,3 = 0) \wedge x_1 \in \mathsf{domain}\ m \wedge \\
\mathsf{lisp\_inv}\ (v_1, \mathsf{car}\ v_1, v_3, v_4, v_5, v_6, l)\ (x_1, m(x_1), x_3, x_4, x_5, x_6, a, m, m_2, m_3)
\end{array}
$$

The following Hoare triple describes the ARM instruction that is to be verified.

$$
\begin{array}{l}
\{\mathsf{r3}\ r_3 * \mathsf{r4}\ r_4 * \mathsf{m}\ m * \mathsf{pc}\ p * \langle (r_3\,\&\,3 = 0) \wedge r_3 \in \mathsf{domain}\ m \rangle\ \} \\
\ p : \texttt{E5934000 [ldr r4,[r3]]} \\
\{\mathsf{r3}\ r_3 * \mathsf{r4}\ m(r_3) * \mathsf{m}\ m * \mathsf{pc}\ (p{+}4)\ \}
\end{array}
$$

Application of the frame rule, from Section 3.5, produces:

$$
\begin{array}{l}
\{\mathsf{r3}\ r_3 * \mathsf{r4}\ r_4 * \mathsf{m}\ m * \mathsf{pc}\ p * \langle (r_3\,\&\,3 = 0) \wedge r_3 \in \mathsf{domain}\ m \rangle\ * \\
\ \mathsf{r5}\ x_3 * \mathsf{r6}\ x_4 * \mathsf{r7}\ x_5 * \mathsf{r8}\ x_6 * \mathsf{r10}\ a * \mathsf{m}\ m_2 * \mathsf{m}\ m_3\ * \\
\ \langle \mathsf{lisp\_inv}\ (v_1, v_2, v_3, v_4, v_5, v_6, l)\ (r_3, r_4, x_3, x_4, x_5, x_6, a, m, m_2, m_3) \rangle\} \\
\ p : \texttt{E5934000 [ldr r4,[r3]]} \\
\{\mathsf{r3}\ r_3 * \mathsf{r4}\ m(r_3) * \mathsf{m}\ m * \mathsf{pc}\ (p{+}4) * \langle (r_3\,\&\,3 = 0) \wedge r_3 \in \mathsf{domain}\ m \rangle\ * \\
\ \mathsf{r5}\ x_3 * \mathsf{r6}\ x_4 * \mathsf{r7}\ x_5 * \mathsf{r8}\ x_6 * \mathsf{r10}\ a * \mathsf{m}\ m_2 * \mathsf{m}\ m_3\ * \\
\ \langle \mathsf{lisp\_inv}\ (v_1, v_2, v_3, v_4, v_5, v_6, l)\ (r_3, r_4, x_3, x_4, x_5, x_6, a, m, m_2, m_3) \rangle\}
\end{array}
$$

Now the postcondition can be weakened to the desired expression:

$$
\begin{array}{l}
\{\mathsf{r3}\ r_3 * \mathsf{r4}\ r_4 * \mathsf{m}\ m * \mathsf{pc}\ p * \langle (r_3\,\&\,3 = 0) \wedge r_3 \in \mathsf{domain}\ m \rangle\ * \\
\ \mathsf{r5}\ x_3 * \mathsf{r6}\ x_4 * \mathsf{r7}\ x_5 * \mathsf{r8}\ x_6 * \mathsf{r10}\ a * \mathsf{m}\ m_2 * \mathsf{m}\ m_3\ * \\
\ \langle \mathsf{lisp\_inv}\ (v_1, v_2, v_3, v_4, v_5, v_6, l)\ (r_3, r_4, x_3, x_4, x_5, x_6, a, m, m_2, m_3) \rangle\} \\
\ p : \texttt{E5934000 [ldr r4,[r3]]} \\
\{\ \mathsf{lisp}\ (v_1, \mathsf{car}\ v_1, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p + 4)\ \}
\end{array}
$$

Since variables r_3, r_4, x_3, x_4, x_5, x_6, m, m_2, m_3 do not appear in the post-condition, they can be existentially quantified in the precondition, which then strengthens as follows:

$$
\begin{array}{l}
\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p * \langle \exists x\ y.\ \mathsf{Dot}\ x\ y = v_1 \rangle\ \} \\
\ p : \texttt{E5934000 [ldr r4,[r3]]} \\
\{\ \mathsf{lisp}\ (v_1, \mathsf{car}\ v_1, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p + 4)\ \}
\end{array}
$$

The specification for car follows by moving the boolean condition, $\langle \exists x ... \rangle$.

Figure 7.2: A proof of the specification for the car primitive.

7.3.2 Memory layout and specification of 'cons' and 'equal'

Two LISP primitives required code longer than one or two machine instructions, namely cons and equal. Memory allocation, i.e. cons, requires an allocation procedure combined with a garbage collector. However, the top-level specification, which is explained next, hides these facts. Let size count the number of Dot-pairs in an expression.

$$\begin{aligned} \mathsf{size}\ (\mathsf{Num}\ w) &= 0 \\ \mathsf{size}\ (\mathsf{Sym}\ s) &= 0 \\ \mathsf{size}\ (\mathsf{Dot}\ x\ y) &= 1 + \mathsf{size}\ x + \mathsf{size}\ y \end{aligned}$$

The specification of cons guarantees that its implementation will always succeed as long as the number of reachable Dot-pairs is less than the capacity of the heap, i.e. less than l. Note that this precondition is sufficient but imprecise, pointer aliasing is ignored.

$$\begin{array}{l} \mathsf{size}\ v_1 + \mathsf{size}\ v_2 + \mathsf{size}\ v_3 + \mathsf{size}\ v_4 + \mathsf{size}\ v_5 + \mathsf{size}\ v_6 < l\ \Rightarrow \\ \{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p\ \} \\ \quad p : \texttt{E50A3018 E50A4014 E50A5010 E50A600C ... E51A8004 E51A7008} \\ \{\ \mathsf{lisp}\ (\mathsf{cons}\ v_1\ v_2, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p + 324)\ \} \end{array}$$

The implementation of cons includes a copying collector which implements Cheney's algorithm [22]. This copying collector requires the heap to be split into two heap halves of equal size; only one of which is used for heap data at any one point in time. When a collection request is issued, all live elements from the currently used heap half are copied over to the currently unused heap half. The proof of cons is outlined in Chapter 5.

The fact that one half of the heap is left empty might seem to be a waste of space. However, the other heap half need not be left completely unused, as the implementation of equal can make use of it. The LISP primitive equal tests whether two s-expressions are equal by traversing the expression tree as a normal recursive procedure. This recursive traversal requires a stack, but the stack can in this case be built inside the unused heap half as the garbage collector will not be called during the execution of equal. Thus, the implementation of equal uses no external stack and requires no conditions on the size of the expressions v_1 and v_2, as their depths cannot exceed the length of a heap half.

$$\begin{array}{l} \{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p\ \} \\ \quad p : \texttt{E1530004 03A0300F 0A000025 E50A4014 ... E51A7008 E51A8004} \\ \{\ \mathsf{lisp}\ (\mathsf{equal}\ v_1\ v_2, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p + 164)\ \} \end{array}$$

7.4 Compiling s-expression functions to machine code

The previous sections described the theorems which state that certain machine instructions execute LISP primitives. These theorems can be used to augment the input-language understood by the proof-producing compiler in Chapter 6. The theorems mentioned above allow

the compiler to accept:

$$\begin{array}{l}\mathsf{let}\ v_2 = \mathsf{car}\ v_1\ \mathsf{in}\ ...\\ \mathsf{let}\ v_1 = \mathsf{plus}\ v_1\ v_2\ \mathsf{in}\ ...\\ \mathsf{let}\ v_1 = \mathsf{cons}\ v_1\ v_2\ \mathsf{in}\ ...\\ \mathsf{let}\ v_1 = \mathsf{equal}\ v_1\ v_2\ \mathsf{in}\ ...\end{array}$$

Theorems for basic tests have also been proved in a similar manner, and can be provided to the compiler. For example, the following theorem shows that ARM instruction `teq r3,#3`, encoded as `E3330003`, assigns boolean value ($v_1 = \mathsf{Sym}$ `"nil"`) to status bit z.

$$\begin{array}{l}\{\ \mathsf{lisp}\ (v_1,v_2,v_3,v_4,v_5,v_6,l) * \mathsf{pc}\ p * \mathsf{s}\ \}\\ \ p : \texttt{E3330003 [teq r3,\#3]}\\ \{\ \mathsf{lisp}\ (v_1,v_2,v_3,v_4,v_5,v_6,l) * \mathsf{pc}\ (p+4) * \mathsf{sz}\ (v_1 = \mathsf{Sym}\ \texttt{"nil"}) *\\ \ \ \exists n\ c\ v.\ \mathsf{sn}\ n * \mathsf{sc}\ c * \mathsf{sv}\ v\ \}\end{array}$$

The compiler can use such theorems to create branches on the expression assigned to status bits. The above theorem adds support for the if-statement:

$$\mathsf{if}\ \ v_1 = \mathsf{Sym}\ \texttt{"nil"}\ \ \mathsf{then}\ ...\ \mathsf{else}\ ...$$

Once the compiler was given sufficient Hoare-triple theorems it could be used to compile functions operating over s-expressions into machine code. An example will illustrate the process. From the following function

$$\begin{array}{lcl}\mathsf{sumlist}(v_1,v_2,v_3) & = & \mathsf{if}\ v_1 = \mathsf{Sym}\ \texttt{"nil"}\ \mathsf{then}\ (v_1,v_2,v_3)\ \mathsf{else}\\ & & \ \ \mathsf{let}\ v_3 = \mathsf{car}\ v_1\ \mathsf{in}\\ & & \ \ \mathsf{let}\ v_1 = \mathsf{cdr}\ v_1\ \mathsf{in}\\ & & \ \ \mathsf{let}\ v_2 = \mathsf{plus}\ v_2\ v_3\ \mathsf{in}\\ & & \ \ \ \ \mathsf{sumlist}(v_1,v_2,v_3)\end{array}$$

the compiler produces the theorem below, also see Figure 7.3, containing the generated ARM machine code and a precondition $\mathsf{sumlist\_pre}(v_1,v_2,v_3)$.

$$\begin{array}{l}\mathsf{sumlist\_pre}(v_1,v_2,v_3)\ \Rightarrow\\ \{\ \mathsf{lisp}\ (v_1,v_2,v_3,v_4,v_5,v_6,l) * \mathsf{pc}\ p * \mathsf{s}\ \}\\ \ p : \texttt{E3330003 0A000004 E5935000 E5934004 E0844005 E2444002 EAFFFFF8}\\ \{\ \mathsf{let}\ (v_1,v_2,v_3) = \mathsf{sumlist}(v_1,v_2,v_3)\ \mathsf{in}\\ \ \ \mathsf{lisp}\ (v_1,v_2,v_3,v_4,v_5,v_6,l) * \mathsf{pc}\ (p+28) * \mathsf{s}\ \}\end{array}$$

The automatically generated pre-functions collect side conditions:

$$\begin{array}{l}\mathsf{sumlist\_pre}(v_1,v_2,v_3) =\\ \ \ \mathsf{if}\ v_1 = \mathsf{Sym}\ \texttt{"nil"}\ \mathsf{then}\ \mathsf{T}\ \mathsf{else}\\ \ \ \ \ \mathsf{let}\ \mathit{cond} = (\exists x\ y.\ \mathsf{Dot}\ x\ y = v_1)\ \mathsf{in}\\ \ \ \ \ \mathsf{let}\ v_3 = \mathsf{car}\ v_1\ \mathsf{in}\\ \ \ \ \ \mathsf{let}\ \mathit{cond} = \mathit{cond} \wedge (\exists x\ y.\ \mathsf{Dot}\ x\ y = v_1)\ \mathsf{in}\\ \ \ \ \ \mathsf{let}\ v_1 = \mathsf{cdr}\ v_1\ \mathsf{in}\\ \ \ \ \ \mathsf{let}\ \mathit{cond} = \mathit{cond} \wedge (\exists m\ n.\ \mathsf{Num}\ m = v_2 \wedge \mathsf{Num}\ n = v_3 \wedge m{+}n < 2^{30})\ \mathsf{in}\\ \ \ \ \ \mathsf{let}\ v_2 = \mathsf{plus}\ v_2\ v_3\ \mathsf{in}\\ \ \ \ \ \ \ \mathsf{sumlist\_pre}(v_1,v_2,v_3) \wedge \mathit{cond}\end{array}$$

The compiler starts its proof from the following theorems describing the test $v_1 = \mathsf{Sym}\ \texttt{"nil"}$ as well as operations car, cdr and plus.

1. $\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p * \mathsf{s}\ \}$
   p : `E3330003 [ teq r3,#3 ]`
   $\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p+4) * \mathsf{sz}\ (v_1 = \mathsf{Sym}\ \texttt{"nil"}) *$
   $\exists n\ c\ v.\ \mathsf{sn}\ n * \mathsf{sc}\ c * \mathsf{sv}\ v\ \}$
2. $(\exists x\ y.\ \mathsf{Dot}\ x\ y = v_1) \Rightarrow$
   $\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p * \mathsf{s}\ \}$
   p : `E5935000 [ ldr r5,[r3] ]`
   $\{\ \mathsf{lisp}\ (v_1, v_2, \mathsf{car}\ v_1, v_4, v_5, v_6, l) * \mathsf{pc}\ (p+4)\ \}$
3. $(\exists x\ y.\ \mathsf{Dot}\ x\ y = v_1) \Rightarrow$
   $\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p * \mathsf{s}\ \}$
   p : `E5933004 [ ldr r3,[r3,#4] ]`
   $\{\ \mathsf{lisp}\ (\mathsf{cdr}\ v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p+4)\ \}$
4. $(\exists m\ n.\ \mathsf{Num}\ m = v_2 \wedge \mathsf{Num}\ n = v_3 \wedge m{+}n < 2^{30}) \Rightarrow$
   $\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p * \mathsf{s}\ \}$
   p : `E0844005 E2444002 [ add r4,r4,r5; sub r4,r4,#4 ]`
   $\{\ \mathsf{lisp}\ (v_1, \mathsf{plus}\ v_2\ v_3, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p+4)\ \}$

The compiler next generates two branches to glue the code together; the branch instructions have the following specifications:

5. $\{\ \mathsf{pc}\ p * \mathsf{sz}\ z * \langle z \rangle\ \}\ p$: `0A000004 [ beq L2 ]` $\{\ \mathsf{pc}\ (p+24) * \mathsf{sz}\ z\ \}$
6. $\{\ \mathsf{pc}\ p * \mathsf{sz}\ z * \langle \neg z \rangle\ \}\ p$: `0A000004 [ beq L2 ]` $\{\ \mathsf{pc}\ (p+4) * \mathsf{sz}\ z\ \}$
7. $\{\ \mathsf{pc}\ p\ \}\ p$: `EAFFFFF8 [ b L1 ]` $\{\ \mathsf{pc}\ (p-24)\ \}$

The theorems above are collapsed into a single one-pass theorem:

8. (if $v_1 = \mathsf{Sym}\ \texttt{"nil"}$ then
   $(new@p = p+28) \wedge (new@v_1 = v_1) \wedge (new@v_2 = v_2) \wedge (new@v_3 = v_3)$
   else
   let $cond = (\exists x\ y.\ \mathsf{Dot}\ x\ y = v_1)$ in
   let $v_3 = \mathsf{car}\ v_1$ in
   let $cond = cond \wedge (\exists x\ y.\ \mathsf{Dot}\ x\ y = v_1)$ in
   let $v_1 = \mathsf{cdr}\ v_1$ in
   let $cond = cond \wedge (\exists m\ n.\ \mathsf{Num}\ m = v_2 \wedge \mathsf{Num}\ n = v_3 \wedge m{+}n < 2^{30})$ in
   let $v_2 = \mathsf{plus}\ v_2\ v_3$ in
   $(new@p = p) \wedge (new@v_1 = v_1) \wedge (new@v_2 = v_2) \wedge (new@v_3 = v_3) \wedge cond) \Rightarrow$
   $\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p * \mathsf{s}\ \}$
   p : `E3330003 0A000004 E5935000 E5934004 E0844005 E2444002 EAFFFFF8`
   $\{\ \mathsf{lisp}\ (new@v_1, new@v_2, new@v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ new@p * \mathsf{s}\ \}$

This one-pass theorem is then used to instantiate the loop rule from Section 4.3.5.

Figure 7.3: Highlights of the proof performed inside the compiler.

7.5 Assembling the LISP evaluator

LISP evaluation was defined as a large tail-recursive function lisp_eval, listed in Appendix B, and then compiled, to ARM, PowerPC and x86, to produce theorems of the following form. The theorem below states that the generated ARM code executes lisp_eval for inputs that do not violate any of the side conditions gathered in lisp_eval_pre.

$$\begin{array}{l}\mathsf{lisp\_eval\_pre}(v_1, v_2, v_3, v_4, v_5, v_6, l) \Rightarrow \\ \{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p * \mathsf{s}\ \} \\ \ p : \ldots \text{ code not shown } \ldots \\ \{\ \mathsf{lisp}\ (\mathsf{lisp\_eval}(v_1, v_2, v_3, v_4, v_5, v_6, l)) * \mathsf{pc}\ (p + 7780) * \mathsf{s}\ \}\end{array}$$

The function lisp_eval evaluates the expression stored in v_1, input v_6 is a list of symbol-value pairs against which symbols in v_1 are evaluated, inputs v_2, v_3, v_4 and v_5 are used as temporaries that are to be initialised with Sym `"nil"`. The heap limit l had to be passed into lisp_eval due to an implementation restriction which requires lisp_eval_pre to input the same variables as lisp_eval. The side condition lisp_eval_pre uses l to state restrictions on applications of cons.

7.6 Evaluator implements McCarthy's LISP 1.5

The previous sections described how a function lisp_eval was compiled down to machine code. The compiler generated some code and derived a theorem which states that the generated code correctly implements lisp_eval. However, the compiler does not (and cannot) give any evidence that lisp_eval in fact implements 'LISP evaluation'. The definition of lisp_eval is long and full of tedious details of how the intermediate stack is maintained and used, and thus it is far from obvious that lisp_eval corresponds to 'LISP evaluation'.

In order to gain confidence that the generated machine code actually implements LISP evaluation, we proved that lisp_eval implements a clean relational semantics of LISP 1.5 [45], listed in Appendix A. We proved that whenever the relation for LISP 1.5 evaluation $\mathsf{R_{ev}}$ relates expression s under environment ρ to expression r, then lisp_eval will do the same. Here t, t′ and u are translation functions, from one form of s-expressions to another. Let nil = Sym `"nil"` and fst $(x, y, \ldots) = x$.

$$\forall s\ \rho\ r.\ \ \mathsf{R_{ev}}(s, \rho)\ r\ \Rightarrow\ \mathsf{fst}\ (\mathsf{lisp\_eval}\ (\mathsf{t}\ s, \mathsf{nil}, \mathsf{nil}, \mathsf{nil}, \mathsf{u}\ \rho, \mathsf{nil}, l)) = \mathsf{t}'\ r$$

7.7 Verified parser and printer

Sections 7.4 and 7.5 explained how machine code was generated and proved to implement a function called lisp_eval. The precondition of the certificate theorem requires the initial state to satisfy a complex heap invariant lisp. How do we know that this precondition

is not accidentally equivalent to false, making the theorem vacuously true? To remedy this shortcoming, we have verified machine code that will set-up an appropriate state from scratch.

The set-up and tear-down code includes a parser and printer that will, respectively, read in an input s-expression and print out the resulting s-expression. The development of the parser and printer started by first defining a function sexp2string which lays down how s-expressions are to be represented in string form, defined in Figure 7.4. Then a function string2sexp was defined, in Figure 7.5, which satisfies:

$$\forall s.\ \ \mathsf{sexp\_ok}\ s\ \Rightarrow\ \mathsf{string2sexp}\ (\mathsf{sexp2string}\ s) = s$$

Here sexp_ok s makes sure that s does not contain symbols that print ambiguously, e.g. Sym "", Sym "(" and Sym "2".

Machine code was written and verified based on the high-level functions string2sexp and sexp2string. Writing these high-level definitions first was a great help when constructing the machine code (using the compiler from Chapter 6).

The overall theorems about our LISP implementations are of the following form. If $\mathsf{R_{ev}}$ relates s with r under the empty environment (i.e. $\mathsf{R_{ev}}(s, [])\ r$), no illegal symbols are used (i.e. sexp_ok (t s)), running lisp_eval on t s will not run out of heap space (i.e. lisp_eval_pre(t s, nil, nil, nil, nil, nil, l)), the string representation of t s is in memory (i.e. string a (sexp2string (t s))), and there is enough space to parse t s and set up a heap of size l (i.e. enough_space (t s) l), then the code will execute successfully and terminate with the string representation of t′ r stored in memory (i.e. string a (sexp2string (t′ r))). The ARM code expects the address of the input string to be in register 3, i.e. r3 a.

$$\begin{array}{l}
\forall s\ r\ l\ p.\\
\quad \mathsf{R_{ev}}(s, [])\ r \wedge \mathsf{sexp_ok}\ (\mathsf{t}\ s) \wedge \mathsf{lisp_eval_pre}(\mathsf{t}\ s, \mathsf{nil}, \mathsf{nil}, \mathsf{nil}, \mathsf{nil}, \mathsf{nil}, l) \Rightarrow\\
\quad \{\ \exists a.\ \mathsf{r3}\ a * \mathsf{string}\ a\ (\mathsf{sexp2string}\ (\mathsf{t}\ s)) * \mathsf{enough_space}\ (\mathsf{t}\ s)\ l * \mathsf{pc}\ p\ \}\\
\quad p : \ldots \text{ code not shown } \ldots\\
\quad \{\ \exists a.\ \mathsf{r3}\ a * \mathsf{string}\ a\ (\mathsf{sexp2string}\ (\mathsf{t}'\ r)) * \mathsf{enough_space}'\ (\mathsf{t}\ s)\ l * \mathsf{pc}\ (p{+}8968)\ \}
\end{array}$$

The input needs to be in register 3 for PowerPC and the eax register for x86.

7.8 Quantitative data

This project has produced in total some 4,580 lines of proof automation and 16,130 lines of interactive proofs and definitions, excluding the definitions of the instruction set models, mentioned in Section 3.2.1. Running through all of the proofs takes approximately 2.5 hours in HOL4 using PolyML. The ARM implementation is 2,601 instructions long (10,404 bytes), x86 is 3,135 instructions (9,054 bytes) and the PowerPC implementation consists of 2,929 instructions (11,716 bytes).

A few experiments suggest that the verified LISP implementations seem to have reasonable execution times: the `pascal-triangle` example, from Section 7.1, executes on a 2.4 GHz x86

$$
\begin{array}{rcl}
\mathsf{sexp2string}\ x & = & \mathsf{aux}\ (x, \mathsf{T}) \\
\mathsf{aux}\ (\mathsf{Num}\ n, b) & = & \mathsf{num2str}\ n \\
\mathsf{aux}\ (\mathsf{Sym}\ s, b) & = & s \\
\mathsf{aux}\ (\mathsf{Dot}\ x\ y, b) & = & \mathsf{if}\ \mathsf{isQuote}\ (\mathsf{Dot}\ x\ y) \wedge b\ \mathsf{then}\ \texttt{"'"} +\!\!+ \mathsf{aux}\ (\mathsf{car}\ y, \mathsf{T})\ \mathsf{else} \\
 & & \quad \mathsf{let}\ (a, e) = (\mathsf{if}\ b\ \mathsf{then}\ (\texttt{"("}, \texttt{")"})\ \mathsf{else}\ (\texttt{""}, \texttt{""}))\ \mathsf{in} \\
 & & \quad\quad \mathsf{if}\ y = \mathsf{Sym}\ \texttt{"nil"}\ \mathsf{then}\ a +\!\!+ \mathsf{aux}\ (x, \mathsf{T}) +\!\!+ e\ \mathsf{else} \\
 & & \quad\quad \mathsf{if}\ \mathsf{isDot}\ y\ \mathsf{then}\ a +\!\!+ \mathsf{aux}\ (x, \mathsf{T}) +\!\!+ \texttt{" "} +\!\!+ \mathsf{aux}\ (y, \mathsf{F}) +\!\!+ e \\
 & & \quad\quad\quad\quad \mathsf{else}\ a +\!\!+ \mathsf{aux}\ (x, \mathsf{T}) +\!\!+ \texttt{" . "} +\!\!+ \mathsf{aux}\ (y, \mathsf{F}) +\!\!+ e \\
\mathsf{isDot}\ x & = & \exists y\ z.\ x = \mathsf{Dot}\ y\ z \\
\mathsf{isQuote}\ x & = & \exists y.\ x = \mathsf{Dot}\ (\mathsf{Sym}\ \texttt{"quote"})\ (\mathsf{Dot}\ y\ (\mathsf{Sym}\ \texttt{"nil"}))
\end{array}
$$

Figure 7.4: Definition of string representation of s-expressions.

Parsing is defined as the follows. Here reverse is normal list reversal.

$$\mathsf{string2sexp}\ str \quad = \quad \mathsf{car}\ (\mathsf{sexp\_parse}\ (\mathsf{reverse}\ (\mathsf{sexp\_lex}\ str))\ (\mathsf{Sym}\ \texttt{"nil"})\ [])$$

The lexing function sexp_lex splits a string into a list of strings, e.g.

$$\mathsf{sexp\_lex}\ \texttt{"(car ('23 . y))"} \quad = \quad [\texttt{"("}, \texttt{"car"}, \texttt{"("}, \texttt{"'"}, \texttt{"23"}, \texttt{"."}, \texttt{"y"}, \texttt{")"}, \texttt{")"}]$$

Token parsing is defined as:

$$
\begin{array}{rcl}
\mathsf{sexp\_parse}\ []\ exp\ stack & = & exp \\
\mathsf{sexp\_parse}\ (\texttt{")"} :: ts)\ exp\ stack & = & \mathsf{sexp\_parse}\ ts\ (\mathsf{Sym}\ \texttt{"nil"})\ (exp :: stack) \\
\mathsf{sexp\_parse}\ (\texttt{"("} :: ts)\ exp\ stack & = & \mathsf{sexp\_parse}\ ts\ (\mathsf{Dot}\ exp\ (\mathsf{head}\ stack))\ (\mathsf{tail}\ stack) \\
\mathsf{sexp\_parse}\ (\texttt{"."} :: ts)\ exp\ stack & = & \mathsf{sexp\_parse}\ ts\ (\mathsf{car}\ exp)\ stack \\
\mathsf{sexp\_parse}\ (\texttt{"'"} :: ts)\ exp\ stack & = & \mathsf{sexp\_parse}\ ts\ (\mathsf{Dot}\ (\mathsf{Dot}\ (\mathsf{Sym}\ \texttt{"quote"}) \\
 & & \quad (\mathsf{Dot}\ (\mathsf{car}\ exp)\ (\mathsf{Sym}\ \texttt{"nil"})))\ (\mathsf{cdr}\ exp))\ stack \\
\mathsf{sexp\_parse}\ (t :: ts)\ exp\ stack & = & \mathsf{sexp\_parse}\ ts\ (\mathsf{Dot}\ (\mathsf{if}\ \mathsf{is\_num}\ t\ \mathsf{then} \\
 & & \quad \mathsf{Num}\ (\mathsf{str2num}\ t)\ \mathsf{else}\ \mathsf{Sym}\ t)\ exp)\ stack
\end{array}
$$

Figure 7.5: Definition of a function for parsing s-expressions.

processor in less than 1 millisecond, on a 1.5 GHz PowerPC processor in 3 milliseconds and on a 67 MHz ARM processor in approximately 90 milliseconds. The ARM implementation is 2,601 instructions long (10,404 bytes), x86 is 3,135 instructions (9,054 bytes) and the PowerPC implementation consists of 2,929 instructions (11,716 bytes).

7.9 Discussion of related work

This project has produced trustworthy implementations of LISP. The VLISP project by Guttman et al. [49] shared our goal, but differed in many other aspects. For example, the VLISP project implemented a larger LISP dialect, namely Scheme, and emphasised rigour, not full formality:

> "The verification was intended to be rigorous, but not completely formal, much in the style of ordinary mathematical discourse. Our goal was to verify the algorithms and data types used in the implementation, not their embodiment in the code."

The VLISP project developed an implementation which translates Scheme programs into byte code that is then run on a rigorously verified interpreter. Much like our project, the VLISP project developed their interpreter in a subset of the source language: for them PreScheme, and for us, the input language of our augmented compiler, Section 7.4.

Work that aims to implement functional languages, in a formally verified manner, include Pike et al. [99] on a certifying compiler from Cryptol (a dialect of Haskell) to AAMP7 code; Dargaye and Leroy [31] on a certified compiler from mini-ML to PowerPC assembly; Li and Slind's work [64] on a certifying compiler from a subset of HOL4 to ARM assembly; and also Chlipala's certified compiler [24] from the lambda calculus to an invented assembly language. The above work either assumes that the environment implements run-time memory management correctly [24, 31] or restricts the input language to a degree where no run-time memory management is needed [64, 99]. It seems that none of the above have made use of (the now large number of) verified garbage collectors (e.g. McCreight et al. [76] have been performing correctness proofs for increasingly sophisticated garbage collectors).

The parser and printer proofs, in Section 7.7, involved verifying implementations of string-copy, -length, -compare etc., bearing some resemblance to pioneering work by Boyer and Yu [16] on verification of machine code. They verified Motorola MC68020 code implementing a library of string functions.

Chapter 8

Conclusions

8.1 Summary

This thesis has explored new ways of using mathematical methods to provide realistic assurances that programs do not contain functional faults. Techniques are proposed for proving the correctness of software with respect to realistic models of real machine languages, where restrictions imposed by resource limitations of the underlying hardware must to be taken into account.

A new machine-code Hoare triple was presented for writing concise specifications of functional correctness, termination and resource usage for machine-code programs. This Hoare triple is the first to be applied simultaneously to multiple off-the-shelf models of machine code (ARM, PowerPC and x86).

A new proof-assistant based approach to program verification was presented; programs are translated via fully-automatic deduction into recursive functions defined in the logic of a theorem prover. This approach contrasts with well-established methods based on Hoare logic and verification condition generation (VCG) by removing the need to annotate programs with assertions, making subsequent verification proofs natural to the theorem prover, and being easier to implement than a trusted VCG.

This translation of programs into recursive functions was applied to machine code in order to create a decompiler, which maps ARM, PowerPC and x86 code into the native language of a theorem prover. Unlike established methods for machine code verification, decompilation allows reuse of verification proofs even between different instruction architectures. Use of decompilation was illustrated by verification of an allocator, with a built-in Cheney collector. The verification proof improves on published work by being shorter, directly reusable on several architectures and handling the 'out-of-memory' case properly.

Techniques for construction of correct code were advanced by a new approach to *proof-producing compilation* by which functions of higher-order logic are mapped to machine code. The prototype compiler presented here is the first certifying/certified compiler to target multiple, carefully modelled, commercial machine-languages. Unlike other work published

on compilation from higher-order logic, this compilation technique allows input functions to be partially specified, and also provides broad support for user-defined extensions.

The potential for creating trustworthy software was illustrated by a case study which produced verified interpreters for a small LISP-like language. This is, as far as the author is aware of, the first implementation of a functional programming language that has been mechanically proved correct with respect to a detailed model of a commercial machine language. LISP was chosen as the example, since future work can build on top of its clean layer of abstraction.

8.2 Future research

This thesis presented a decompiler which transforms machine code into recursive functions operating over a low level of abstraction; memory accesses turn into applications of functions from addresses to machine words. The decompilation algorithm is robust since heuristics are used only for discovery of the control flow in the machine code. An interesting avenue of further research would be to incorporate specialised heuristics into decompilation. For example, a program for linked-list reversal, which currently decompiles to:

$$\begin{array}{rcl} \mathsf{f}(ecx, m) & = & \mathsf{g}(0, ecx, m) \\ \mathsf{g}(ebx, ecx, m) & = & \mathsf{if}\ ecx = 0\ \mathsf{then}\ (ebx, ecx, m)\ \mathsf{else}\ \mathsf{g}(ecx, m(ecx), m[ecx \mapsto ebx]) \end{array}$$

might instead decompile into something more abstract:

$$\begin{array}{rcl} \mathsf{reverse}\ xs & = & \mathsf{rev}(xs, []) \\ \mathsf{rev}(xs, ys) & = & \mathsf{if}\ xs = []\ \mathsf{then}\ (xs, ys)\ \mathsf{else}\ \mathsf{rev}(\mathsf{tail}\ xs, (\mathsf{head}\ xs) :: ys) \end{array}$$

if the verifier or an external tool, e.g. [120], informs the decompiler that ecx initially points to a linked-list. The correctness of reverse can already be expressed rather neatly given an appropriate definition of list, e.g. from classical separation logic [105].

$$\begin{array}{l} \{\ \exists ecx.\ \mathsf{ebx}\ \_ * \mathsf{ecx}\ ecx * \mathsf{edx}\ \_ * \mathsf{list}\ (ecx, xs) * \mathsf{eip}\ p * \mathsf{s}\ \} \\ p : \texttt{BB00000000 83F900 740C 8B11 8919 89CB 89D1 85D2 75F4} \\ \{\ \exists ecx.\ \mathsf{ebx}\ \_ * \mathsf{ecx}\ ecx * \mathsf{edx}\ \_ * \mathsf{list}\ (ecx, \mathsf{reverse}\ xs) * \mathsf{eip}\ (p + 22) * \mathsf{s}\ \} \end{array}$$

The LISP case study might spawn a number of exciting future projects. For example, it might be possible to produce a verified ACL2 evaluator if the implementation of lisp_eval (or an altered version of lisp_eval) can be shown to implement ACL2's notion of evaluation. The LISP case study could also provide safe run-time environments, perhaps extended with fast cryptography functions implemented directly as primitives in machine code.

Other more distant future projects could look into adapting decompilation to deal with concurrency. The decompiler currently produces a function describing the state change. Modifying it to deliver relations meaningful and useful for proofs is a challenge. Ideas based on mixtures of rely/guarantee and separation logic might provide some of the answers [116]. However, reasoning about realistically modelled concurrent machine code may have to wait,

since there seems to be confusion over what the semantics of concurrent machine code is or ought to be [109].

Another exciting avenue of future work is to consider the presented decompiler as an interface tool. Other tools can use decompilation as a way to map verification proofs, in a trusted manner, all the way down to realistically modelled machine code.

Appendix A

Definition of LISP evaluation

Chapter 7 makes use of a relational semantics of the pure core of LISP 1.5. This relational semantics was developed by Gordon [45]. Its HOL4 definition is presented below.

Syntax. Gordon defines the syntax of terms and functions separately.

```
atom = Nil | Number of num | String of string

sexpression = A of atom | Cons of sexpression => sexpression

term = Con of sexpression
     | Var of string
     | App of func => term list
     | Ite of (term # term) list

func = FunCon of string
     | FunVar of string
     | Lambda of string list => term
     | Label  of string => func
```

Utility values and functions are defined:

```
False = A Nil
True = A(String "t")

isTrue s = (s ≠ False) ∧ (s ≠ A (String "nil"))

Car(Cons s1 s2) = s1
Cdr(Cons s1 s2) = s2

delete_Nil_aux Nil = String "nil"
delete_Nil_aux (Number n) = Number n
delete_Nil_aux (String s) = String s
delete_Nil (A a) = A (delete_Nil_aux a)
delete_Nil (Cons s t) = Cons (delete_Nil s) (delete_Nil t)
```

```
Equal (x,y) = if delete_Nil x = delete_Nil y then True else False

Atomp (A a) = True
Atomp _ = False

Consp (A a) = False
Consp _ = True

Numberp (A (Number n)) = True
Numberp _ = False

Symbolp (A (String s)) = True
Symbolp (A Nil) = True
Symbolp _ = False

Add  ((A(Number m)),(A(Number n))) = A(Number(m + n))
Sub  ((A(Number m)),(A(Number n))) = A(Number(m - n))
Mult ((A(Number m)),(A(Number n))) = A(Number(m * n))
Div  ((A(Number m)),(A(Number n))) = A(Number(m DIV n))
Mod  ((A(Number m)),(A(Number n))) = A(Number(m MOD n))
Less ((A(Number m)),(A(Number n))) = if m < n then True else False

FunConSem s sl =
 if s = "car"     then Car(EL 0 sl)                else
 if s = "cdr"     then Cdr(EL 0 sl)                else
 if s = "cons"    then Cons(EL 0 sl) (EL 1 sl)     else
 if s = "+"       then FOLDL Add (A(Number 0)) sl  else
 if s = "*"       then FOLDL Mult (A(Number 1)) sl else
 if s = "-"       then Sub(EL 0 sl,EL 1 sl)        else
 if s = "div"     then Div(EL 0 sl,EL 1 sl)        else
 if s = "mod"     then Mod(EL 0 sl,EL 1 sl)        else
 if s = "<"       then Less(EL 0 sl,EL 1 sl)       else
 if s = "equal"   then Equal(EL 0 sl,EL 1 sl)      else
 if s = "atomp"   then Atomp(EL 0 sl)              else
 if s = "consp"   then Consp(EL 0 sl)              else
 if s = "numberp" then Numberp(EL 0 sl)            else
 if s = "symbolp" then Symbolp(EL 0 sl)            else
 ARB

FunConSemOK s sl =
 if s = "car"     then ∃u v. sl = [Cons u v]   else
 if s = "cdr"     then ∃u v. sl = [Cons u v]   else
 if s = "cons"    then ∃u v. sl = [u; v]       else
 if s = "+"       then (∀x. MEM x sl ⇒ ∃n. x = A (Number n)) else
 if s = "-"       then ∃m n. sl = [A (Number m); A (Number n)] else
 if s = "*"       then (∀x. MEM x sl ⇒ ∃n. x = A (Number n)) else
 if s = "div"     then ∃m n. sl = [A (Number m); A (Number n)] else
 if s = "mod"     then ∃m n. sl = [A (Number m); A (Number n)] else
 if s = "<"       then ∃m n. sl = [A (Number m); A (Number n)] else
 if s = "equal"   then ∃u v. sl = [u; v]       else
 if s = "atomp"   then ∃u.   sl = [u]          else
 if s = "consp"   then ∃u.   sl = [u]          else
 if s = "numberp" then ∃u.   sl = [u]          else
 if s = "symbolp" then ∃u.   sl = [u]          else
 F
```

An environment (called `alist`) is a finite function from names (strings) to values of type `:sexpression + func`, i.e. variables and `Label`-defined functions share the name-space.

```
VarBind a [] sl = (a : (string |-> sexpression + func))
VarBind a (x::xl) [] = (VarBind (a |+ (x, INL(A Nil))) xl [])
VarBind a (x::xl) (s::sl) = (VarBind (a |+ (x, INL s)) xl sl)

FunBind (a:string|->sexpression+func) f fn = a |+ (f, INR fn)
```

Semantics. The operational semantics is defined using three inductive relations,

```
R_ap (fn,args,a) s  - fn applied to args evaluates to s with alist a
R_ev (e,a) s        - term e evaluates to s-expression s with alist a
R_evl (el,a) sl     - term list el evaluates to s-expression list sl with alist a
```

as follows:

```
(∀s a.
  R_ev (Con s, a) s)
∧
(∀x a.
  x ∈ FDOM a ∧ ISL (a ' x)
  ⇒ R_ev (Var x, a) (OUTL(a ' x)))
∧
(∀fc args a.
  FunConSemOK fc args
  ⇒ R_ap (FunCon fc,args,a) (FunConSem fc args))
∧
(∀fn el args s a.
  R_evl (el,a) args ∧ R_ap (fn,args,a) s ∧ (LENGTH args = LENGTH el)
  ⇒ R_ev (App fn el,a) s)
∧
(∀a.
  R_ev (Ite [], a) False)
∧
(∀e1 e2 el s a.
  R_ev (e1,a) False ∧ R_ev (Ite el,a) s
  ⇒ R_ev (Ite ((e1,e2)::el),a) s)
∧
(∀e1 e2 el s1 s a.
  R_ev (e1,a) s1 ∧ isTrue s1 ∧ R_ev (e2,a) s
  ⇒ R_ev (Ite ((e1,e2)::el),a) s)
∧
(∀x fn args s a.
  R_ap (fn,args,FunBind a x fn) s
  ⇒ R_ap(Label x fn,args,a) s)
∧
(∀xl e args s a.
  (LENGTH args = LENGTH xl) ∧ R_ev (e,VarBind a xl args) s
  ⇒ R_ap (Lambda xl e,args,a) s)
∧
```

```
(∀fv args s a.
  fv ∉ {"quote";"cond";"car";"cdr";"cons";"+";"-";"*";"div";"mod";"<";
        "equal";"atomp";"consp";"symbolp";"numberp"} ∧
  fv ∈ FDOM a ∧ ISR (a ' fv) ∧ R_ap (OUTR(a ' fv),args,a) s
  ⇒ R_ap (FunVar fv,args,a) s)
∧
(∀a.
  R_evl ([],a) [])
∧
(∀e el s sl a.
  R_ev (e,a) s ∧ R_evl (el,a) sl
  ⇒ R_evl (e::el,a) (s::sl))
```

Translation. Chapter 7 defines s-expressions differently from the above definition. The following functions translate into the representation used in Chapter 7.

```
atom2sexp Nil = Sym "nil"
atom2sexp (Number n) = Num n
atom2sexp (String s) = Sym s

sexpression2sexp (A a) = atom2sexp a
sexpression2sexp (Cons x y) = Dot (sexpression2sexp x) (sexpression2sexp y)

list2sexp [] = Sym "nil"
list2sexp (x::xs) = Dot x (list2sexp xs)

func2sexp (FunCon s) = Sym s
func2sexp (FunVar s) = Sym s
func2sexp (Lambda vs x) = list2sexp [Sym "lambda"; list2sexp (MAP Sym vs); term2sexp x]
func2sexp (Label l f) = list2sexp [Sym "label"; Sym l; func2sexp f]

term2sexp (Con y) = list2sexp [Sym "quote"; sexpression2sexp y]
term2sexp (Var v) = Sym v
term2sexp (App f xs) = list2sexp (func2sexp f::MAP (λx. term2sexp x) xs)
term2sexp (Ite cs) = list2sexp (Sym "cond"::
               MAP (λ(t1,t2). list2sexp [term2sexp t1; term2sexp t2]) cs))

fmap2list f = MAP (λx. (x,f ' x)) (SET_TO_LIST (FDOM f))

pair2sexp (s,INL x) = Dot (Sym s) (sexpression2sexp x)
pair2sexp (s,INR y) = Dot (Sym s) (func2sexp y)

alist2sexp al = list2sexp (MAP pair2sexp al)
```

Theorem. The author of this dissertation has proved the following HOL4 theorem relating the LISP semantics from above, which was defined by Gordon [45], to the function `lisp_eval` defined in the next appendix. Here `NIL = Sym "nil"`.

```
⊢ ∀exp alist result l.
    R_ev (exp,alist) result ⇒
      (FST (lisp_eval (term2sexp exp,NIL,NIL,NIL,NIL,alist2sexp (fmap2list alist),l)) =
       sexpression2sexp result)
```

Appendix B

Definition of LISP evaluation as a tail-recursive function

The HOL4 definition of lisp_eval is listed below. Comments are written: `(* ... *)`

Chapter 7 uses variables v_1, v_2, v_3, v_4, v_5, v_6 and l, which are here referred to by names `exp`, `x`, `y`, `z`, `stack`, `alist` and `l`, respectively.

```
(lookup_aux (exp,x,y,z,stack,alist,l) =
   let x = CAR y in
   let x = CAR x in
     if exp = x then
       let x = CAR y in
       let exp = CDR x in
         (exp,x,y,z,stack,alist,l)
     else
       let y = CDR y in
         lookup_aux (exp,x,y,z,stack,alist,l))

(lisp_lookup (exp,x,y,z,stack,alist,l) =
   let y = alist in
   let (exp,x,y,z,stack,alist,l) = lookup_aux (exp,x,y,z,stack,alist,l) in
     (exp,x,y,z,stack,alist,l))

(zip_yz (exp,x,y,z,stack,alist,l) =
   if isDot y then
     let alist = exp in
     let exp = CAR y in
     let x = CAR z in
     let exp = Dot exp x in
     let x = alist in
     let exp = Dot exp x in
     let y = CDR y in
     let z = CDR z in
       zip_yz (exp,x,y,z,stack,alist,l)
   else
     (exp,x,y,z,stack,alist,l))

(lisp_length (exp,x,y,z,stack,alist,l) =
   if isDot x then
     let exp = LISP_ADD exp (Val 1) in
     let x = CDR x in
```

```
       lisp_length (exp,x,y,z,stack,alist,l)
   else
     (exp,x,y,z,stack,alist,l))

(lisp_less (exp,x,y,z,stack,alist,l) =
   if getVal exp < getVal x
   then let exp = Sym "t" in (exp,x,y,z,stack,alist,l)
   else let exp = Sym "nil" in (exp,x,y,z,stack,alist,l))

(lisp_symbolp (exp,x,y,z,stack,alist,l) =
   if isSym exp
   then let exp = Sym "t" in (exp,x,y,z,stack,alist,l)
   else let exp = Sym "nil" in (exp,x,y,z,stack,alist,l))

(lisp_consp (exp,x,y,z,stack,alist,l) =
   if isDot exp
   then let exp = Sym "t" in (exp,x,y,z,stack,alist,l)
   else let exp = Sym "nil" in (exp,x,y,z,stack,alist,l))

(lisp_less (exp,x,y,z,stack,alist,l) =
   if getVal exp < getVal x
   then let exp = Sym "t" in (exp,x,y,z,stack,alist,l)
   else let exp = Sym "nil" in (exp,x,y,z,stack,alist,l))

(lisp_atomp (exp,x,y,z,stack,alist,l) =
   if isDot exp
   then let exp = Sym "nil" in (exp,x,y,z,stack,alist,l)
   else let exp = Sym "t" in (exp,x,y,z,stack,alist,l))

(lisp_numberp (exp,x,y,z,stack,alist,l) =
   if isDot exp
   then let exp = Sym "nil" in (exp,x,y,z,stack,alist,l)
   else if isSym exp
   then let exp = Sym "nil" in (exp,x,y,z,stack,alist,l)
   else let exp = Sym "t" in (exp,x,y,z,stack,alist,l))

(lisp_add (exp,x,y,z,stack,alist,l) =
   if isDot y then
     let x = CAR y in
     let y = CDR y in
     let exp = LISP_ADD exp x in
       lisp_add (exp,x,y,z,stack,alist,l)
   else
     (exp,x,y,z,stack,alist,l))

(lisp_mult (exp,x,y,z,stack,alist,l) =
   if isDot z then
     let x = CAR z in
     let z = CDR z in
     let (exp,x,y) = (LISP_MULT exp x,Sym "nil",Sym "nil") in
       lisp_mult (exp,x,y,z,stack,alist,l)
   else
     (exp,x,y,z,stack,alist,l))

(lisp_func (exp,x,y,z,stack,alist,l) =
   if isDot x then
     let y = CDR x in
     let x = CAR x in
       if x = Sym "lambda" then
         let z = exp in      (* z := evaluated args   *)
         let x = exp in
         let exp = Val 0 in
         let (exp,x,y,z,stack,alist,l) = lisp_length (exp,x,y,z,stack,alist,l) in
         let x = stack in
         let exp = Dot exp x in
```

```
      let x = exp in
      let exp = CDR y in  (* exp := body of lambda *)
      let exp = Dot exp x in
      let stack = exp in
      let y = CAR y in    (* y := parameter names  *)
      let exp = alist in
      let x = y in
      let (exp,x,y,z,stack,alist,l) = zip_yz (exp,x,y,z,stack,alist,l) in
      let alist = exp in
      let exp = CAR stack in
      let stack = CDR stack in
      let exp = CAR exp in
      let z = TASK_EVAL in
        (exp,x,y,z,stack,alist,l)
    else (* if x = Sym "label" *)
      let z = exp in
      let exp = Val 1 in
      let x = stack in
      let exp = Dot exp x in
      let stack = exp in
      let x = CDR y in
      let exp = CAR y in
      let x = CAR x in
      let exp = Dot exp x in
      let x = alist in
      let exp = Dot exp x in
      let x = CDR y in
      let alist = exp in
      let exp = z in
      let x = CAR x in
      let z = TASK_FUNC in
        lisp_func (exp,x,y,z,stack,alist,l)
else (* x must be a symbol *)
  let z = TASK_CONT in
    if x = Sym "car" then
      let exp = CAR exp in
      let exp = CAR exp in
        (exp,x,y,z,stack,alist,l)
    else if x = Sym "cdr" then
      let exp = CAR exp in
      let exp = CDR exp in
        (exp,x,y,z,stack,alist,l)
    else if x = Sym "cons" then
      let x = CDR exp in
      let exp = CAR exp in
      let x = CAR x in
      let exp = Dot exp x in
        (exp,x,y,z,stack,alist,l)
    else if x = Sym "+" then
      let y = exp in
      let exp = Val 0 in
      let (exp,x,y,z,stack,alist,l) = lisp_add (exp,x,y,z,stack,alist,l) in
        (exp,x,y,z,stack,alist,l)
    else if (x = Sym "-") then
      let x = CDR exp in
      let exp = CAR exp in
      let x = CAR x in
      let exp = LISP_SUB exp x in
        (exp,x,y,z,stack,alist,l)
    else if (x = Sym "*") then
      let z = exp in
      let exp = Val 1 in
      let (exp,x,y,z,stack,alist,l) = lisp_mult (exp,x,y,z,stack,alist,l) in
      let z = TASK_CONT in
        (exp,x,y,z,stack,alist,l)
```

```
        else if (x = Sym "div") then
          let x = CDR exp in
          let exp = CAR exp in
          let x = CAR x in
          let (exp,x,y) = (LISP_DIV exp x,Sym "nil",Sym "nil") in
            (exp,x,y,z,stack,alist,l)
        else if (x = Sym "mod") then
          let x = CDR exp in
          let exp = CAR exp in
          let x = CAR x in
          let (exp,x,y) = (LISP_MOD exp x,Sym "nil",Sym "nil") in
            (exp,x,y,z,stack,alist,l)
        else if (x = Sym "<") then
          let x = CDR exp in
          let exp = CAR exp in
          let x = CAR x in
          let (exp,x,y,z,stack,alist,l) = lisp_less (exp,x,y,z,stack,alist,l) in
            (exp,x,y,z,stack,alist,l)
        else if (x = Sym "atomp") then
          let exp = CAR exp in
          let (exp,x,y,z,stack,alist,l) = lisp_atomp (exp,x,y,z,stack,alist,l) in
            (exp,x,y,z,stack,alist,l)
        else if (x = Sym "consp") then
          let exp = CAR exp in
          let (exp,x,y,z,stack,alist,l) = lisp_consp (exp,x,y,z,stack,alist,l) in
            (exp,x,y,z,stack,alist,l)
        else if (x = Sym "numberp") then
          let exp = CAR exp in
          let (exp,x,y,z,stack,alist,l) = lisp_numberp (exp,x,y,z,stack,alist,l) in
            (exp,x,y,z,stack,alist,l)
        else if (x = Sym "symbolp") then
          let exp = CAR exp in
          let (exp,x,y,z,stack,alist,l) = lisp_symbolp (exp,x,y,z,stack,alist,l) in
            (exp,x,y,z,stack,alist,l)
        else if (x = Sym "equal") then
          let x = CDR exp in
          let exp = CAR exp in
          let x = CAR x in
          let exp = LISP_EQUAL exp x in
            (exp,x,y,z,stack,alist,l)
        else (* if none of the above, then lookup in alist and repeat *)
          let z = exp in
          let exp = x in
          let (exp,x,y,z,stack,alist,l) = lisp_lookup (exp,x,y,z,stack,alist,l) in
          let x = exp in
          let exp = z in
          let z = TASK_FUNC in
            lisp_func (exp,x,y,z,stack,alist,l))

(rev_exp (exp,x,y,z,stack,alist,l) =
   if isDot z then
     let x = exp in
     let exp = CAR z in
     let exp = Dot exp x in
     let z = CDR z in
       rev_exp (exp,x,y,z,stack,alist,l)
   else
     let x = y in
       (exp,x,y,z,stack,alist,l))

(reverse_exp (exp,x,y,z:SExp,stack,alist,l) =
   let z = exp in
   let exp = Sym "nil" in
   let (exp,x,y,z,stack,alist,l) = rev_exp (exp,x,y,z,stack,alist,l) in
   let z = TASK_FUNC in
```

```
     (exp,x,y,z,stack,alist,l))

(repeat_cdr (exp,x,y,z,stack,alist,l) =
   if x = Val 0 then (exp,x,y,z,stack,alist,l) else
     let alist = CDR alist in
     let x = LISP_SUB x (Val 1) in
       repeat_cdr (exp,x,y,z,stack,alist,l))

(lisp_cont (exp,x,y,z,stack,alist,l) =
   let x = CAR stack in
   let stack = CDR stack in
     if ~isDot x then (* drop elements from alist *)
       let (exp,x,y,z,stack,alist,l) = repeat_cdr (exp,x,y,z,stack,alist,l) in
         (exp,x,y,z,stack,alist,l)
     else
       let z = x in
       let x = CAR stack in
         if x = Sym "cond" then (* deal with conditional *)
           let stack = CDR stack in
             if exp = Sym "nil" then (* guard is false *)
               let exp = x in
               let x = CDR z in
               let exp = Dot exp x in
               let z = TASK_EVAL in
                 (exp,x,y,z,stack,alist,l)
             else (* guard is true *)
               let exp = CAR z in
               let exp = CDR exp in
               let exp = CAR exp in
               let z = TASK_EVAL in
                 (exp,x,y,z,stack,alist,l)
         else
           let y = CAR z in (* list of unevaluated args *)
           let x = CDR z in (* list of evaluated args *)
             if isDot y then (* still args to evaluate *)
               let z = CAR y in
               let exp = Dot exp x in
               let x = exp in
               let exp = CDR y in
               let exp = Dot exp x in
               let x = stack in
               let exp = Dot exp x in
               let stack = exp in
               let exp = z in
               let z = TASK_EVAL in
                 (exp,x,y,z,stack,alist,l)
             else
               let y = CAR stack in
               let stack = CDR stack in
               let exp = Dot exp x in
               let (exp,x,y,z,stack,alist,l) = reverse_exp (exp,x,y,z,stack,alist,l) in
                 (exp,x,y,z,stack,alist,l))

(lisp_app (exp,x,y,z,stack,alist,l) =
   if x = Sym "quote" then
     let exp = CAR exp in
       (exp,x,y,z,stack,alist,l)
   else if x = Sym "cond" then
     if isDot exp then
       let z = exp in
       let exp = x in
       let x = stack in
       let exp = Dot exp x in
       let x = exp in
       let exp = z in
```

```
        let exp = Dot exp x in
        let stack = exp in
        let exp = CAR z in
        let exp = CAR exp in
        let z = TASK_EVAL in
          (exp,x,y,z,stack,alist,l)
      else
        (exp,x,y,z,stack,alist,l)
    else (* normal function: push function onto stack, push args onto stack *)
      if isDot exp then (* there is at least one arg *)
        let y = CAR exp in
        let z = CDR exp in
        let exp = x in
        let x = stack in
        let exp = Dot exp x in
        let stack = exp in
        let exp = z in
        let x = Sym "nil" in
        let exp = Dot exp x in
        let x = stack in
        let exp = Dot exp x in
        let stack = exp in
        let exp = y in
        let z = TASK_EVAL in
          (exp,x,y,z,stack,alist,l)
      else (* there are no args *)
        let z = TASK_FUNC in
          (exp,x,y,z,stack,alist,l))

(lisp_eval (exp,x,y,z,stack,alist,l) =
    if z = TASK_EVAL then
      let z = TASK_CONT in
      if isSym exp then (* exp is Sym *)
        let (exp,x,y,z,stack,alist,l) = lisp_lookup (exp,x,y,z,stack,alist,l) in
          lisp_eval (exp,x,y,z,stack,alist,l)
      else if isDot exp then (* exp is Dot *)
        let x = CAR exp in
        let exp = CDR exp in
        let (exp,x,y,z,stack,alist,l) = lisp_app (exp,x,y,z,stack,alist,l) in
            lisp_eval (exp,x,y,z,stack,alist,l)
      else (* exp is Num *)
        lisp_eval (exp,x,y,z,stack,alist,l)
    else if z = TASK_FUNC then (* function=x, args stored as list in exp *)
      let (exp,x,y,z,stack,alist,l) = lisp_func (exp,x,y,z,stack,alist,l) in
        lisp_eval (exp,x,y,z,stack,alist,l)
    else (* if z = TASK_CONT then *)
      if isDot stack then (* something is still on the to-do list *)
        let (exp,x,y,z,stack,alist,l) = lisp_cont (exp,x,y,z,stack,alist,l) in
          lisp_eval (exp,x,y,z,stack,alist,l)
      else (* something is still on the to-do list *)
        (exp,x,y,z,stack,alist,l))
```

Bibliography

[1] Andrew W. Appel. Foundational proof-carrying code. In *Logic in Computer Science (LICS)*. IEEE Computer Society, 2001.

[2] Michael A. Arbib and Suad Alagic. Proof rules for gotos. *Acta Informatica*, 11:139–148, 1979.

[3] Ralph-Johan Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, Åbo Akademi, Department of Computer Science, Finland, 1978. Report A–1978–4.

[4] Ralph-Johan Back and Joakim von Wright. Refinement calculus I: Sequential nondeterministic programs. Reports on Computer Science and Mathematics 92, Åbo Akademi, 1989.

[5] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998. Graduate Texts in Computer Science.

[6] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Principles of Programming Languages (POPL)*. ACM, 2002.

[7] Nick Benton. Abstracting allocation: The new new thing. In *Computer Science Logic (CSL)*, Computer Science Logic. Springer, 2006.

[8] Nick Benton. Machine obstructed proof (abstract). In *Workshop on Mechanizing Metatheory*, 2006.

[9] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects (FMCO)*. Springer, 2005.

[10] Yves Bertot. A short presentation of Coq. In *Theorem Proving in Higher Order Logics (TPHOLs)*. Springer, 2008.

[11] William Bevier, Warren Hunt, J Strother Moore, and William Young. Special issue of system verification. *Journal of Automated Reasoning*, 5(4), 1989.

[12] William R. Bevier. *A verified operating system kernel*. PhD thesis, University of Texas at Austin, 1987.

[13] L. Birkedal, N. Torp-Smith, and J. Reynolds. Local reasoning about a copying garbage collector. In *Principles of Programming Languages (POPL)*. ACM, 2004.

[14] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. TCP, UDP, and sockets: rigorous and experimentally-validated behavioural specification. Volume 2: The specification. Technical Report UCAM–CL–TR–624, University of Cambridge, 2005.

[15] R. S. Boyer and J S. Moore. Proving theorems about pure LISP fucntions. *J. ACM*, 22(1):129–144, 1975.

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

[17] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[18] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. In *Programming Language Design and Implementation (PLDI)*. ACM, 2007.

[19] Juanito Camilleri and Tom Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, Computer Laboratory, University of Cambridge, August 1992.

[20] The FLINT Group. Yale University. `http://flint.cs.yale.edu/`.

[21] Juan Chen, Dinghao Wu, Andrew W. Appel, and Hai Fang. A provably sound TAL for back-end optimization. In *Programming Language Design and Implementation (PLDI)*. ACM, 2003.

[22] C. J. Cheney. A non-recursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, 1970.

[23] Adam J. Chlipala. Modular development of certified program verifiers with a proof assistant. In *International Conference on Functional Programming (ICFP)*. ACM, 2006.

[24] Adam J. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Programming Language Design and Implementation (PLDI)*. ACM, 2007.

[25] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

[26] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV)*. Springer, 2000.

[27] D. L. Clutterbuck and B. A. Carré. The verification of low-level code. *Software Engineering Journal*, 3:97–111, 1988.

[28] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering (ICSE)*, 2000.

[29] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*. ACM, 1977.

[30] Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. Technical Report CMU-CS-03-108, Carnegie Mellon University, 2003.

[31] Zaynah Dargaye and Xavier Leroy. Mechanized verification of CPS transformations. In *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*. Springer, 2007.

[32] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[33] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Inc., 1976.

[34] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[35] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3):241–266, 1982.

[36] Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *Programming Language Design and Implementation (PLDI)*. ACM, 2008.

[37] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *J. Funct. Program.*, 13(4):709–745, 2003.

[38] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *Principles of Programming Languages (POPL)*. ACM, 2001.

[39] R. W. Floyd. Assigning meanings to programs. In *Symposia in Applied Mathematics (Vol XIX)*, pages 19–32. American Mathematical Society, 1967.

[40] Anthony Fox. Formal specification and verification of ARM6. In David Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*. Springer, 2003.

[41] Herman H. Goldstine and John von Neumann. Planning and coding problems for an electronic computing instrument. In *John von Neumann, Collected Works*, volume V, pages 34–235. Pergamon Press, Oxford, 1961.

[42] Georges Gonthier. Verifying the safety of a practical concurrent garbage collector. In *Computer Aided Verification (CAV)*. Springer, 1996.

[43] M. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. Springer, 1 edition, January 1980.

[44] Michael J. C. Gordon. Mechanizing programming logics in higher order logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer, 1989.

[45] Mike Gordon. Defining a LISP interpreter in a logic of total functions. In *the ACL2 Theorem Prover and Its Applications (ACL2)*, 2007.

[46] Mike Gordon, Juliano Iyoda, Scott Owens, and Konrad Slind. Automatic formal synthesis of hardware from higher order logic. *Electr. Notes Theor. Comput. Sci.*, 145:27–43, 2006.

[47] David Greve, Raymond Richards, and Matthew Wilding. A summary of intrinsic partitioning verification. In *ACL2 Theorem Prover and Its Applications (ACL2)*, 2004.

[48] David Gries. An exercise in proving parallel programs correct. *Commun. ACM*, 20(12):921–930, 1977.

[49] Joshua Guttman, John Ramsdell, and Mitchell Wand. Vlisp: A verified implementation of scheme. *Lisp and Symbolic Computation*, 8(1/2):5–32, 1995.

[50] David S. Hardin, Eric W. Smith, and William D. Young. A robust machine code proof framework for highly secure applications. In Panagiotis Manolios and Matthew Wilding, editors, *ACL2 Theorem Prover and Its Applications (ACL2)*, 2006.

[51] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[52] Peter V. Homeier and David F. Martin. A mechanically verified verification condition generator. *Comput. J.*, 38(2):131–141, 1995.

[53] Samin S. Ishtiaq and Peter W. O'Hearn. Bi as an assertion language for mutable data structures. In *Principles of Programming Languages (POPL)*, New York, NY, USA, 2001. ACM.

[54] Shin-ya Katsumata and Atsushi Ohori. Proof-directed de-compilation of low-level code. In *European Symposium on Programming (ESOP)*, UK, 2001. Springer.

[55] M. Kaufmann, R. S. Boyer, and J Moore. The Boyer-Moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*, 29(2):27–62, 1995.

[56] Matt Kaufmann and J. Strother Moore. An ACL2 tutorial. In *Theorem Proving in Higher Order Logics (TPHOLs)*. Springer, 2008.

[57] James Cornelius King. *A program verifier*. PhD thesis, Pittsburgh, PA, USA, 1969.

[58] Gerwin Klein and Tobias Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.

[59] C. League and V. Trifonov Z. Shao. Precision in practice: A type-preserving Java compiler. Technical Report YALEU/DCS/TR-1223, Department of Computer Science, Yale University, 2002.

[60] K. Rustan M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.

[61] K. Rustan M. Leino and Greg Nelson. An extended static checker for modular-3. In *Compiler Construction (CC)*. Springer, 1998.

[62] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Programming Language Design and Implementation (PLDI)*, 2002.

[63] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Principles of Programming Languages (POPL)*. ACM, 2006.

[64] Guodong Li, Scott Owens, and Konrad Slind. A proof-producing software compiler for a subset of higher order logic. In *European Symposium on Programming (ESOP)*. Springer, 2007.

[65] Guodong Li and Konrad Slind. Compilation as rewriting in higher order logic. In *International Conference on Automated Deduction (CADE)*. Springer, 2007.

[66] Guodong Li and Konrad Slind. Trusted source translation of a total function language. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2008.

[67] Hanbing Liu and J Strother Moore. Java program verification via a JVM deep embedding in ACL2. In *Theorem Proving in Higher Order Logics (TPHOLs)*. Springer, 2004.

[68] V. M. Luchangco. *Memory consistency models for high-performance distributed computing.* PhD thesis, MIT, 2001.

[69] Panagiotis Manolios and J. Strother Moore. Partial functions in ACL2. *J. Autom. Reasoning*, 31(2):107–127, 2003.

[70] John Matthews, J. Strother Moore, Sandip Ray, and Daron Vroon. Verification condition generation via theorem proving. In *Logic Programming and Automated Reasoning (LPAR)*, 2006.

[71] W. D. Maurer. Proving the correctness of a flight-director program for an airborne minicomputer. In *ACM SIGMINI/SIGPLAN interface meeting on Programming systems in the small processor environment*. ACM, 1976.

[72] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. 1987.

[73] John McCarthy. Towards a mathematical science of computation. In *International Federation for Information Processing Congress (IFIP Congress)*, 1962.

[74] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual*. The MIT Press, 1966.

[75] John McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In J. T. Schwartz, editor, *Symposium in Applied Mathematics, Mathematical Aspects of Computer Science*. American Mathematical Society, 1967.

[76] Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. A general framework for certifying garbage collectors and their mutators. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Programming Language Design and Implementation (PLDI)*. ACM, 2007.

[77] Thomas Meyer and Burkhart Wolff. Tactic-based optimized compilation of functional programs. In *Types for Proofs and Programs (TYPES)*. Springer, 2004.

[78] J Strother Moore. Symbolic simulation: An ACL2 approach. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 334–350, 1998.

[79] J. Strother Moore. Inductive assertions and operational semantics. In *Correct Hardware Design and Verification Methods (CHARME)*. Springer, 2003.

[80] Carroll Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10(3):403–419, 1988.

[81] Carroll Morgan. *Programming from specifications*. Prentice-Hall, Inc., USA, 1990.

[82] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Workshop on Compiler Support for System Software*, 1999.

[83] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Principles of Programming Languages (POPL)*. ACM, 1998.

[84] Magnus O. Myreen. Verification of LISP interpreters. In *TPHOLs Emerging Trends*. Department of Electrical Engineering and Computer Science, University of Concordia, 2008.

[85] Magnus O. Myreen. Verified implementation of LISP on ARM, x86 and PowerPC. In *Theorem Proving in Higher-Order Logics (TPHOLs)*. Springer, 2009.

[86] Magnus O. Myreen, Anthony C.J. Fox, and Michael J.C. Gordon. A Hoare logic for ARM machine code. In *Fundamentals of Software Engineering (FSEN)*. Springer, 2007.

[87] Magnus O. Myreen and Michael J. C. Gordon. Verification of machine code implementations of arithmetic functions for cryptography. In *TPHOLs Emerging Trends, Report 367/07*. Department of Computer Science, University of Kaiserslautern, 2007.

[88] Magnus O. Myreen and Michael J.C. Gordon. A Hoare logic for realistically modelled machine code. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2007.

[89] Magnus O. Myreen and Michael J.C. Gordon. Transforming programs into recursive functions. In *Brazilian Symposium on Formal Methods (SBMF)*. Elsevier, 2008.

[90] Magnus O. Myreen, Konrad Slind, and Michael J. C. Gordon. Machine-code verification for multiple architectures – An application of decompilation into logic. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2008.

[91] Magnus O. Myreen, Konrad Slind, and Michael J.C. Gordon. Extensible proof-producing compilation. In *Compiler Construction (CC)*. Springer, 2009.

[92] George C. Necula. Proof-carrying code. In *Principles of Programming Languages (POPL)*. ACM, 1997.

[93] George C. Necula. Translation validation for an optimizing compiler. In *Programming Language Design and Implementation (PLDI)*. ACM, 2000.

[94] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[95] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. *ACM SIGPLAN Notices*, 41(1):320–333, January 2006.

[96] Zhaozhong Ni, Dachuan Yu, and Zhong Shao. Using XCAP to certify realistic systems code: Machine context management. In *Theorem Proving in Higher Order Logics (TPHOLs)*. Springer, 2007.

[97] Sam Owre and Natarajan Shankar. A brief overview of PVS. In *Theorem Proving in Higher Order Logics (TPHOLs)*. Springer, 2008.

[98] Matthew J. Parkinson, Richard Bornat, and Cristiano Calcagno. Variables as resource in Hoare logics. In *Logic in Computer Science (LICS)*. IEEE Computer Society, 2006.

[99] Lee Pike, Mark Shields, and John Matthews. A verifying core for a cryptographic language compiler. In *the ACL2 Theorem Prover and its Applications (ACL2)*. HappyJack Books, 2006.

[100] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. Springer, 1998.

[101] Wolfgang Polak. *Compiler Specification and Verification*. Springer, 1981.

[102] Certified Program Verification with Coq. `http://proofos.sourceforge.net/`.

[103] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*. Springer, 1982.

[104] Proceedings of the Working Conference on Reverse Engineering. IEEE. 1995–.

[105] John Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*. IEEE Computer Society, 2002.

[106] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*. Palgrave, 2000.

[107] David M. Russinoff. A mechanically verified incremental garbage collector. *Formal Asp. Comput.*, 6(4):359–390, 1994.

[108] Ando Saabas and Tarmo Uustalu. A compositional natural semantics and Hoare logic for low-level languages. *Electr. Notes Theor. Comput. Sci*, 156(1):151–168, 2006.

[109] Susmit Sarkar, Pater Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *Principles of Programming Languages (POPL)*. ACM, 2009.

[110] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Theorem Proving in Higher Order Logics (TPHOLs)*. Springer, 2008.

[111] Typed Assembly Language Compiler. `http://www.cs.cornell.edu/talc/`.

[112] Gang Tan and Andrew W. Appel. A compositional logic for control flow. In E. Allen Emerson and Kedar S. Namjoshi, editors, *Verification, Model Checking and Abstract Interpretation (VMCAI)*. Springer, 2006.

[113] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: a case study on instruction scheduling optimizations. In *Principles of Programming Languages (POPL)*. ACM, 2008.

[114] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *Principles of Programming Languages (POPL)*. ACM, 2007.

[115] Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. University Mathematical Laboratory, Cambridge, 1949.

[116] Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In *Concurrency Theory (CONCUR)*. Springer, 2007.

[117] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2001.

[118] David von Oheimb and Tobias Nipkow. Machine-checking the java specification: Proving type-safety. In *Formal Syntax and Semantics of Java*, 1999.

[119] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle framework. In *Theorem Proving in Higher Order Logics (TPHOLs)*. Springer, 2008.

[120] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In *Computer Aided Verification (CAV)*. Springer, 2008.